

Charting Java

Mary Cosway and Andrea di Pietro

August 26, 2002

Special Thanks for the support given by Vincent and Jerome.

Copyright © by Haiko Schmarsow. All rights reserved. This publication or any part of it may not be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without being able to present the prior written permission issued by the owner of the copyright. The copyright and the statements in the following paragraphs extend non-exclusively to the downloadable data offered under the domain-names www.chartingJava.org and www.chartingJava.com. Data from these sites may be downloaded solely for private and non-commercial use.

The use of general descriptive names, registered names, trade names, trademarks, etc, in this publication or the associated Internet-sites, even if any of the former are not especially identified, does not imply that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

All names and services identified throughout this book or the associated Internet-sites, also those which are, or which are related, to trademarks or registered trademarks of their respective companies, are not used to convey endorsement or other affiliation with this publication.

This publication and the associated Internet-sites are provided "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. The publisher or owner of the copyright assumes no responsibility of any kind for errors or omissions and ensuing damages resulting from the use of the information contained in this publication or contained in the associated Internet-sites.

Written with LyX, the latest version used was 1.1.6fix2, figures and images were realized using Xfig 3.2 and Gimp Version 1.2 on a Linux/KDE platform (SuSE 7.2).

Publisher, holder of copyright and contact for feedback is (September 2002) Haiko Schmarsow (haiko@schmarsow.de)

ISBN 3-00-009429-6

Contents

Preface	17
This Book	18
Structure	18
Content	18
Limitations	20
The Reader	20
The Authors	21
Conventions	21
 I. Computer-Models	 23
Elementary Computer-Models	27
A Computer-Model for Programmers	28
Computers with Multiple Processors and Networks	30
Summary: Computer-Models	30
A Computer-Model for Hardware Technicians	30
 Processor-Commands, Languages	 33
Bits and Bytes used For Coding Human Readable Symbols and Numbers	33
Commands on Processor-Level: Machine-Language and Assembler	35
 Languages for easier Coding	 37
Language-Processing: Interpreting, Compiling, Virtual Machine	38
Languages: High-Level-Concepts	40
 Operating-Systems	 43
File-Systems	46
 II. The Java Runtime Environment	 49
Installing Java Development Kits (JDKs)	59
Preparations: Generalities on (Downloading) and Installation	59
PATH and CLASSPATH for Unix/Linux or Windows-based Operating-Systems	60
ORGANIZE: Apple Mac OS	61
ORGANIZE: Mac OS 9.x: Installing the Java Development Kit (JDK):	61
ORGANIZE: Mac OS 9.x: Using the Java Development Kit (JDK)	61
ORGANIZE: BeOS	62
ORGANIZE: Linux	62
ORGANIZE: Linux: Getting the Java Development Kit (JDK):	62
ORGANIZE: Linux: Using the Java Development Kit (JDK):	62
ORGANIZE: Windows	63
ORGANIZE: Windows: Installing the Java Development Kit (JDK):	63

ORGANIZE: Windows: Using the Java Development Kit (JDK):	63
ORGANIZE: Development-Environments	64
Organize: Extending Java	64
How Java Code is Run	65
Compiling Source-Code	65
Running Programs/Applications	65
Running Applets	67
 III. Basic Programming I	
The Language and Utilities	69
Hello World Program	71
 Data-Storage — Data-Structures	73
Concept: Literal, Type, Name and Variable	73
Data in Form of Literals — What is stored in the computer memory?	73
Type or Class of Literals — How is it stored into the computer memory?	73
Identifier, Name — Where is it stored in the computer memory?	73
Concept: Declaration, Allocation, Instantiation of a Variable	74
Concept: Statically-, Dynamically-, Runtime-Typed Programming-Languages	75
Java: Declaration, Allocation, Instantiation: Examples and Outlook	75
Java: Literals	77
Integer-Literals	77
Floating-Point-Literals	78
Boolean-Literals	79
Character-Literals	79
String-Literals	81
Java: Identifier, Name	81
Naming Conventions	83
Java: Primitive-Types	83
Logical Type	84
Numeric Type	84
Integral Types	85
Floating-Point Type	86
Java: Declaration, Instantiation of Variables with Primitive-Type	87
Java: Type-Casting I	87
 Processing Data	89
Concept: Expressions, Statements, Control Structures	89
Java: Expressions	90
Operator Expressions — Calculate, Compare and Change	92
Increment/Decrement and Prefix Operators	92
Arithmetic: Multiplicative Operators	93
Arithmetic: Additive Operators	94
Arithmetic: Roots, Exponentials, Logarithms and Powers	94
Non-Arithmetic: String Concatenation	95
Bitwise: Shift-Operators	95
Non-Arithmetic: Relational Expression	96
Un/Equality: Comparison Operators	96
Bitwise and Logical Operators (Single Character: &, ^,)	97

Logical Operators (Double Characters: &&,)	98
Conditional-Operator	99
Assignment Operators	99
Type-Casting II (Casting of Primitive-Types)	100
Type-Casting implicit	100
Type-Casting explicit	100
Precedence	101
Java: Statements, Control Structures and Programs	103
Java: Statements	103
Block	105
Java: Branching	105
Binary Branching — If-Statement	105
Multiple Branching/Cascading — Switch-Statement	106
Java: Looping, Iteration	108
Post-condition Loops — While, For-Statements	108
Pre-condition Loops — Do-Statement	109
Restricted Jumping — Break, Continue-Statements — Pre- and Post-Condition Mix	110
Restricted Jumping — Try-Catch-Finally	112
Structuring Code: Proceduralization	115
Concept: Methods are more than Named Blocks of Code	115
Java: Methods	116
Java: Method-Declaration	117
Java: Method-Call	118
Concept: Method-Call Arguments — Transfer By-Value or By-Reference	119
Java: Local Variables - Scoping and Shadowing 1	121
Scope	122
Shadowing Variables with Methods	123
Concept and Java: Overloading, Recursion, Callback	124
Overloading Methods	124
Recursion of Method-Calls	125
Callback of Methods	127
Concept: Triggerable-Methods Or Who calls the main()-Method of an Application?	127
Summary and Outlook: Proceduralization	128
Structuring Code and Data: Object-Orientation	131
Concept: Object/Class-Model	131
Example: Object, Class and Class-Hierarchy	132
Overriding Methods and Shadowing Variables	135
Class-Extension — Inheritance with multiple Super-Classes	136
Communication-Mechanisms	137
Object/Class-Model Keywords	138
Concept: Object/Class-Model within Programming-Languages	140
Java: Classes as Programmer-Definable Types and Programs	142
Java: Reference-Types and their Instantiation with the new-Keyword	142
Java: Classes	145
Java: Class-Declaration	145
Java: Class-Instantiation/Object-Creation	149
Java: Defining Constructors	149
Java: Class-Instantiation with Constructors and Factory-Methods	150

Java: Class- and Object Lifespan	151
Java: Introducing and Accessing Class- and Instance-Members	154
Java: Kinds of Classes	157
Java: Anonymous Objects and -Classes	159
Singleton	160
Comparing Objects	161
Java: Identity of Objects	161
Concept: Equality of Objects	162
Java: Comparing Objects (1)	162
Java: Class-Extension	163
The Class <code>java.lang.Object</code>	164
Class-Extension: Member-Access	165
Java: Class-Extension with Constructors and Finalizers	166
Java: Overriding Methods and Shadowing Variables (2)	167
Java: Type-Casting III (Casting of Reference-Types)	169
Object to Primitive-Type and Reverse	169
Type-Casting among Identifiers of Reference-Type: Widening and Narrowing	169
Java: Validity of Identifiers: Scoping and Access-Control	171
Un-Shadowing and Un-Overriding	172
Un-Shadowing and Un-Overriding FROM INSIDE AN OBJECT	172
Un-Shadowing and Un-Overriding FROM OUTSIDE AN OBJECT	174
Access-Control	175
Modifiers and Accessibility	180
Java: Other Modifiers	181
Java: Class-Extension with Abstract Classes and Interfaces	183
Java: Abstract Classes and Abstract Methods	183
Java: Interface-Type	185
Java: Comparing (2) and Copying	190
Summary: Object-Orientation and Java Classes	193
Java and the Class-Concept	194
Concept: Polymorphism/Monomorphism	194
Concept: Encapsulation	195
Structuring Code: Multithreading	197
Concept: Concurrency and Parallel Computing	197
Concept: Threads	198
Concept: Scheduling of Threads (Non-/Preemptive and Priorities)	200
Java: Basic Thread Resource Distribution on a Java Platform	203
Java: Handling Single/Isolated Threads	205
Java: Spawning a Thread	206
Thread-Object contains <code>run()</code> -Method	207
<code>run()</code> -Method in Anonymous Class	208
Thread-Object and <code>run()</code> -Method in Different Classes	209
Thread-Object is Member in Class of <code>run()</code> -Method	209
Basics of Controlling a Single Thread	210
Concept: Handling Multiple/Communicating Threads	213
Concept: Atomicity and Exclusive Access (Locking)	213
Concept: Waiting	215
Java: Handling Multiple/Communicating Threads	216
Java: Isolating Code and Data from other Threads	216

Java: Waiting for other Threads	220
Java: Daemon-Threads And Shutting Down of JRE	223
Problems specifically arising with Thread-Communication:	224
Dangers: Starvation	224
Dangers: Race-Conditions	225
Dangers: Deadlock	226
Dangers: Lifelock	226
Concept: Pragmatics in Thread-Specific Programming	227
How can Threads interact elementary on a Level above Scheduling?	228
A Multithreaded Application	228
Concept: Multiprocessing and Multithreading	230
Concept: Blocking and Non-Blocking Methods	231
Summary: Multithreading	232
Concept and Java: Aggregation of Data	233
Java: Arrays and Array-Type	234
Java: Vectors	236
Java: StringTokenizer	237
Java: Collection-Framework	237
Transforming Aggregates	237
Structuring Code: Packaging	239
Java: How to Write and How to Install a Package	240
How to install a pre-written Package	243
How to use an installed Package	244
Package Introspection	245
Compilation, Packages and Compilation Units	246
Summary: Packages	246
Standard Code-Libraries: Pre-programmed Packages	249
Pre-programmed Packages	249
Core-Classes and the Java Runtime System: Location/Content	249
java.lang: Some Standard Classes	252
Platform and Runtime Environment	258
Internationalization - Localization	259
Identifying Locality with java.util.Locale-Objects	260
Example: Calendar and TimeZone	261
Example: Resource-Bundle	263
Character Codes and Fonts	265
Java: Encodings and Fonts	266
IV. Basic Programming II	
Internal Communication	267
Java: Communication: Introducing Classes across Files of Source-Code	269
Communication: Messaging and Streaming	271
Concept: Basic Communication-Models	271
Message-Model	271
Stream-Model	272
Operating-System/Runtime System: Sockets for Input and Output	273
Protocol-Model	274
Initial Triggering	275
Practical Event-driven Communication	277

Code Structuring: Messaging	279
Java: Implemented: Communication by “Throwing Objects”	279
Source: Who throws How from Where?	279
What is thrown?	280
What’s the Switchboard-Mechanism and Who catches?	282
Summary: “Throwing Objects”	284
Java: To be Implemented: Observable-Class and Observer-Interface	284
Java: Event Model — A Mechanism fully to be Implemented	286
Source	287
Event	288
Drain	288
Trigger	288
Trivial Example	288
Remarks about Non-Trivial Extensions	289
Concept: Streams in Java	291
Java: Streams	295
Java: Runtime Input/Output-Streams	297
Java: Piping with Streams	300
Java: More on Java Streams	301
Java: Filtering and Formatting with Streams	303
Java: Serialization of Objects, Persistence	306
Summary: Streams	310
Code-Structure: Software-Components	311
Concept: Software-Components	311
Concept: Object-Oriented Approach to Software-Components	311
Java-Beans: Software-Components in Java	313
Standard Property	313
Indexed Property	313
Event-Communication between Software-Components	314
Bound and Constrained Properties	314
Persistence (of Objects)	318
Software-Design using Components with Reflection and Introspection	318
Summary: Software-Components	318
Persistence	321
Documenting and Archiving Code	323
Comments and Documentation	323
Java: javadoc	324
Java: jar-Tool for Archiving	325
Java: Running Java Byte-Code	326
Summary: Java Messaging	329
Thread-Communication	329
Exception Throwing	329
Observable/Observer-Mechanism	330
Framework: java.util.EventObject, java.util.EventListener	331
Java-Beans — Software-Components	331
Abstract Windowing Toolkit (AWT) and Swing	331

Summary: Java Language	333
V. Communication between Computers Networks and Distributed Computing	335
Concept: Classification of Computer-Networks	339
Maximal Number of Hosts and Network-Graph	340
Geographic Extension	340
Quality of Service QoS (Flow Capacity, Load and Reliability)	341
Communication in a Network — Protocols	341
How are the (Host) Computers identified in a Network? (Host Identification)	342
Inside the Network, how to find the Path from the Source of the Data to the Drain? (Pathfinding)	342
How is the Data actually sent, in what Form does the Data pass the Network? (Path Using)	342
Concept: Logical Connections, Virtual Networks	343
Concept: Data-Transfer with Packets and through Channels	345
Concept: Protocols and their Layer-Model	347
Protocol and Type	347
Layers of Protocols	347
More Classifying Criteria for Protocols	349
Client-Server, Peer-to-Peer Communication	352
Communication within the Same Levels of the Layer-Model	353
OSI-related Layer-Model	353
Networking-Protocols	357
The Internet Protocol (IP) — a packet-routing-, connectionless Protocol	357
IP-Addressing of Hosts	357
IP-Addressing of Hosts: Domain Name Service (DNS)	359
TCP/UDP Transport-Protocols	362
TCP - The Transmission Control Protocol — connection- and packet-oriented	363
UDP - The User Datagram Protocol — connectionless and packet-oriented	363
Addressing in TCP/IP-Networks: Sockets and URLs	363
Application-Protocols (HTTP, FTP, . . .), Types and Network Applications	367
Universally Recognizable Types and Formats	368
HTTP (Hypertext Transfer Protocol) the Protocol of the WWW	369
Hypertext	371
Copying Files over the Network	371
Private Mail	371
Other Applications	371
Triggering: URL- Network-wide Invoking of Application-Services	373
Uniform Resource Locator (URL) — An Introduction	373
URL-Encoding	375
Summary: Internet- and Transport-Protocols, Addresses, Domain-Names	377
ORGANIZE: Your Network-Connection	378

Java: Programming in Networks	381
Java: Relating IP-Address, Host-Name and Owner-Information	381
Java: UDP Sockets and Datagrams	384
Java: TCP-Sockets and Network-Connections	385
Java: URL, Application-Protocol- and Content-Handlers for CLIENTS	388
java.net.URL — The Default Mechanism	390
Java-Concept: Uniform Resource Locator in Client-side Connecting to a Network	390
Optional: java.net.URL — The Potential	391
Java: Applets as Programs in a Client-Side-Controllable Environment	396
Applet-Concept	397
Applet-Infrastructure	397
A Simple Applet	399
Organize: Browser with Java Runtime Environment: To Do	400
Applet Threading	401
Applet Introspection	402
Applet Communication	403
Java: Servlets / (CGI)	405
Servlet-Concept	406
Servlet-Infrastructure	407
A Simple Servlet	408
Organize: Install Server with Servlet-Container: To Do	409
Organize: Run a Servlet	410
Servlet Threading	411
Servlet Introspection	411
Servlet Communication	412
HTTP-Servlet Extension	413
Java: Remote Method Invocation (RMI) / (RPC)	414
Organize: JRE-Extension: What is needed for a RMI-Connection?	415
A RMI-Connection	416
How is the Remote Method Invocation Service activated?	418
What more?	419
Outlook: Java and Networking	419
Summary: Java Communication	421
Concept: Distributed Computing and Programming	423
 VI. Java Runtime Environment and Security	 425
Concept/Java: Scope, Access-Areas, Name-Spaces	429
Java: Runtime Environment	431
Class-Loading	433
Class-Loading: The Initial Process	434
Class-Loading: The Secondary Process	435
Class-Loading: Customized Class-Loaders	438
java.lang.ClassLoader — Simple Extension	438
java.net.URLClassLoader — Simple Instantiation	438
The Full Loading-Process of Instances derived from java.lang.ClassLoader	440
Maximally programmer implemented Class-Loader	441

Class-Loaders of Arrays?	443
Simple Class-Loading: new-Keyword and the Class.forName() -Method	443
Object Storing and Loading: Serialization	443
RAM Recycling: Garbage-Collection and Weak References	444
Security: Basic Concepts	447
Java Language- and Java Compiler-Security	450
Java Runtime Security	451
Security: Policy-Based	453
Java: Access-Control Mechanism	453
Secure Class-Loaders and Protection-Domains	454
From File to Policy-Object	455
Security-Manager and Access-Controller	455
Java User and Administrator	457
Text-File: java.security	457
Policy Text-Files	458
Selecting Policies for Running Applications and Applets	460
Some Practical Hints	461
Java Programmer	462
Java Security-Manager	462
The check...() -Methods of the Security-Managers	463
User and Administrator Use of a Security-Manager	465
Why a Security-Manager if there is an Access-Controller for consulting an External Policy?	465
Java Access-Controller: Policy	465
Java Access-Controller: Custom Permissions	467
Java Access-Controller: Protection-Domain	467
Java Access-Controller: Method-Call History	470
Java Access-Controller: Privileged Code	471
Java Access-Controller: Policy-Secured Environment extends to Objects	474
Applets and Security	475
Summary: Policy-Based Access-Control Mechanisms in Java	475
Security-Related Coding of Data	477
Basic Concepts of Security through Data-Processing	477
Social Aspects: TRUST	478
Concepts: Some Results of the Mathematical Theory	478
Cryptography with a Secret Key (Symmetric Key)	478
Cryptography with a Public/Private Key-Pair (Asymmetric Key)	479
Hashcode - Message-Digest - Digital Fingerprint - Checksum	481
Signing/Authenticating a Message Using Encrypted Message Digests	482
Securely Associating an Identity with the Owner of a Key	483
Variants of an Encryption Algorithm (Modes and Padding)	484
More Concepts	484
Java Provider-Infrastructure: Concept and Implementation	485
User/Administrator: Key-Management and Signing Files	488
Access-Controlling with Signed Data	492
User/Administrator: How to encrypt Files	492
Application-Programmer: Signing and Encrypting	492
Installing the Java Cryptography Extension (JCE)	496
Using the JCE	496
Infrastructure-Programming: Signing and Encrypting	497
Concept: Network-Security	500

VII. Single User Interface	503
Concept: User Interfaces	507
Hardware User Interface	507
Graphical User Interfaces (of Operating-Systems)	508
Graphical User Interface of Operating-Systems: Desktop - Icons - Windows	508
Graphical User Interface of Operating-Systems: Screen-Components	510
OS-GUI: File-System: Opening Files and Folders	510
GUI: Giving Commands: Clicking - Drag and Drop (D&D, dnd)	510
GUI: Giving Commands: MenuBars - PopupMenus - Toolbars	511
GUI: Windows as Containers of Screen-Components	512
Graphical User Interface of Application-Software	512
Design of Graphical User Interfaces (GUIs)	513
Concept: Programmer's View of User-Interface (UI)	515
Program-Structure: Model View Controller (MVC)	515
Screen Output	519
Concept: Computers and Images	520
The Concept of Image-Formats	521
Concept: Colors	522
The RGB Color-Model	523
The CMY Color-Model	524
The CMYK Color-Model solves the Printing-Problem: $C+M+Y \neq \text{Black}$	525
The HSV and HSB Color Models	525
Two Basic Image- and Color-Formats	525
Direct Color-Model: ARGB	526
Indirect Color-Models	526
Java: Transparency, Partial Transparency and Opaqueness	527
Java: <code>java.awt.Image</code>	527
Java: Simple Bitmapped Images	530
Concept: Graphical-Objects	534
Painting Graphical-Objects	536
Java: Object-Painting on Screen-Components	538
Concept: Screen-Components	540
Java: Basic Swing Screen-Components	543
Lightweight-/Heavyweight Components — Java: AWT versus Swing	548
Platform and Java Runtime Points-of-Contact	551
Java: Basics of Screen-Components	551
Java: <code>java.awt.Component</code>	552
Java: <code>java.awt.Container</code>	556
Java: <code>javax.swing.JComponent</code>	559
Java: Frame and JFrame	561
AWT and Swing Class-Hierarchies	564
Java: <code>ContentPane</code> and <code>JLayeredPane</code>	566
Java: Fonts	569

Event-Communication - Mouse and Keyboard	571
Event Delegation within the Abstract Window Toolkit (AWT)	571
Concept: Mouse Interaction	571
Concept: Keyboard Interaction	572
Java: The AWT-Event-Communication	572
Java: Delegation Event-Handling	574
Java: Triggering	574
Java: Switchboarding/Event-Scheduling	574
Java: Event Types	575
Java: Sources, Event-Objects, Sinks (Listeners)	576
Summary: Java Variants of Event-Listening	579
Java: Adapter-Classes	580
Java: Examples for Mouse- and Keyboard Event Handling	581
AWT-Switchboarding and Java	586
Using the Event-Dispatch-Thread	587
Java Programmer: Sources of AWTEvent-Objects	591
Java: processEvent () -Method as Drain and Source of AWTEvent-Objects	593
Interaction: Screen, Keyboard and Mouse	595
Java: Changing Components and Images On-Screen	595
Java: Painting-Methods	595
Pixel-Based Images	596
Graphics-Objects	597
Screen-Components	597
Summary: Drawing On-Screen	599
Java: User Changes User Interface	599
Java: Animation	602
Concept: Time-Delayed Processing	605
Java: Time-Delayed Data	606
Java Overview: Interfaces and Classes for Time-Delay	606
The Workings of an ImageObserver-Object	607
java.awt.image.ImageProducer	610
java.awt.image.ImageConsumer	610
Java Swing Desktop	610
Java: Applets - Applet Runtime-Environment	613
Triggering And Configuring an Applet-Run	614
Class-Hierarchy and Class-Structure of Applets	615
A Simple AppletViewer	619
Summary: Graphical User Interfaces (GUIs)	621
Outlook: Single User Interface	623
VIII Appendix	625
Reflection, Introspection and Profiling	629
Classes	629
Objects	630
System	631
Applets	631
Servlets	631

How to read Programs?	633
How to write and check Programs?	637
If Something goes Wrong	637
How to make Programs faster?	639
Further Questions	641
List: Prototypes of Programs	645
Primitive-Types and Control Structures	645
Code Structuring: Proceduralization-Methods	645
Code- and Data-Structuring: Object-Orientation: Class-Object	646
Code Structuring: Object-Orientation: Class-Extension	646
Code Structuring: Multithreading	646
Code Structuring: Aggregation: Arrays and Container-Classes	647
Code Structuring: Packaging and Libraries	647
Communication: Messaging	647
Communication: Streams and Pipes, Serialization (package <code>java.io</code>)	647
Communication: Component/Bean-Based Programming, Documentation	648
Communication: Computer-to-Computer: IP (Internet Protocol)-Networks (package <code>java.net</code>)	648
Java Runtime Environment, Class-Loading and Security: Class-Loading	649
Java Runtime Environment, Class-Loading and Security: Policy-Based Security	649
Java Runtime Environment, Class-Loading and Security: Security by Data-Processing	649
Single User Interface: Graphical User Interface (GUI) (package <code>java.awt</code> , <code>javax.swing</code>)	650
Appendix	651
List: Mentioned Packages, Classes and Interfaces	653
List: Syntax-Diagrams	655
Backus-Naur Form (BNF)	655
Syntax- or Railroad-Diagrams	655
List	656
Little Classification of some Notions	657
Sources	659
Sources On Java	659
Sources on the Internet	659
Sources on Programming	659
Literature	660

Preface

First are presented some questions and some attempts to answer them:

What is a computer? A machine that can be ordered to process data. The machine gets data and commands from keyboard, mouse, storage-disks, network-connections (like the Internet). The machine stores the processed data (a fax sent, an email received) on storage-disks and can show it on output devices like screen or printer. For more see the text beginning with page 28.

What is a program? A program is given by a sequence of commands which tell the computer what to do. Programmers store programs by writing those sequences of commands into text-files, as if writing a letter or an email. First examples of a program are given on page 37 and on page 71.

What is an operating-system? A program that organizes technical-related tasks of hardware-management (allocating storage-space, accessing hard-drives (also called hard-disks), disk-drives establishing fully-functional network-connections . . .). For more see page 43.

What is a platform? The computer hardware together with the working software of an operating-system is called a computer-platform (this is considered more extensively in the context of operating-systems).

What is Java? The term Java may be heard to be describing the entire **virtual platform**¹ given by the Java Runtime Environment (JRE, past page 51). Or the name Java may be used for the entire Java Development Kit, a **software-kit** that enables a programmer to write programs in the Java programming-language. But the term Java may also denote the Java **programming-language**. (For more information on programming-languages see pages around page 37.)

There are several overlapping, sometimes also rather fragmented, approaches to a multi-faceted piece of knowledge like the Java programming-language:

Handbook view: Notions are presented as far as possible in their strict logical dependence, assuming that the reader furnishes some familiarity with the concepts. The topics are usually covered extensively in an analytic, often rigid way. The ➡☐ ➡☐ Java Language Specification [JLS2] may be considered as an example of a publication with such a view.

Reference view: References collect notions, more or less commented, in a dictionary-like presentation, sometimes in the form of an index or a glossary. Examples are the ➡☐ ➡☐ Java Platform API Specification [JPAPIS] or the Java Developers Almanac [JDA].

Practical/Textbook view: Learning with examples that introduce and illustrate concepts. Synthetic experiences and experimenting becomes possible as part of homework problems. The Online Tutorials on *Sun's* Java-related Internet-site (<http://java.sun.com>) can be seen as examples for this kind of approach.

Programmed Learning (of practical skills): Learning in a textbook way by predominantly solving tasks under written guidance. Some certification books may represent examples for this approach. Search for “Java certification” in the Internet.

Theory-Oriented This means predominantly learning concepts of programming. In this context practical applications would distract from the central idea. Example: Algorithms (sorting, cryptography, routing packages of bytes through networks) can be formulated in many different programming-languages, their ideas remains the same. Many books with the notion “Algorithms” in their title may fit into this category.

This classification could make a reader aware of the choices and possibilities when searching for information. This present text could be classified as a textbook, although within the sections titled “Concept: . . .” this text attempts a theory-related introduction to the notions.

¹A computer-platform runs a program, called the Java Runtime System (JRS) that makes the computer mimic another platform: the so-called Java Runtime Environment (**JRE**).

This Book

This book tries to present a reasonable broad approach to computers, networks and Java programming (Java 2 Standard Edition, Version 1.4. The Java 2 Micro Edition (J2ME) and the Java 2 Enterprise Edition (J2EE) haven't been covered here.). As a result of some reading of this book, a beginner should be able to categorize notions of that field and be ready for learning advanced programming techniques. The text works as a collection of concepts and their transformation into the Java language. Marginal or advanced topics have been deliberately omitted. The Java programs presented are prototypes, maybe inviting to be changed and to be extended by the reader. All in all, the text should give a fast practical access to the Java language and its basic concepts. This book represents no atlas, rather a collection of charts.

Structure

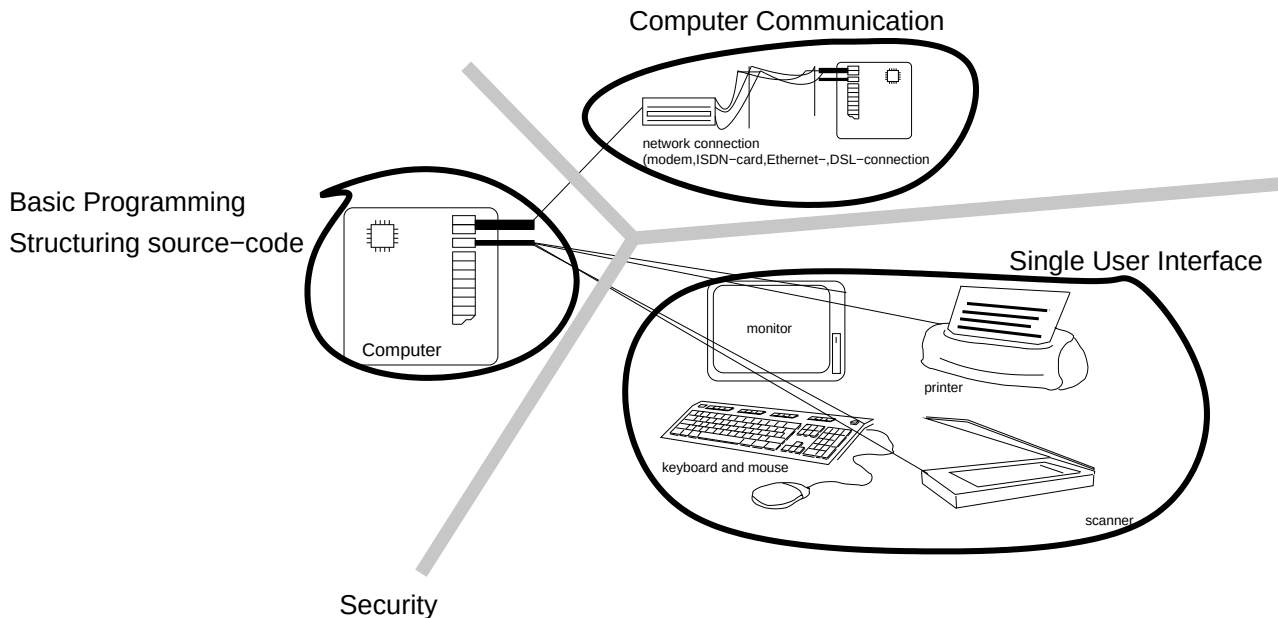
The text has been structured into *conceptual sections* and *Java-specific sections*. Conceptual sections describe general notions and give an overall idea of the content. The knowledge of these sections may be also applicable in non-Java contexts. The *Java-specific sections* demonstrate the realization of those concepts in the case of the Java programming-environment. — Headers of the parts and sections of the book denote the main view-point of these passages. Nevertheless *some* extensions of those main ideas can be found in *other* parts of the book. But these extensions are bound to their respective main passage by cross-references, so that they should be easily relatable. — Many programs have their descriptions added to the program's (source-)code as comments. This may better integrate the programs into the text and may make it straightforward to identify program-specific information. — Concepts and principles are introduced predominantly without referring to any special programming-language. Those paragraphs should furnish the theory that makes much of the  Java Platform API Specification self-explaining. (See the concept of color on page 522 which gives some of the basic information used in the Java-class `java.awt.Color`) — The text presents simple procedural programs first; the structuring of code with methods, classes and threads comes later. — To keep the ideas above in a good order, some notions of the Java language are anticipated at times (especially the construct of a `main()`-method inside a class-declaration, being indispensable when writing regular programs.). — All the examples can be refined and extended and to get practice in Java; the reader may benefit from doing some creative work with them. — The text allows the reader several lines of access to spot specific information:

- Table of Content
- Index
- Chapter and Section Introductions/Summaries should make a small text-booklet of an overview
- Images as a picture book of ideas
- Most of the gray frames are part of a glossary. Related details can be found in the context of these gray frames.
- List of classes or interfaces introduced (in the Appendix)
- List of Programs (in the Appendix)
- List of Syntax-diagrams (in the Appendix)

Another approach may be taken by browsing through the text, then checking and maybe even extending the programs, thereby referencing the text if necessary. A text on the Java Application Programming Interface (API) available for parallel reading may give details which have been deliberately omitted in this text. Some of those details also may have been changed at the time this text has been published.

Content

The central parts of that book are basic programming, structuring source-code, communication between computers, security and single user interfaces:



The first chapters describe the building elements of the Java language-expressions and statements. These building elements do not differ very much from the basic structures in languages like C/C++, Pascal, Component Pascal (extension of Oberon-2) or COBOL. — Then chapters titled Proceduralization, Object-Orientation, Multithreading, Packaging and to a certain extent the chapter about communication (messaging, streaming, component-programming) consider the structuring of code.

A computer-model is introduced to envision the hardware- and software-context in which Java is running. Basic programming introduces to basic programming notions (variables and their creation, expressions like assignment) and their realization in Java. Control structures essentially constitute the acting source-code (if-then- and loop-statements), the rest of the items describe methods to structure and organize the code:

Proceduralization means sorting out a relatively independent sub-task and giving that sub-task a name of its own. This introduces the notion of a Java-method (in Pascal or C/C++ a function or a procedure). The idea is to **give reusable chunks of code a name**, and to call that piece of code again as needed (page 115).

Object-Orientation (page 131) develops the idea of code-reuse in different directions: One aspect means furnishing **data together with the methods to manipulate the data**; this also means extending the notion of a type of a variable. Another aspect means **designing code for extensibility as another way to organize code for reuse**. A third aspect means organizing or **indexing code in a hierarchy** for easier reference. If larger programming tasks have to be solved in a collaborative approach by several persons, then object-oriented structuring of code may prove to be an advantageous approach.

Multithreading (page 197) means running **independent sub-tasks parallel** on multiple processors; and letting them communicate with each other: This means, for example, that some threads wait for information another thread generates.

Packaging (page 239) describes collecting code in a software-library, this just means collecting several meaningfully-related entities of code (Java-classes) and giving them a common surname for easier reference.

Leaving the elements of the core language-functionalities opens a field of selected issues: The part about Communication (page 271) includes a description of how to realize a delegation event-model consisting of source, message and drain. Throwing objects, especially so-called exceptions or errors, happens within a Java-internal communication-infrastructure. The concept of a stream may describe a one-way buffer of incoming- or outgoing data. Wrapping streams includes the idea of “on-the-fly” processing of stream-data (after reading in-, before writing out the data).

Basics of the CPU-to-CPU communication (computer-networks, page 337) stress TCP/IP-communication. Java-specifics like Remote Method Invocation (RMI) are introduced.

Part of the text also considers how to turn a file on the hard-drive, maybe downloaded from a network, into a program ready to be run. This part of activation of program-code is also called “class-loading” (page 431). The area of computer-related security (page 447) is considered by trying to answer questions like the following: How to control a user’s access or a fellow program’s access to hardware and software on a computer-platform, especially in the Java Runtime Environment (JRE), the Java virtual platform. Finally, single user interfacing (page 507) introduces concepts and examples of how monitor, keyboard,

mouse interact with Java and the end-user. There, the Model-View-Controller (MVC)-concept describes another way of structuring Java source-code. The Appendix may be seen as collecting the remainder.

Limitations

This book tries to get the concepts clear and give a basic introduction that helps the reader to get own programming experiences. The book does not provide exhaustive information on the Java language like any handbook or reference book. For in-depth systematics see the   Java Language Specification [JLS2] or other Java Language References. Thus, ramifications of the Java language and Application Programming Interfaces (APIs) are left to be found in the   Java Platform API Specification [JPAPIS] and inside handbooks on this subject as mentioned in the Appendix. This especially concerns information which Java version (1.0.1, 1.1, 1.2, 1.3, 1.4) offers which options.

This text presents no in-depth description of standard software-libraries of Java (Application Programming Interfaces (APIs) of Java). For exhaustive information see the books from Addison-Wesley or O'Reilly. For an index-like compilation of the software-libraries see the online-documentations from *Sun Microsystems Inc.* or the Java Developer's Almanac [JDA].

The book is based on the **second edition of the**   **Java Language Specification** [JLS2] and covers the **Java 2 Standard Edition (J2SE)** (not the Java 2 Micro Edition or the Java 2 Enterprise Edition). Only the Java 2 Standard Edition (J2SE) has been considered in this book, aspects of the Java 2 Micro Edition (J2ME) and the Java 2 Enterprise Edition (J2EE) have been omitted. The implementation and some aspects of the Java design may be subject to criticism; this book has been written in the belief that the drawbacks are well outweighed by the advantages when comparing Java to other existent programming-environments.

Algorithms (roughly identifiable as the language-independent version of a program), such as sorting algorithms or cryptographic algorithms, are not described in this text. See the Appendix for hints on sources of algorithms.

The writing of source-code as presented here, does not show how to organize the writing of a **large software-project**. This field is usually treated in books about Object Oriented Design of programming tasks. No extended discussion of component-software is to be found; neither are mentioned other means of structuring code independently of a programming-language, for example the UML (Unified Modeling Language).

Documents, databases, their formats, production and handling, and **image-manipulation** are not covered in this text. This means NO DOCUMENT PROCESSING, NO DATABASES, NO IMAGE-PROCESSING. This text provides no information on how to program text-editors and related tasks (character encodings surpassing the ASCII²-format, the Unicode-format or a Unicode Transformation Format (UTF) for Unicode-characters, font-management, document-types like the Portable Document Format (PDF), Extended Markup Language (XML), drag and drop implementation). — Relational- or object-databases are not mentioned, neither has been introduced the Java DataBase Connection (JDBC), a Java software-library for accessing data in relational databases with the Structured Query Language (SQL).

References to hard- and software are limited to those products that are likely to be used in a more or less average home-environment: Mouse, keyboard, monitor. The text assumes that the reader has comfortable access to a computer, an installed operating-system, at least a dial-up Internet-connection and maybe two computers connected with each other over, for example, an Ethernet connection. The additional Java software should be downloadable for free from the Internet. Differences between platforms (Unix-type like Linux, WindowsNT/95/98, Mac OS) are not treated exhaustively.

Many of the introduced programs have been tested on 233MHz G3-Mac and Intel-type machines. Generally the programs are written reduced to the essentialities, so that they should present the gist of the functionality they intend to demonstrate.

The Reader

A reader might have the following interests or goals: Know what the Java language means and to be introduced to some of its central capabilities. Wants to get an overview and a short introduction. Wants to check the prototype programs and use them as a basis for own programming efforts. Know how to install a working, cost-free Java Runtime System (JRS) and development-environment on Apple, Linux and Windows platforms. Use another book for learning programming algorithms and other advanced programming-techniques (for example text- and image-processing or database-programming).

²American Standard Code for Information Interchange

The Authors

Almost every view on a subject cannot be complete or ignore the descriptor's point of view. The non-completeness aspect clearly can be seen by the limited size of the book. The authors point-of-view clearly sets its limits and preferences. The goal was to establish a foundation from which the wealth of the Java programming system can be explored without getting lost in too many details.

Conventions

The names of subjects are chosen to avoid unnecessarily specialized jargon. Nevertheless in some contexts the usage of notions may vary from that chosen in this book. The index and the explanations in the gray rectangles try to take account of that. The symbol ➡☐ means, there exists an online-source for further information and ➡☐ ➡☐ indicates that this online-source can be considered as highly recommendable for further reading.

Source-code of the (Java) programming language and the content of text-files are presented in typewriter-font.

Parts of the text, that represent commands or programs (in typewriter-font), may contain passages written in *italics* or passages parenthesized by “smaller than” (<) and “larger than” (>) symbols. This indicates that these passages are placeholders for values and have to take a concrete value to give a valid expression. Often these placeholder-passages indicate what values they expect. For an example see the graphic past the program-code on page 71. And see the code fragments at the beginning of the section on the switch-statement on page 107. In the part of the text about networking and HTML-formats, the “smaller than” (<) and “larger than” (>) symbols are also used as so-called HTML-tags (page 370). In Bachus-Naur expressions, which should be clearly discernable from HTML-text or terminal-commands or programs, the “smaller than” (<) and “larger than” (>) symbols indicate that the enclosed entity occurs one or multiple times (page 655).

Arrows (→), within the text of the gray boxes, denote notions which have been referenced by the index with a capital letter. In other context, arrows within the text can be substituted by the words “is super-class of” (see page 615, following the graphic).

Part I.

Computer-Models

Computer-Models — Wanting to know all about a computer may resemble wanting to know all about a car. Most people want to use the car for driving, not as an object of study to become a car-mechanic. So, the average user of a car has limited skills for repairing or servicing the car. To the average user the car represents just an object that helps to accomplish personal purposes. Therefore, every motorist has a well-developed set of ideas about a car's behavior, maybe not as refined as that of a test-driver, but enough to serve the daily needs well.

The relation between a user of a computer-system and the computer exhibits aspects of the same kind. The user of a computer-system confines her- or himself to those few skills which are necessary to get the computer-system's support in accomplishing the daily real-world tasks. The notion of a computer-model intends to give this collection of skills and ideas a consistent environment.

The following part of the text considers simpler computer-models, computer-languages, operating-systems and file-systems. Extensive information about end-user interfaces of various operating-systems can be found in the last part of this text beginning with page 507.

Elementary Computer-Models

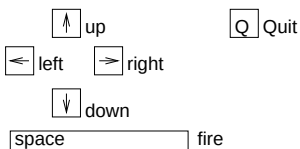
Elementary Computer-Models — First, this table shows different ways to use a computer:

recreationalist	uses a computer-game	with joystick, CD-ROM, monitor
writer	uses a text-formatter	with keyboard, monitor, black and white printer
layouter in press	uses graphics-software	with scanner, printers
application-programmer	uses interpreter, compiler, debugger, component-construction software to program: graphics-software, text-formatters or computer games for an operating-system environment	with keyboard and monitor, modems
utility-programmer infrastructure-programmer system-programmer	programs drivers, assembler, interpreter, compiler, debugger, low-level network-software, operating-systems for a specific hard- or software	with keyboard and monitor, modems
person, task	uses software	with additional hardware

Different users have different ideas how they access their computers and receive services from their computers, they have different ideas about their computer's physical components and software functionalities. End-users, who want to be helped by the computer in solving some of their real-life-tasks cannot be required to have the knowledge of a programmer. To be able to discern pure technicalities from skills for effectively solving real-life-problems with the computer, often requires a considerable knowledge in advance.

Depending on the software or hardware the user faces, each user assigns the computer-system a different set of functionalities. Such a set of functionalities can be described coherently by constructing an appropriate computing-model. In that sense the model can be used to “explain” the specific computing system (to the user).

A part of a game player's computer model



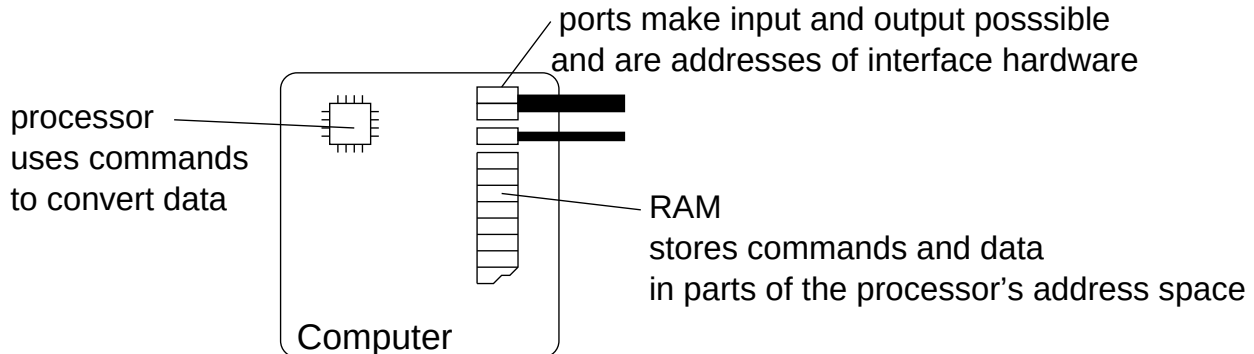
Some computing models are simple, like a computing model for a video game. Although the computing-model for the video game tells what can be done, it doesn't reflect complexity and richness of the video display and it does not tell how to acquire the skill to master the game.

The more functionalities the software furnishes, the longer the list of commands and the more difficult it may be to remember them. Even well-designed models have to reconcile contradicting perspectives: Sim-

plicity of design makes learning and standard work fast, complexity of the models slows the learning-process but adds flexibility, opening ways to more efficient work.

A Computer-Model for Programmers

A Computer-Model for Programmers — For programmers an introductory model of a computer can be made fairly simple:



The processor, its address-space, within that space the Random Access Memory (RAM) and the ports, are given here as central notions of a computer-model. Input-data appears in the Random Access Memory (RAM) of the computer via ports, that occupy address-space which the RAM does not use. Output-data leaves the computer's Random Access Memory (RAM) by transferring the data to addresses occupied by output-ports. The Random Access Memory (RAM) stores the data. Part of the data gets recognized by the processor as sequences of processor-commands. Thus the processor can convert, manipulate or process data autonomously. A processor³ manipulates, changes data which has been stored in the Random Access Memory (RAM).

Processor: The piece of hardware inside a computer that manipulates the data it fetches from the Random Access Memory or hardware-ports. After the processor has fetched and manipulated the data, the new data is put back into the Random Access Memory or written into a port that represents a connection to a hard-disk or a computer-network. Multiprocessor-computers have multiple processors built into them; their function has to be coordinated by appropriately designed hard- and software.

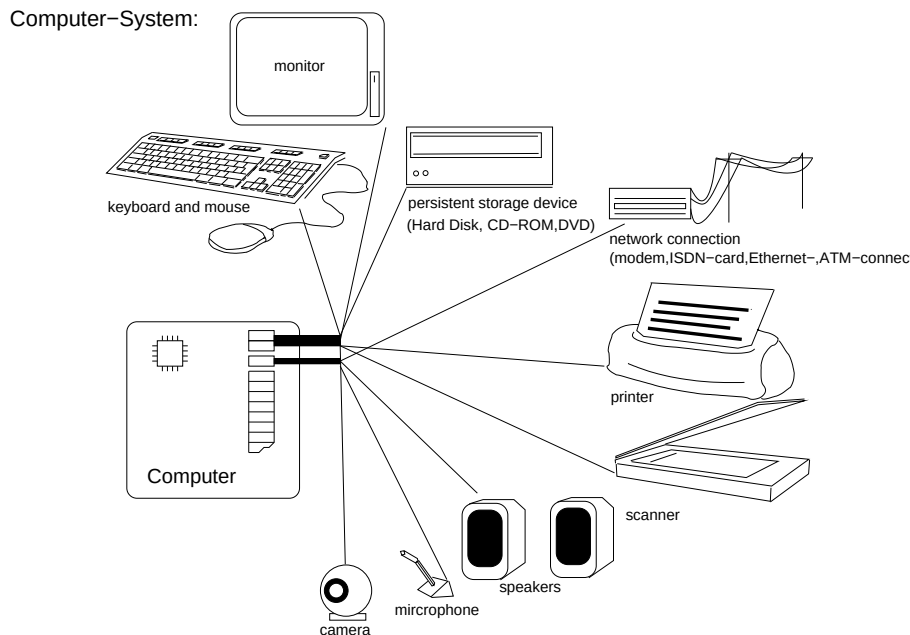
CPU: Acronym for Central Processing Unit. 1. The processor(s) of a computer; the part of a computer-hardware that does the main data-manipulation. 2. A Central Processing Unit consists of the microprocessor(s), the Random Access Memory (RAM) and busses that connect both with Input/Output interface-hardware. 3. An entire personal computer, excluding keyboard and monitor.

Again: The acronym CPU stands sometimes for the processor alone, sometimes for the processor and the RAM and more seldom even for an entire personal computer (which includes an internal hard-drive but without monitor and keyboard).

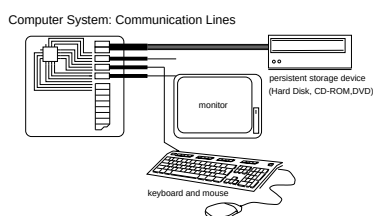
The command-sequences, that make the processor work on the data, are stored together with the data-to-be-manipulated in the Random Access Memory (RAM).

RAM: Acronym for Random Access Memory. Often simply denoted "memory". Stores data, which includes coded information and sequences of processor-commands. Generally, the processor can access each unit of information of the Random Access Memory directly. Access to data usually happens fast (in comparison with devices like hard-disks and network-connections). But data residing in Random Access Memory usually is stored non-persistently, that means the data is lost if not saved to a hard-disk before turning off the computer's power.

³Most processors in personal computers are, technically speaking, special integrated circuits – so-called chips – that have been given a set of very general functions. An integrated circuit is made of a collection of connected transistors and other electronic elements built into a chip primarily made of crystalline silicon. ("silicone" denotes a rubbery compound of silicon)



A **computer-system** is made of the computer itself and additional devices like low data-rate input/output computer-system (I/O) devices, for example keyboard and mouse, and higher data-rate I/O devices like monitor, storage devices (hard-drive, CD-ROM), printer, scanner, microphone, loudspeakers, camera, modem or network-card (usually of the Ethernet-technology). The maximal data-rate of the communication-lines between components of a computer and the maximal data-rate of the components itself determine the performance of parts or the overall computer-system. (Analogously to a network of several computer-systems which draws its overall performance from the data-rate of the data-lines and the rate with which the individual computers can process the data.) That's much like a system of pumps and pipes for transporting water: The pipes' capacities have to measure up to the pump's power and vice versa.



A communication-line, given by a bundle of wires (more than two usually around fifty or hundreds), is called "**bus**". Those lines are usually more efficient than the single lines. Busses generally are used between hardware-components of a computer-system that have to maintain a high data-rate. Communication lines of different technologies are connected either by a "**bridge**" or by an "**interface**" (card); that's an electrical circuit for establishing the connection between otherwise unreconcilable types of communication-lines. If the user is not concerned with modifying the hardware of the computer, then there won't occur

many occasions for handling hardware-interfaces.

Above have been described some conceptual similarities of different computer-systems. But different types of computer-systems differ significantly in technical details. For example, different makes of processors take different sets of commands.

Especially the technical development, which furnishes faster and hopefully easier-to-use hardware, makes older technical architectures obsolete. Consider some averaged performance-data of computer-systems:

year	processor-speed (roughly in commands per second):	Random Access Memory size:
1990	20MHz	4MB
1998	200MHz	40MB
2002	2000MHz	400MB

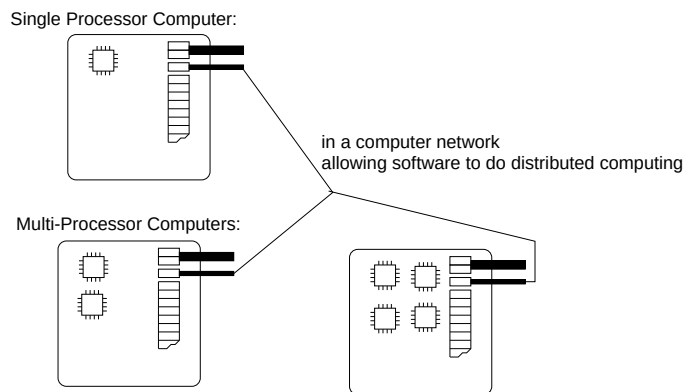
Next to the computer's internal bus that connects the processor and the Random Access Memory, there are communication-"lines" *within* the computer: The PCI (Programmable Communication Interface) with up to 64 parallel data-lines yielding a data-flow of up to 132MB per second. The EIDE (Enhanced Integrated Device Equipment) bus with up to 33MB per second. Also, predominantly for adding external hardware to a given computer, are mentioned the SCSI (Small Computer Systems Interface) bus with up to 40MB per second for connecting up to seven external devices to the Central Processing Unit. The IEEE1394-line (also called FireWire or iLink) with up to 50MB per second and the USB (Universal Serial Bus) with up

to 15MB per second as well as the Ethernet hardware with up to 100MB per second. The technological data is changing and may have changed when this has been published.

The acronym MHz (Mega-Hertz) indicates the processing-speed and the acronym MB (MegaByte) indicates the volume of the data. For what purposes may be possibly needed such information? When buying a computer, various computer-systems can be roughly compared. Then, using a computer, performance limits can be assessed, thus making it possible to quantify the volume of the Random Access Memory (RAM) or to estimate the dimension of a needed storage-device. For example, error-messages like “out of memory” may get a clearer meaning and may actually result in adding RAM to the computer. For more on these units see the subsections past the next for “MHz”, and for more on the unit of the data-“volume” (MB) see the page 34.

Computers with Multiple Processors and Networks

Computers with Multiple Processors and Networks — Multiprocessor-computers consist of several processors, connected with each other by the fast computer-internal bus-system. These multiple processors (usually 2, 4, 8, or 16) use more or less the same Random Access Memory (RAM) and usually can access the same hardware-ports. Non-expensive consumer computers usually are equipped with only one processor. But those one-processor-computers can emulate a multiprocessor-environment to allow multi-threaded programming too (see page 197 for multi-threaded programming in Java).



Networks connect entire computers mostly by external wiring, for example with versions of Ethernet-hardware, or even by wireless radio-connections. These external connections among computers usually do transfer data slower than the internal bus-systems of the individual computer. The difference between a network and a multiprocessor-computer also can be formulated as follows: Each computer in a network has its own Random Access Memory (RAM). In a multiprocessor computer the processor have to share the Random Access Memory. For more about networks see the part of this text, beginning on page 337.

Summary: Computer-Models

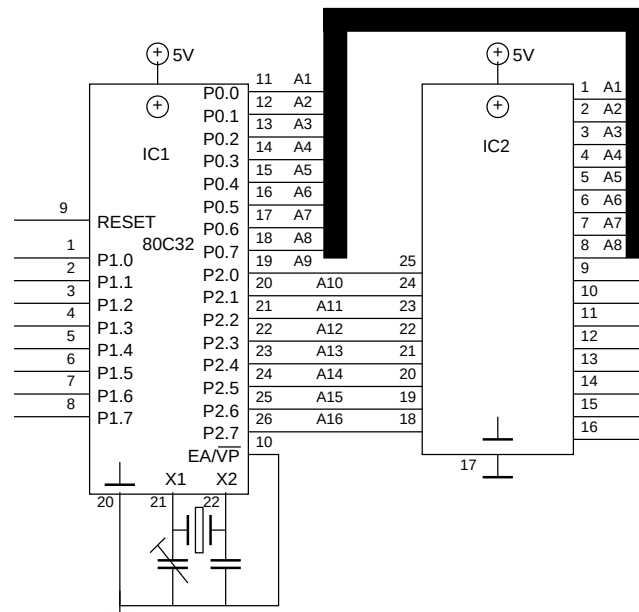
Summary: Computer-Models — There are imaginable different computer-models; one of those models, that gives many details independently of the technology, has been presented more extensively: The processor takes commands and data and returns transformed data. The Random Access Memory stores data, part of that data are sequences of processor-commands. Communication-lines like busses transfer the data between components of the computer-system. Why, next to Random Access Memory (RAM), are needed other storage-devices such as hard-drives (magnetic disks) or CD-ROMs? RAM works fast, is expensive and loses the data (and programs) in the moment the computer-system’s power is turned off. Compared to RAM, hard-drives are inexpensive and hold the data permanently (persistently) until deleted by the user. For storage-purposes their slowness doesn’t matter. CD-ROMs are what the name indicates, Read Only Memories on compact-disks of about 600MB. They are easy to transport and their data cannot be changed, which can be seen as a security advantage. The same applies to DVD-RAMs and DVD-ROMs, but they can store about 5000MB, about ten times the data-volume of a CD-ROM.

How to start a computer? The model described above lacks any related description. In the technical context, this process requires additional features to be included into the model. An end-user may answer that question saying: “By pressing the power button.” This turns out to be a workable and rather sensible answer on the level of the end-user’s model and working-requirements.

A Computer-Model for Hardware Technicians

A Computer-Model for Hardware Technicians — Even hardware technicians use computer-models. Their models are much more refined towards the technical aspect of computing.

The graphic below is meant only for getting a feeling for the technicalities involved. The large rectangles represent individual silicon-chips. Each line represents a single electrical connection. The symbol that looks like a little sandwiched rectangle, in the lower left part of the graphic, near the number 22, represents the oscillating crystal, which gives the frequency (also called clock-rate) according to which the computer works. The number of times an event reoccurs per time-interval is called frequency. The crystal frequency inside the computers wobbles millions of times per second. This frequency of oscillation (millions of times per second) of such a crystal is called one Mega-Hertz. An average car-engine turns about hundreds of Mega-Hertz times per second (thousands of rotations per minute), which is slower about a factor ten-thousand than the frequency of a computer's oscillating crystal.

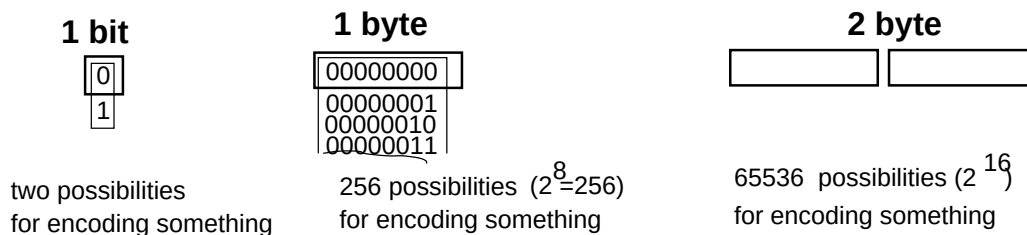


Processor-Commands, Languages

Processor-Commands, Languages — How and by what means gets the processor told what to do? Direct communication with a processor happens by signaling sequences of voltage-/no-voltage-pulses synchronized by the computer's clock rate. These sequences code data and commands that tell the processor what to do with the data. For evading this level of technical complexities have been designed programming-languages.

Bits and Bytes used For Coding Human Readable Symbols and Numbers

Bits and Bytes used For Coding Human Readable Symbols and Numbers — Coded real-life-data and data in form of processor-commands enter the processor as a clock-controlled stream of voltage-pulses of several volts (usually about 5 Volts but also 3 Volts in some computer-architectures). A clock, that can be seen as part of the CPU and generally has been realized by using an oscillating quartz-crystal, gives a beat that synchronizes the voltage pulses (See the symbol in the previous graphic marked with the number 22). The information that is transferred by either sending a voltage pulse (for describing-purposes coded as a 1) or not sending one (coded as 0 (zero)) is called one **bit**⁴. One **byte** is the name for a sequence of eight bit and byte bits.



A byte or multiple bytes are chosen to be the units of accessibility of data: The RAM stores files as sequences of bytes; the processor-commands are coded in bytes. Furthermore bytes can be used for coding human readable symbols and numbers (real-world-data); depending on the context (the coding-scheme). Thus a byte can be given different meanings:

A byte can be seen as coding a number; as can be seen above, a byte can code 256 different numbers. Or a byte can be used to code characters of the alphabet, like in the ASCII-coding-scheme. A pair of bytes can be used to code 65536 different symbols, characters or numbers, this can encompass most alphabets of the world and such an encoding-scheme is given by the Unicode-encoding-scheme which extends the ASCII-scheme.

⁴ A flashlight and a regular clock, together with a given time-interval, for example ten seconds, can be used to transfer one bit of information every ten seconds by coding 1="flashlight on" and 0="flashlight off" during each ten-second-interval.

Number-representation in depth: Number-coding-schemes often follow the mathematical background of number-representation. A natural number 123 (hundred and twenty-three) can be rewritten in the decimal number-system the following way (In 123_{10} the subscript 10 stands for the decimal system, in which numbers usually are written. No subscript means that the number is written in the decimal system by default.):

$$123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 \text{ or}$$

$$123_{10} = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1111011_2$$

in the binary number-system. So, the 256 possible sequences of bits in a byte can be interpreted as the binary number codes for the decimal numbers 0 to 255. For humans, the binary number-system usually gets tedious to work with; the decimal system has been proven more useful for everyday life. The so-called hexadecimal number-system is widely in use for programming with processor-commands (as can be seen with so-called “hex-dumps” of programs written in processor-commands, one of these “hex-dumps” has been printed out on page 36.):

$$123_{10} = 7 \cdot 16^1 + B \cdot 16^0 = 7B_{16} \text{ where } A=10, B=11, C=12, D=13, E=14, F=15.$$

Addition, subtraction, multiplication and division can be described in any of the number-systems above. In computers, those elementary mathematical operations are hard-wired into the electrical circuits for the use with binary numbers. As an example see the aforementioned hex-dump on page 36, an Ethernet network-address in its usual hexadecimal format on page 342 and the coding of the color of an image point within a colored, possibly partially transparent image on page 556. In Java, the class `java.lang.Integer` furnishes a method `parseInt( String s, int numberbase )` for analyzing a given string (s) as a number of a given number-base and returning that number in a decimal notation as an `int`-value^a.

^aMore can be found in the TechTip of December 4th of 2001.

As has been indicated above, a byte may code the numbers 0 to 255. But a byte may code another range of numbers, *for example* those from -128 over 0 to 127, this coding-scheme has the name “byte” in the Java language⁵.

The widely used ASCII-encoding-scheme assigns different bytes to different letters and symbols. This becomes useful when identifying keyboard-inputs or to print characters on the screen or on paper. The ASCII-coding-scheme includes the ranges of numerals $'0' \equiv 48_{10} \dots '9' \equiv 57_{10}$, the range of upper-case letters $'A' \equiv 65_{10} \dots 'Z' \equiv 90_{10}$ and the range of the lower-case letters $'a' \equiv 97_{10} \dots 'z' \equiv 122_{10}$.

The Unicode-encoding-scheme extends the ASCII-coding-scheme by using a second byte; with two bytes Unicode can encode many of the world’s alphabetical symbols. For the full range of characters encoded see   <http://www.unicode.org/charts>, there can be found the associated character charts.

Characters	'0'	...	'9'	'A'	...	'Z'	'a'	...	'z'	'α'	...	'ω'
Hexadecimal Code	0030	...	0039	0041	...	005A	0061	...	007A	0381	...	03C9
Decimal Code	48	...	57	65	...	90	97	...	122	897	...	969
Binary Code	110000	...	111001	1000001	...	1011010	1100001	...	1111010	1110000001	...	1111001001

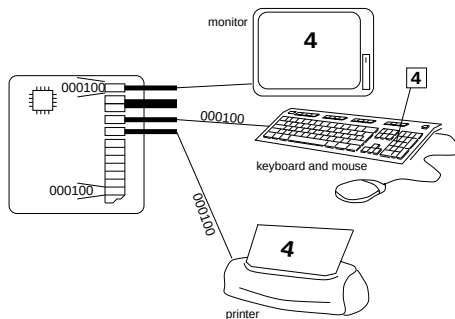
But the character-coding-schemes do not furnish the shapes needed to actually draw the character onto the screen or onto paper. Usually, the shapes or images that represent the characters are given by fonts or glyph-software and have to be incorporated separately into the computer-platform! (Concepts of encoding images are first presented on page 521)

Unicode: An association of a two-byte unsigned number (0, 1, 2, ... 65535) to a set of characters (a,b, ... , α , β , ...). The details of this convention can be found under <http://www.unicode.org>.

Any information that goes into the computer, like programs, images and texts, any such entity — which usually is given as a file — is coded as a sequence of bytes. Of course, a large number of bytes is needed to code an entire program, book or email. Ranges go from a few hundred bytes to thousands of bytes

⁵Coding-schemes in languages often additionally furnish means of transforming and changing the informational content of the data; coding-schemes thus extended are called types. — Complications that arise from applying different coding-schemes to the same byte(-sequence) are demonstrated on page 88 with a program that interprets the same bytes in two different ways.

($\approx kB = 1024$ bytes “kilobytes”), millions of bytes ($\approx MB = 1024 * 1024 = 1,048,576$ bytes “MegaBytes”) or even billions of bytes ($\approx GB = 1024 * 1MB = 1,073,741,824$ bytes “GigaBytes”). Second, in some technical descriptions another unit for counting the amount of data is used: $1Kb = 1\text{kilobit}$ are 1024 **bits**. $1Mb = 1\text{Megabit}$ are $1024 * 1024 = 1,048,576$ **bits**. The values, in comparison to their above-mentioned counterparts in **bytes**, are thus shifted by a factor eight. (Third, the values are not exactly those expected from the usage in the natural sciences, where M stands for a million ($=1,000,000$) and kilo means thousand. For example, $1kHz$ means $1000cps$, a thousand cycles per second. On page 341, $64kbp$ means 64000 bits per second.)



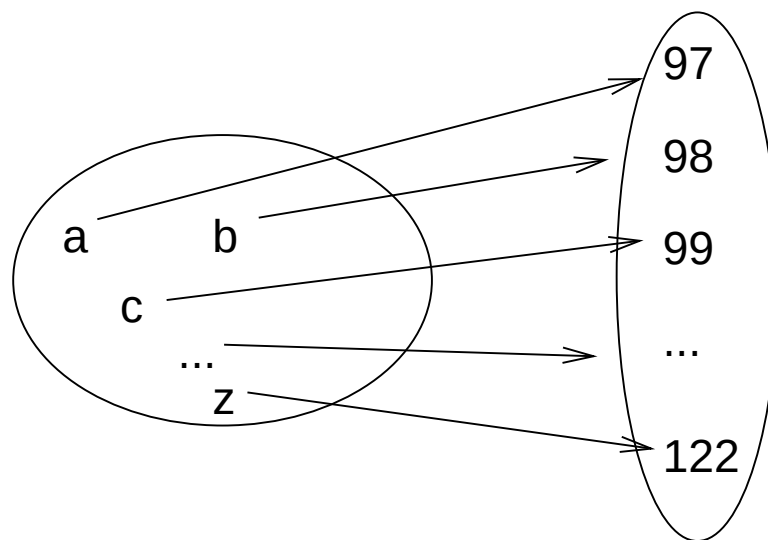
Consider a set of items, for example characters, which are used to represent information as sequences of these items. A **coding-scheme** or **format** takes the set of items, usually characters, and assigns each of them a different number. This numbering-scheme can be used to store the information as sequences of numbers in the computer; and the scheme is also used to retrieve that information from the computer. (The notion of a type extends the coding-transformation by adding additional content-transformations that change the information of the data, for example by incrementing the stored number by one or by adding another number to it (semantic manipulation). For more on coded information see page 73.)

Format, Encoding: A coding-scheme, a format or an encoding describes a mapping from a set of entities to (binary) numbers.

coding scheme (part of ASCII):

set of symbols or characters

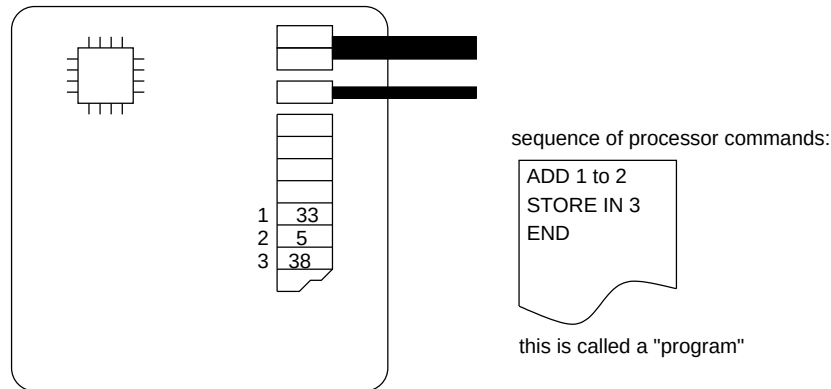
set of numbers



For more on coding-schemes, fonts and their glyphs see page 265. An Internet-source of general formats is also given by <http://www.wotsit.org>.

Commands on Processor-Level: Machine-Language and Assembler

Commands on Processor-Level: Machine-Language and Assembler — Programs for computers are sequences of commands. The processor works through these sequences of commands, command after command. The typical case of adding two numbers is illustrated in the graphic below:

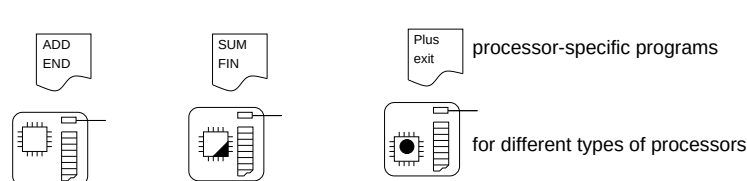


The program-commands in the graphic above are also stored within Random Access Memory as a sequence of bytes.

As another example consider the piece of Java byte-code that makes the Java Virtual Machine (JVM), a (virtual)-processor⁶, manipulate some data:

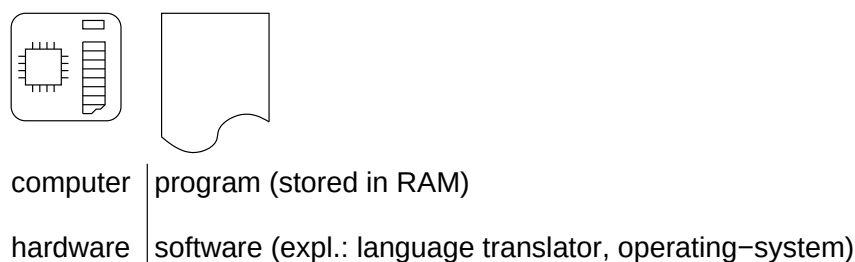
```
cafe babe 0003 002d 0020 0000 1507 0012 0700 1a07 001b 0700 1c0a 0004 0009 0900 0500 0a0a 0003 000b 0c00 0f00 0c0c 001e 0017 0c00 1f00 0d01 0003 2829 5601 0015 284c
6a61 7661 2f6c 616e 672f 5374 7269 6e67 3b29 5601 0016 285b 4c6a 6176 612f 6c61 6e67 2f53 7472 696e 673b 2956 0100 063c 696e 6974 3e01 0004 436f 6465 0100 0d43 6f6e
7374 616e 7456 616c 7565 0100 0e45 7861 6d70 6c65 5072 6f67 7261 6d01 0013 4578 616d 706c 6550 726f 6772 616d 2e6a 6176 6101 000a 4578 6365 7074 696f 6e73 0100 0c48
656c 6c6f 2057 6f72 6c64 2101 000f 4c69 6e65 4e75 6d62 6572 5461 626c 6501 0015 4c6a 6176 612f 696f 2f50 7269 6e74 5374 7265 616d 3b01 000e 4c6f 6361 6c56 6172 6961
626c 6573 0100 0a53 6f75 7263 6546 696c 6501 0013 6a61 7661 2f69 6f2f 5072 696e 7453 7472 6561 6d01 0019 6a61 7661 2f6c 610e 672f 4f62 6a65 6374 0100 106a 6176 612f
6c61 6e67 2f53 7973 7465 6d01 0004 6d61 696e 0100 036f 7574 0100 0770 7269 6e74 6c6e 0021 0002 0004 0000 0000 0002 0009 001d 000e 0001 0010 0000 0025 0002 0001 0000
0009 b200 0712 01b6 0008 b100 0000 0100 1600 0000 0a00 0200 0000 0300 0000 0200 0100 0f00 0c00 0100 1000 0000 1d00 0100 0100 0000 052a b700 06b1 0000 0001 0016 0000
0006 0001 0000 0001 0001 0019 0000 0002 0013
```

The data is given in the Unicode-coding-scheme and presented as hexadecimal numbers. Each pair of characters or digits represents one byte. Thus each block of four symbols represents one Unicode-character. The first four bytes of the sequence of bytes above identify it as Java byte-code, in hexadecimal form they read “ca fe ba be”.



Different kinds of processors from different makers use different commands and those processor-specific commands make processor-specific programs. Therefore, each time a new type of processor (680x0, Pentium, PowerPC) has been introduced by a firm (*Motorola, Intel, IBM*), these

programs have to be adapted to the new processor. Furthermore, writing programs as sequences of processor-commands is tedious, because this means working with binary code, shortened to hex-code by translator programs, which themselves have to be written in binary code in the first place. Even so-called assembler-programs, allowing the programmer to enter processor-commands as character-sequences like ADD, END or SUM, FIN, are processor- and hardware-specific and therefore have to be rewritten each time a hardware-component changes.



When considering different types of processors, then different sets of processor-commands make them work, consequently different programs have to be written to handle the same task. Thus, the programs have to be compatible with the hardware on which they run!

Software, Hardware: Programs are called **software**, the physical parts of the computer are called **hardware**. To be run, the programs have to be stored in the computer's Random Access Memory.

⁶Here “virtual” means software-emulated. Later, more will be said about virtual processors (implemented Java Virtual Machine) and virtual platforms (Java Runtime Environment, JRE). At this point the Java Virtual Machine (JVM) can be imagined as a real piece of hardware, a silicon-chip with processor functions.

Languages for easier Coding

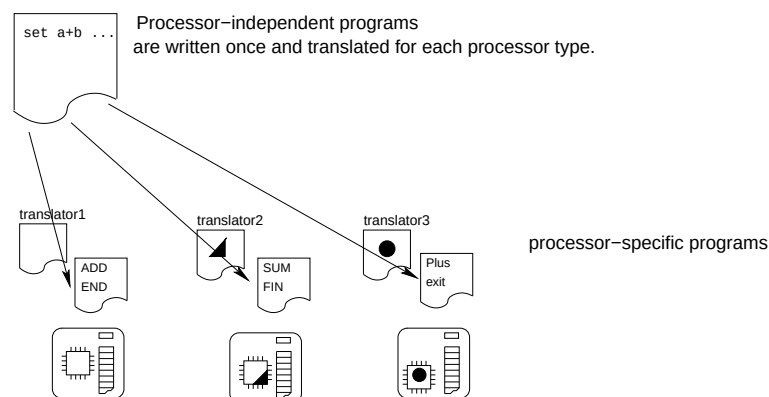
Languages for easier Coding — Since programs written with pure processor-commands cannot be run on other types of processors, so-called high-level languages have been designed to allow formulating processor-independent program-code. This so-called high-level program-code (or **source-code**) is translated into sequences of processor-commands (the **machine-code**). The two major advantages of this strategy are the processor-independence of the source-code and the fact that the translator-program has to be written only once for each processor-type. Java represents one language among other high-level languages like Pascal, Object Pascal, Ada, C/C++, Fortran, Basic, Cobol. The hex-program above, becomes much more intelligible when written in such a high-level language:

```
public class ExampleProgram
{ public static void main( String unused[] )
  { System.out.println( "Hello World!" );
  }
}
// The source-code above prints the string "Hello World!" into the standard output,
// usually this standard output is a terminal-window.
// The class with the name ExampleProgram is declared inside the outer level of braces.
// The so-called main()-method makes the class a program.
// The main()-method is called by issuing
// the command java ExampleProgram inside a terminal-window.
// The main()-method contains the println()-command inside the inner level of braces.

/* Typical dialog in the terminal-window:
[localhost:~] marycosway% javac ExampleProgram.java
error: cannot read: ExampleProgram.java
1 error
[localhost:~] marycosway% cd JavaCodeFiles/ComputerModels
[localhost:~/JavaCodeFiles/ComputerModels] marycosway% javac ExampleProgram.java
[localhost:~/JavaCodeFiles/ComputerModels] marycosway% java ExampleProgram
Hello World!
[localhost:~/JavaCodeFiles/ComputerModels] marycosway%
*/
```

The high-level program above can be written and stored as a simple text-file (for more see page 71). The lines with the double slashes (// . . .) contain comments written into the program by the Java programmer, these comments are ignored when the program is translated into machine-language for running. The user's input and computer's output inside the terminal-window of the native operating-system is printed out above inside a multi-line comment (/* . . . */).

The following strategy can be applied to make program-writing a processor-independent task: Design a processor-independent language, write a translator that transforms processor-independent language-constructs into sequences of processor-commands. Then all other programmers need only write programs in the processor-independent language. And if the processor changes, then *only the translator needs to be rewritten for the new processor*.



So, high-level programming-languages have two advantages over programming with processor-commands:

- Processor-independence frees the programmer from machine-related technicalities.
- Problem-oriented language-design (Fortran for calculations, Cobol for business programs) often makes programming easier.

But programming in processor-commands often remains the solution that makes the programs run faster. Programmers who are programming translators and other hardware-specific programs may be called **system-programmers** because they predominantly solve problems that are specific to the computer-system they work on. Whereas **application-programmers** try to use the computer in solving real-world-problems.

Programming: The process by which a person (or a group of persons) develops a program, to be run on a computer.

Application-Programming predominantly describes writing programs for solving real-life-problems, outside the domain of the computer-platform and outside the domain of the computer-network. See also the notion of an “application-program” below.

Infrastructure-Programming This may include writing helper-programs that make application-programming easier. That may include an event-routing program (page 633), an implementation of a cryptographic algorithm (ROT, page 498), a file-compression program, as introduced on page 325, and similar tasks for solving secondary problems which occur in applications.

System-programmers concern themselves mainly with hardware-specific problems, like writing an operating-system, driver-software for some new hardware, or implementing a compiler for a given programming-language.

This text tends to favoring the direction of application-programming.

Application: 1. A program for solving such real-life-problems. A text-formatter or a database of addresses, are examples of **applications**. End-users usually work with applications. (More on applications is said beginning with page 512.)

2. A Java application is a Java program, characterized by having a `main()`-method, that does something related to the end-user, from printing a string “Hello World” onto the screen, to a fully-developed Internet-browser. A piece of Java source-code becomes an application when containing a `main()`-method, otherwise it can be seen as a software library. Many pieces of source-code provided in this text are rudimentary applications. For one of the simpler examples see page 37. Non-applicative programs are operating-systems (page 43), file-systems (page 46) and drivers (page 508) which represent software that organizes the inner workings of a given computer.

Language-Processing: Interpreting, Compiling, Virtual Machine

Language-Processing: Interpreting, Compiling, Virtual Machine — Human readable high-level code, as presented in the ExampleProgram above, is given as a text-file and has to be translated into processor-instructions to become a runnable computer-program. This human readable, high-level code also is called **source-code**, because this is the place in which the program is initially written.

Source-Code: Human-readable text that represents a Java program or any other program written in a high-level language. In this text, pieces of Java source-code are printed in typewriter-style.

There are imaginable different methods of translating the source-code into processor-commands: Translators, that translate the source-code each time the program is run, are called **interpreters**. Interpreters for high-level languages tend to let the programs run slow, therefore another approach to translating high-level programs had to be devised:

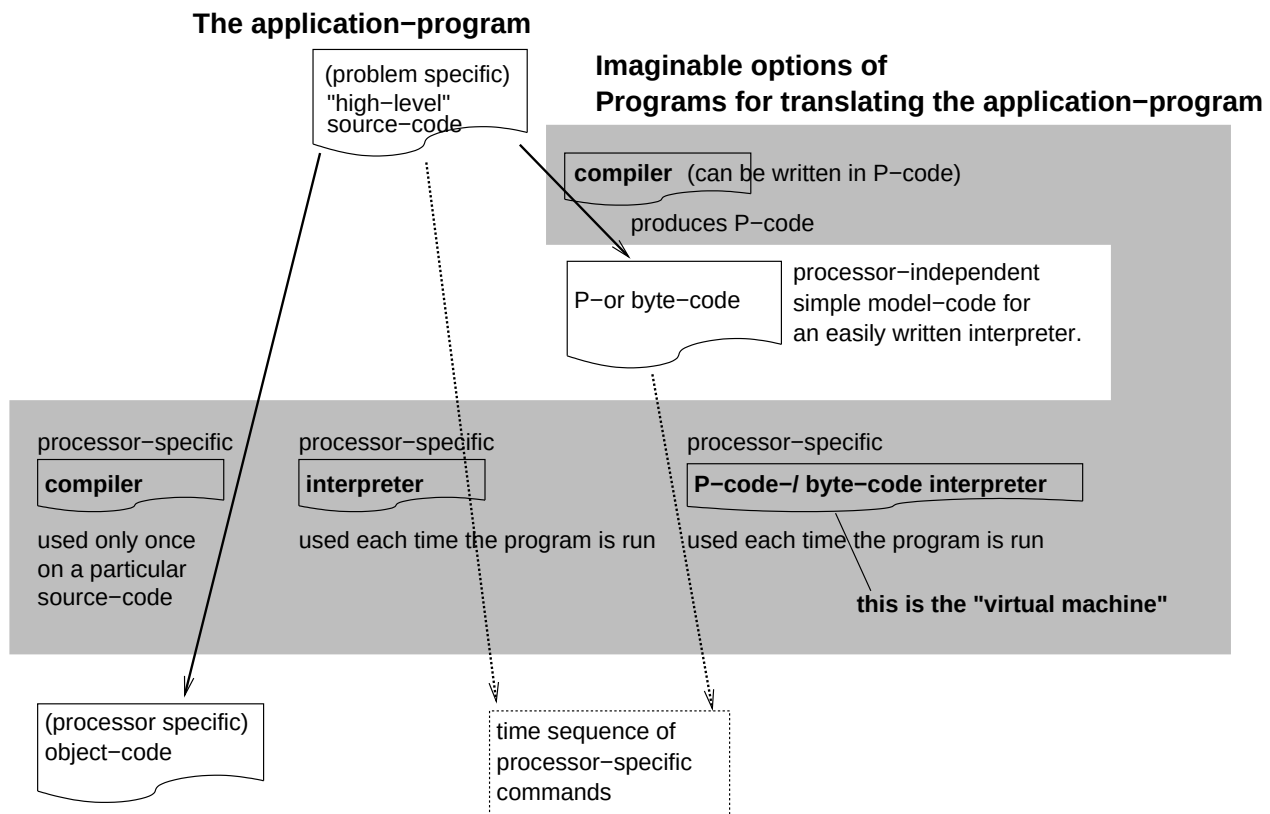
There exist translators just producing a file that contains the processor-commands which realize the functionality intended by the source-code! Those translators do not run the program which is given by the source-code of the high-level language. The translated code, which then has become processor-specific, is also called **object-code** or machine-code. Now, the file of processor-commands can run on the processor on its own, the translator was only necessary once to create that file. Such a kind of translator is called **compiler**. Remember that programs which can be run directly on the processor are identified by the term object-code or machine-code.

Thus, there are several ways in which the translation of source-code to object-code is achieved:

- One operational approach means to transform the source-code into object-code and then being able to run the object-code arbitrary many times (independent of what happens to the source-code). Translators of this kind are called compilers.

- There also can be imagined a kind of translator that, while scanning the source-code, produces a flow of commands that is worked through by the processor (quasi-)concurrently with translating the source-code into that flow. The users or application-programmers get the impression of the source-code running directly on the computer. Each command of the produced flow is discarded after having been worked over, so this kind of translator produces no object-code; each time the program is run its source-code has to be translated (quasi-)parallel. This kind of translator is called **interpreter**. Due to the organizational overhead of concurrent translation, interpreters usually produce a slower program-run than compilers.
- Writing compilers or interpreters for high-level languages can be a very demanding task, and rewriting these kinds of translators for every new processor-type may prove error-prone. A solution to that problem can be found by using a processor-model, to write a compiler once and be able to continually test it and improve it for that processor-model. Then remains the final translation of code from the simpler processor-model to the actual processor. This final translation is done by a small and relatively easy-to-be-(re-)written interpreter or compiler. (Each time the processor has to be changed, only the simple translator from processor-model to actual processor has to be rewritten.) The processor-model is called **virtual machine**, sequences of its commands that form programs are called **pseudo-code** or **P-code**. The design of such a virtual machine just follows a given convention.

In the graphic below are visualized the various ways for translating a program, written in a high-level language, into a sequence of processor-commands. The translation-process has to be achieved by programs (called interpreter or compiler) usually written directly in processor-commands:



As can be seen, the third concept makes it possible to write complex compilers once in a simple pseudo-code. And while being able to reuse the compiler, having only to rewrite a simple processor-leveled interpreter. A slight variation of this concept, where the code is run parallel to its translation into object- or pseudo-code, is called Just-In-Time code generation.

Interpreter: Program that translates other programs into a runnable form and runs them during the translation-process. (Advantage over → compilers: A user may issue one single command to make the program run. Disadvantage: The integrated translation-process lets the program run relatively slow.)

Compiler: Program that translates mostly text-files of source-code into a runnable form and stores the result in a file. The actual run of the program may occur later in time. (Advantage over → interpreters: Faster, since no translation during program-run becomes necessary. Disadvantage: Two-step-process of getting a program running: 1. Translation 2. Run. This consumes time during program-development.)

byte-code,
virtual
processor

The third concept has been adopted by the Java language: In the context of Java programs, the Pseudo-code is called byte-code and the byte-code interpreter, which emulates or rather implements the Java Virtual Machine (JVM), acts together with the original processor as a so-called virtual processor.

Java Virtual Machine (JVM): 1. The processor-model that forms the center of every → Java Runtime Environment (JRE). The processor-model that existed before any Java-enabled hardware was produced and which often is emulated by software on other processors.
2. The implementation of that specification (often called Java Virtual Machine too) comes into existence as a native program in a specific operating-system that translates Java → byte-code to processor-commands of the microprocessor-hardware of that computer. Since that translation-process usually includes the immediate run of the Java byte-code, this implemented Java Virtual Machine generally works as an → interpreter for the Java byte-code (and not as a → compiler).

Reconsider the graphic above: What is necessary to run a program? An interpreter always needs the source-code of the program. Whereas, if the source-code has been compiled into processor-specific object-code, then ONLY the object-code is needed for running the program on a given computer. In the third case, the pseudo-code (in Java, the byte-code) only can be run if there exists a pseudo-code-interpreter for the computer.

Languages: High-Level-Concepts

Languages: High-Level-Concepts — A prescribed coding format, like the Unicode-format makes it possible to assign “meaning” to a sequence of bytes, as described beginning with page 33. A sequence of bytes, marked by a four-byte-header reading “ca fe ba be”, is recognized and treated by the implemented Java Virtual Machine (JVM) as Java byte-code (page 36). In a similar way, other high-level languages mark their data to know what kind of data has been stored as a sequence of bytes. The implemented Java Virtual Machine, essentially is just the byte-code interpreter which transforms byte-code into a sequence of processor-commands.

variable

(VARIABLE) Most environments of high-level programming-languages introduce the concept of a **variable**:

- The actual data has to be furnished usually in literal form coded into a sequence of bytes.
- The type-name tells what kind (number, character, string) of data can be stored AND furnishes methods for its manipulation (addition and multiplication for numbers, concatenation of strings and extracting substrings).
- Finally, a name, assigned to that chunk of data, for accessing and storing the data in the high-level environment. Such a name is also called variable-name or (variable)-identifier. For more see page 73.

type =
encoding +
means of
manipulation

variable = data
+ type + name

Variable: Within a high-level language the concept of a variable is given by three aspects: Data, type and name. The name usually includes the area of validity of this name within the source-code or the runtime-environment; such an area of validity is called → scope. A type usually differs from a simple format in that it additionally furnishes methods for data-manipulation.

(JAVA, A PROCEDURAL PROGRAMMING-LANGUAGE) There are several criteria according to which high-level programming-languages can be classified. Procedural programming resembles the writing of an instruction manual for the computer; this may include, for example, making the computer sort a list of names. A selection of some procedural programming-languages is listed as follows: **Java**, C++, C, Pascal, Object Pascal, Ada, Fortran, Basic.

Non-procedural programming is focused on introducing, describing or selecting data. Some non-procedural languages are concerned with knowledge-coding and -retrieval like Prolog, or database maintenance like SQL (Structured Query Language) for relational databases.

All these types of programs use symbols to convey the meaning of their commands, like manuals that code their information by using printed letters. This is also called symbol-oriented programming. (Non-symbolic information-processing can be done in (symbolically) programming neural-networks and train them, for example in recognizing handwriting or faces.)

(ALGORITHMS AND PROGRAMS) A procedural program can be seen as a printed recipe for making a soup. The printed recipe can be written in different languages (English, French, Dutch, ...) nevertheless when cooked, the soup remains the same. The recipe itself, ideally, needs no changes when translated to another language. In computing, there are standard recipes for solving problems like sorting a list of

items according to their values (Quicksort, and others); these recipes remain the same independently of the language in which they are implemented. A method for solving a problem, formulated independently of its implementation by a programming-language, is called **algorithm**. If a book wants to stress its goal of showing how to reach results by using a computer for tasks like sorting, calculating numbers, . . . , without necessarily binding itself to a particular programming-language, then it will speak of algorithms when presenting solutions.

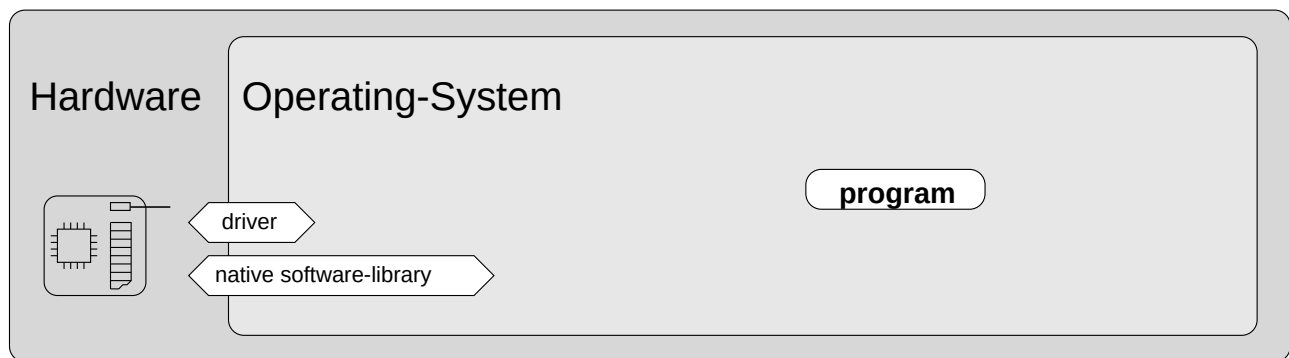
In this text, only the simple cryptographic ROT-algorithm is presented around page 478, before formulating the associated Java program.

Operating-Systems

Operating-Systems — Let a computer be given as a pure piece of hardware: Any program supposed to run on that computer, would have to organize its use of Random Access Memory (RAM), the input- and output routines (keyboard, mouse, monitor, hard-disk reading and writing, ...) and much more reoccurring hardware-related tasks.

Usually, these common basic tasks are programmed once for the given hardware and are given a hardware-independent (application-) programming interface for further use. This strategy resembles that one described with higher languages: These hardware-specific utility-programs, also called Application Programming Interfaces (API) give application-programmers an overall hardware-independent standardized way of handling the computer.

An integrated collection of these pre-programmed basic hardware-related tasks — this lowest-level-program which provides these services constantly to the user of the working computer — is called **operating-system**. On a computer-platform, the environment of the operating-system presents itself usually when operating-starting the computer session, by establishing a terminal window or by depicting a desktop on the system computer-screen.



hardware + operating-system = native platform

Hardware-specific (smaller) programs for establishing connections to peripheral devices, like a CD-ROM, are called **drivers**. Hardware-specific programs for making hardware-functions accessible by easier-to-use commands are collected in native software-libraries. If software-libraries are given standardized commands, which also may be implemented within other hardware environments, then those software-libraries with a standardized set of commands are also called **Application Programming Interfaces (API)**.

Application Programming Interface (API) An API tells a programmer how to use services provided by the given, pre-written code. Usually API's suggest a sort of standardized selection of functionalities.

1. An API may just denote a convention for using a given, pre-written software-library. 2. API may be called the code of such a software-library.

An API may ease working with hardware but an API or the API-implementing software-library may also furnish services which are hardware-independent: There are API's for sorting lists (`java.util.Arrays`), API's for transforming conventions for measuring and representing time (`java.util.Calendar`) and much more.

Thus, operating-systems usually provide more or less standardized, basic software-libraries or even a standardized one, an API. Moreover an operating-system

- begins to work when the computer is turned on and ends when the computer is turned off,
- gives the user and application-programmer a relatively hardware-independent environment, and
- shields the user and application-programmer from too many hardware-specific routine tasks by providing standard software interfaces.

Initially custom programs called "operating-systems" organized the management of

- storing, retrieving and cataloging files for the user. Then they
- assigned memory to programs that were running.

Now operating-systems

- detect and reconfigure external devices like printers or scanners,
- they assign processor-time and memory to programs so that they can run (quasi-)parallel. They manage the connections to networks.

These are some examples of resource-management of the operating-systems.

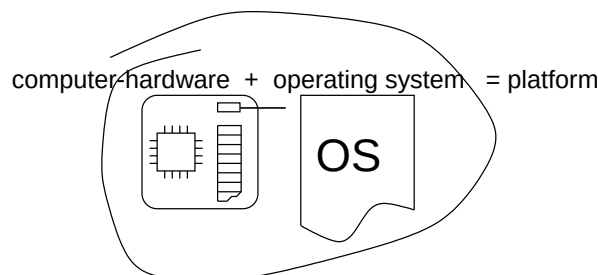
Resource: Examples for resources are printers, scanners, hard-disks, exchangeable disks, programs, documents. A resource may be some hardware-device or a piece of software (program, class or data). Processor-time or assigned network-capacities also are taken to be valuable resources. The denotation resource stands for anything that is related to computing and may furnish some service.

As could be seen above, there have been developed different high-level programming-languages, similarly there have been developed different operating-systems: BeOS, Linux, Mac OS, Unix, Windows. Programs that run in one operating-system usually do not run in another, this is also called lack of portability.

Operating-System (OS): A program, automatically organizing the use of the computer's resources (processor-time, memory allocation, keyboard and mouse-signal interpretation), thus supporting application-programs. The operating-system starts when the computer is turned on and stops when the computer is turned off. The operating-system acts as a supporting runtime-environment for the programs that run in it.

The hardware-type of the computer and the operating-system taken together constitute what is called a **computer-platform**.

computer-
platform



Platform = computer-hardware + operating-system

Examples for computer-platforms are compiled in the following table:

Operating-System (=OS)	hardware-type		produced by
Mac OS	Apple Computers	classic	Apple (www.apple.com)
Linux	Intel-based hardware	stable	public (www.linux.org)
BeOS	Intel-based hardware	fast	Be (www.be.com)
Windows	Intel-based hardware	known	Mircosoft (www.microsoft.com)

Operating-systems can communicate with the user over command-lines on the screen: The user has to enter commands that have been documented in more or less voluminous books. Linux (a free derivate of Unix) is such an operating-system. — Today, most operating-systems furnish a **mouse-oriented Graphical User Interface (GUI)**: The screen functions as a desktop-like plane on which folders and files can be stored, opened and worked on: Positioning the mouse-pointer on a folder icon on the screen and clicking once or twice opens a window that shows the contents of the folder. Commands are entered in these Graphical User Interfaces by drawing menus with the mouse; by marking checkboxes with the mouse or seldom by actually entering key sequences from the keyboard. Up to Versions 9.x, the Mac OS was a pure implementation of that concept; now Mac OS X and the other operating-systems have a terminal (or shell) icon, by clicking it, the user can open a terminal-window offering a command-line-based communication: Inside the terminal-window appears a line – a **command-line** – which shows the input which

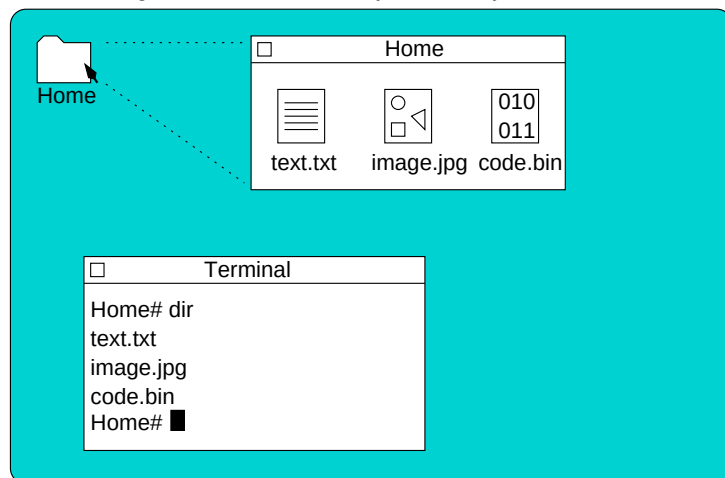
Graphical User
Interface

the user typed on the keyboard. Typing the RETURN-key marks the input as ready for processing by the computer. For more see the text beginning with page 507.

Pointing the mouse-arrow onto the "Home"-icon and clicking it opens the "Home"-window.

Clicking "text.txt"- or "image.jpg"-icons opens them to be edited or viewed.

Clicking the "code.bin" icon may run the object-code.



A Graphical User Interface

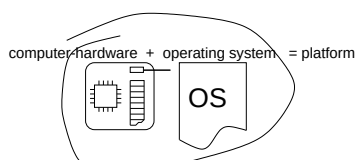
Typing "dir" lists names of files.

Typing in those file-names opens or runs these files.

A colored box (=cursor) shows where the computer takes a keyboard-input.

Some general aspects of operating-systems cause equally general problems:

(Extensibility-Problem) Unfortunately many operating-systems are not designed to support a broad range of existent hardware-types. Therefore, often processor-specific, assembler programs which run on single types of processors with unique hardware surroundings have to supplement the operating-system. These supplements extend the operating-system and are also called plug-ins, patches or drivers. For example, such patches may allow the operating-system to use the USB-card for addressing any USB-device (USB=Universal Serial Bus). Some operating-systems (with a "micro-kernel", like Unix-based ones) are designed to use patches or modules as a standard means for configuration, thus being able to adapt to the specific hardware configuration.



(Portability-Problem) IDEALLY writing platform-independent programs in a high-level programming-language should be no problem. These programs should, properly translated, run the same on different operating-systems; the same as they run within different hardware-environments. This property of a source-code is called **portability**.

portable
source-code

Usually, source-code of a high-level language, translated in differing operating-systems, produces programs which run deficiently. Why? Different operating-systems use different ways to address their resources, like threading, input/output routines (keyboard, monitor, mouse), they use differing native software-libraries, or even differing Application Programming Interfaces (APIs). The above-mentioned extensibility-problem contributes to this basic portability-problem. Many programs only can be used on the computer architecture (hardware plus operating-system, also called "platform") they were specifically designed for. That means, most programs are "incompatible with another operating-system" or cannot be "ported to another architecture without being severely modified". As can be seen below, incompatibilities even begin on the level of path-names that identify files. Therefore rewriting program versions for different operating-systems becomes tedious and error-prone. A program, written to use the USB-connection, will only work if the current operating-system supports the use of USB-hardware. So, almost all programs that are written in the same high-level programming-language (C, C++, Pascal, ...) have to be adapted to different operating-systems. Exactly this portability-problem tries to solve the Java Runtime Environment (JRE), as described beginning with page 51.

There are some general strategies to circumvent portability-problems:

- Cross-compilers are translators that can translate one given piece of code into a program for another processor.
- The Java computer-model furnishes a high-level language to program with. But the Java computer-model also furnishes a runtime-environment with its own Application Programming Interface, to

run these programs. So, ideally, Java programs are supposed to survive not only processor-changes but also platform/architecture-changes. Thus, such a Java Runtime Environment (JRE) constitutes a virtual computer-platform.

So, Java not only uses a high-level language for writing processor-independent source-code, but Java also furnishes Application Programming Interfaces independent of an operating-system. Thus striving to establish a runtime-environment (a virtual platform), which works independent from native implementations, and thereby being able to produce truly portable code running the same in Java Runtime Environments (JREs) on different computer-platforms. For more see the next part of this text beginning with page 51.

File-Systems

File-Systems — Programs are stored the same way as other data: As files either in the Random Access Memory (RAM) or on some persistent storage-medium like a hard-disk or a removable disk. Such persistent storage-media are an answer to the following request: Although the content of the Random Access Memory is lost after the computer's power has been turned off; each time the computer is restarted, data and programs shouldn't need to be reentered from keyboard into the computer-system!

Programs are sequences of bytes to be found, for example, in the Random Access Memory. Such a sequence of bytes can be given a name and be stored persistently on a hard-disk. A sequence of bytes that is given a name is called a file. The format in which are saved name and number of bytes (including data like the date of last saving, ...) is prescribed by the file-system, which has to be supported by the operating-system.

File-System: 1. The convention according to which files are made accessible or persistently storable. There are hierarchical- and flat file-systems.
2. The piece of software that makes the existent files of a computer-platform accessible (especially the persistently stored ones). An end-user can see a file-system just as a special database with hierarchically organized file-names and the associated files. All common operating-systems usually support at least one type of file-system.

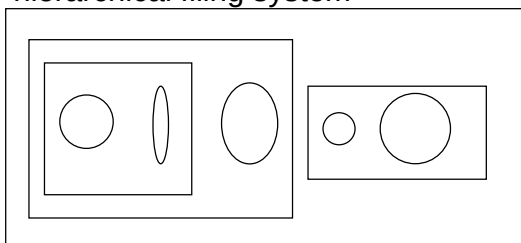
There are many types of file-systems for system-programmers, differing in technical specifications (ReiserFS, hfs, vfs, ...); an application-programmer or an end-user usually gets along well with knowing the basic idea of those Hierarchical File Systems:

(Concept: Hierarchical File System) Imagine a set of boxes and toys. Put toys into boxes, boxes and toys into larger boxes until everything is stuffed inside the largest box. Multiple toys can be in one box enclosed by multiple other boxes. A toy can be in a box next to another box. The listing, that indicates what boxes have to be opened to get a specific toy, characterizes a Hierarchical File System.

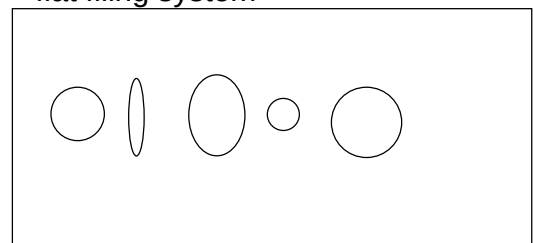
One single box, that contains all the toys, needs only a list of toys to get the information of the content of the box. Such a list is also called a **flat file-system**.

squares are boxes, circles and ovals are toys:

hierarchical filing system

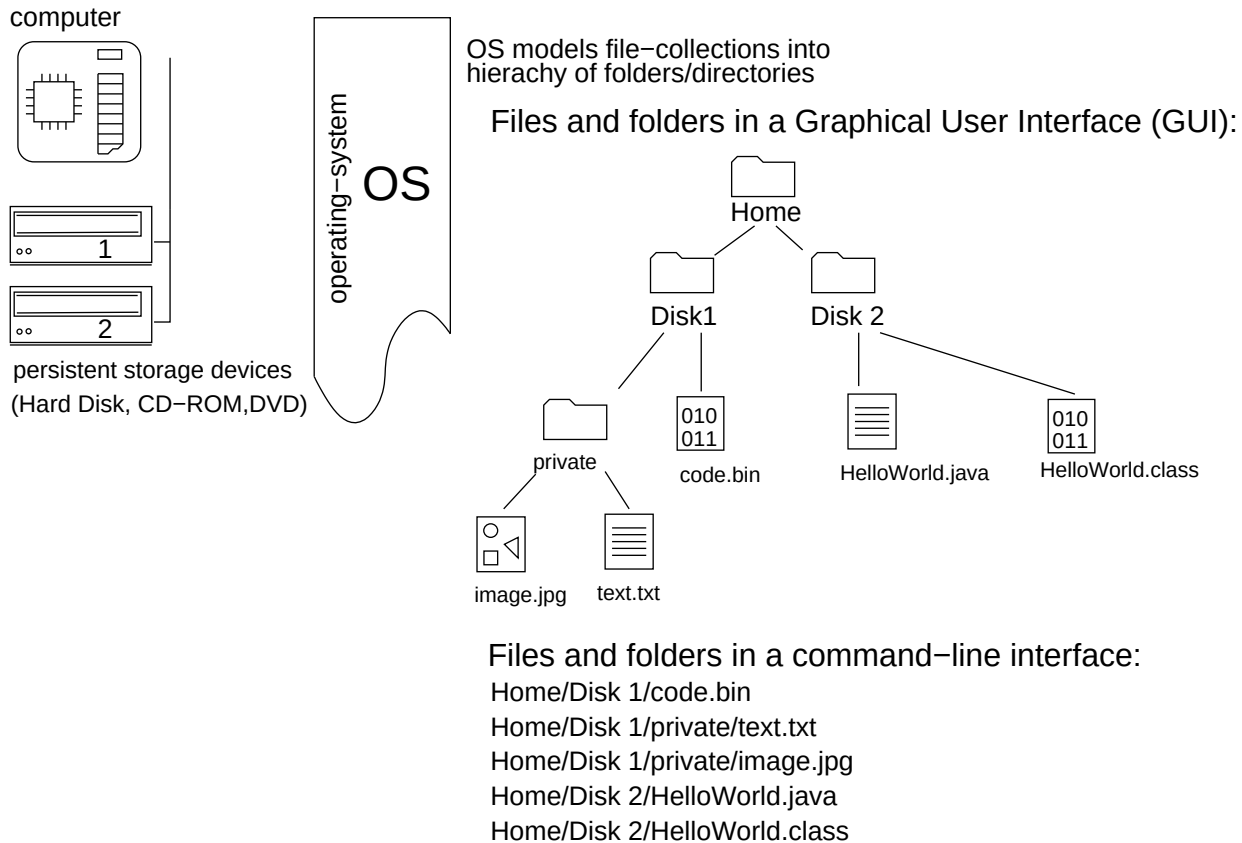


flat filing system



As a practical example consider four files named "image.jpg", "text.txt", "code.bin", "HelloWorld.java", and "HelloWorld.class". With a single command like "dir" (in DOS) or "ls" (in UNIX), these five files can be easily overseen by the user, but things become more difficult with four-hundred or even four-thousand files. Therefore, the names of individual files can be prefixed with a separator-symbol (usually a slash "/" as used in the graphic below) and other names that act like surnames. A simple and efficient method of prefixing file-names results in the so-called **Hierarchical File System (HFS)**: The overall naming-pattern is given a tree-structure that makes searching for the user or the computer more efficient. The prefixing parts of the file-names are called **directories** or **folders**:

Hierarchical
File System
(HFS)
directory or
folder



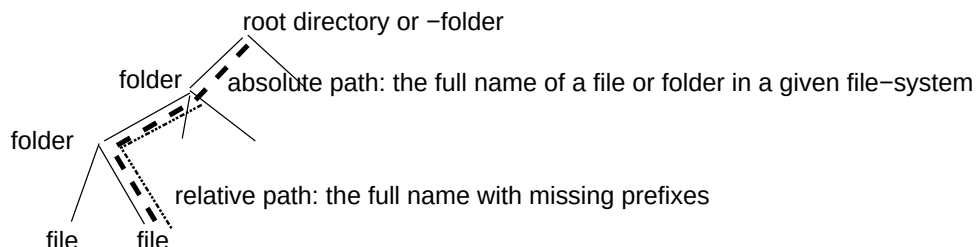
The lines in the graphic above indicate a containment relation. Observe also that the structure of files and folders is the same one as visible in the introductory graphic, which introduced the hierarchical filing system by using boxes and ovals. Try to relate ovals to the files.

The name-prefixes can be used as patterns for searching: In the example above searching using the command "ls Home/Disk 1/private" produces a list containing the file-names "image.jpg" and "text.txt". Such name-prefixes are also called **paths** or **path-names**.

paths or path-names

Path or Path-Name: 1. A slash separated sequence of folder- or directory-names. (The symbols used for separating the folder-names can vary with differing operating-systems, which constitutes an obvious source for portability-problems!) Often path-names of a Hierarchical File System (HFS) end with a file-name, those path-name represent so-called extended file-names.
 (2. Sometimes a concatenation of paths, where individual paths are identifiable by interspersed path-separators, also is called path-name!)

Path-names which do not begin with a home- or root-directory are also called relative path-names, because they usually are given relative to a specific folder or directory in the hierarchy of folders. For example "code.bin" can reference "image.jpg" by adding a relative path-name to the file-name: "private/image.jpg"; the resulting extended name also may be called path-name.



a hierarchical name-space

The notion of a path can be immediately transferred to the example of boxes containing boxes and ovals: An **absolute path** describes what - beginning with the biggest, all-enclosing box - sequence of boxes has to be opened to get a specific toy. Using a **relative path** assumes that some boxes are already open and, beginning with the current box at hand, the given "relative" sequence of boxes still has to be opened.

How to access files in Hierarchical File Systems from Java is shown on page 259. The concept of a relative path occurs as a variation when the idea of a relative URL is introduced past page 373.

relative path-names

Absolute path-name: An absolute path-name identifies a folder or file within a Hierarchical File System (HFS). An absolute path-name is just a path that begins with the root-directory of the given Hierarchical File System.

Relative path-name: Consequently, a relative path-name is a path-name that does not start with the root-directory. Starting from a given folder within the Hierarchical File System, for example a marked- or current directory, a relative path-name refers to another file or directory of the given Hierarchical File System.

By introducing a so-called **path-name separator** (here a colon “:”), multiple path-names can be collected in one string "private/image.jpg : private/text.txt". These string-constructs often also are called “path-name” and are used to help programs in searching needed data-files. The system-variable CLASSPATH, that is used by Java Runtime Systems (JRSs) of Versions below 1.2 to locate their core files, is such a kind of path-variable. Java Runtime Systems of Versions 1.2 and above, only use the system-variable CLASSPATH to find non-essential files.

Conventions for separators of path-names and prefixes vary between operating-systems:

operating-system	file/folder/directory separator	current directory	home-directory	path-separator
Linux/UNIX/MacOSX	/ (slash)	.	~	: (colon)
Windows/DOS	\ (backslash)	.		; (semicolon)
MacOS 9.x and below	: (colon)			/ (slash)

Examples: A Unix path-name, containing multiple paths, may read "private/image.jpg : private/text.txt". The same path-name in a Windows/DOS environment would have to be written "private\image.jpg ; private\text.txt". The path-name consists of folder- and file-names separated by file-separators. Paths are separated by using path-separators.

When working with terminal-windows, as indicated within the graphic on page 45, *one* directory or folder, temporarily, has a special role:

Current Directory: The user-chosen default directory inside a given terminal-window. (In Unix-related terminals, directories can be changed with the command “cd”, short for change directory.) Often the terminal-window shows the list of folder-names which form the path to the current directory. (In operating-systems of Unix-type, enter the command “pwd” to see current directory, also called the Present-Working-Directory)

Part II.

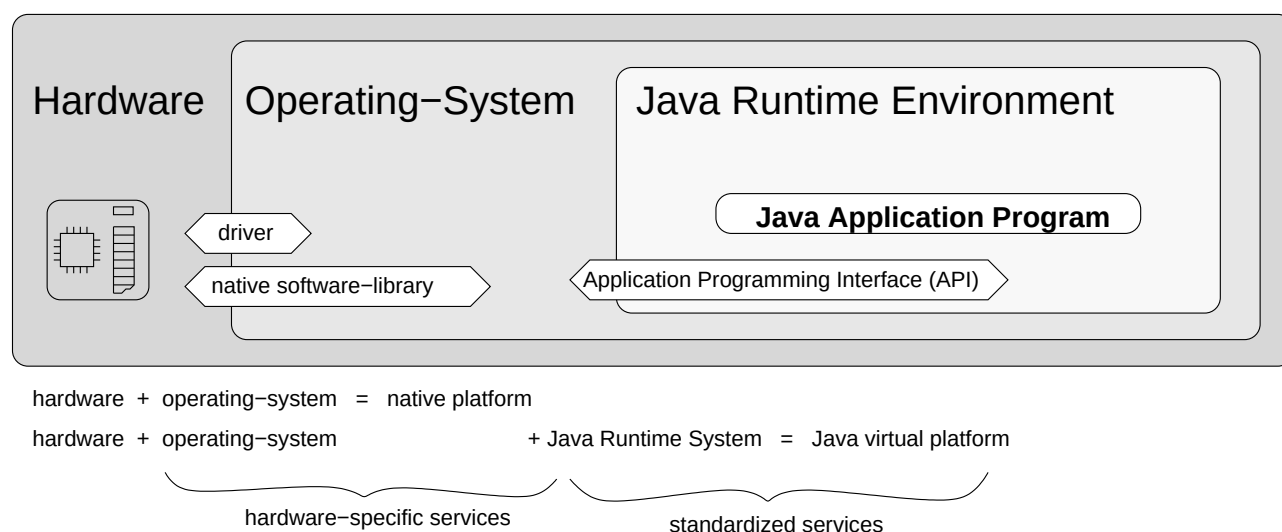
The Java Runtime Environment

The Java Runtime Environment — High-level programming-languages were designed to produce source-code that remains independent of the processor-type of a computer. Ideally what has been said about higher-level languages in the section beginning on page 37 should suffice to write truly portable programs. Source-code, once written, should be reusable on different, even newly developed processor-types by solely rewriting the translator for the source-code.

But there occurs an additional complication: Operating-systems often furnish their own specific input- and output-routines or file-system designs which are not covered by many high-level programming-languages. And there may be software-libraries (Application Programming Interfaces (APIs), page 43), often associated only with certain operating-systems (OSs). So, platform-specific software-libraries, and to make things worse, even language dialects (C/C++ variants), made it necessary to tediously adapt software to different platforms. This platform dependence of software is known as “lack of portability”.

As a possible solution to this portability-problem consider the design of an entire platform with standardized software-libraries (Application Programming Interfaces, API) and a standardized high-level programming-language. To emulate this standardized platform on another platform, a special emulating-software, also called runtime-environment, has to be written only once in the native code for each type of native platform⁷. Then, the programs, written for this standardized platform, do run the same in each runtime-environment independently of the underlying platform. Program and runtime-environment — the program written in a standardized language, the runtime-environment written only once for each type of platform — promise a truly portable program-code.

What is Java? A realization of the idea above. Java tries to solve the portability-problem by furnishing software emulating a standardized computer-platform on top of a given operating-system and its associated computer-hardware. This standardized computer-platform is also called “virtual platform” or **Java Runtime Environment (JRE)**. — So what’s essentially new with Java? Java models an entire computer-system to assure portability of programs written in the Java programming-language. Conventional high-level programming often lacked this universal approach.



The software — specifically written for the native platform — that makes it possible to establish a Java virtual platform on a given computer-system is also called the **Java Runtime System (JRS)**. This piece of software has to be installed onto the native platform, exactly then (portable) Java programs (Java applications) can be run. Thus a Java Runtime System (JRS) contains an implementation of the Java Virtual Machine (JVM, Java’s processor specification), as a piece of software⁸ plus everything else that is needed to make Java programs running, especially Application Programming Interfaces (APIs), thus furnishing an entire virtual computer-platform. Java Runtime Systems exist, adapted once, for various native computer-platforms (Intel, PowerPC, plus Linux, Mac OS, Windows); once installed, any Java-code finds the same Java Runtime Environment (JRE) everywhere.

As soon as a Java program is started (this is usually done by entering the “java *programName*”-command into a terminal-window of the operating-system, graphic on page 45), a Java Runtime Environment (JRE) is started and the necessary Java base-classes are loaded automatically. So, the notion Java Runtime

⁷Faint analogy: High-level programming-languages had the advantage of needing a new translator-software written only once for each new processor-type. The programs written in this high-level language only had to be retranslated, not to be rewritten by the programmer!

⁸There also exists processor-hardware that implements (part of) the Java Virtual Machine (JVM).

Environment may especially be used for Java Runtime System in action. (There also may be multiple Java Runtime Environments running from different terminal-windows on the same computer-platform!)

Thus, Java emulates such a model-platform (hardware+operating-system) on another computer. This requires additional translation-time for the real computer-system, a disadvantage. The advantage is seen in the intended portability of Java programs. That means, if the specific computer-platform is changed, then Java programs should run unaffectedly. An example of this may be seen *Sun's* Java compiler, which is written in Java and thus has not to be rewritten for each computer-system.

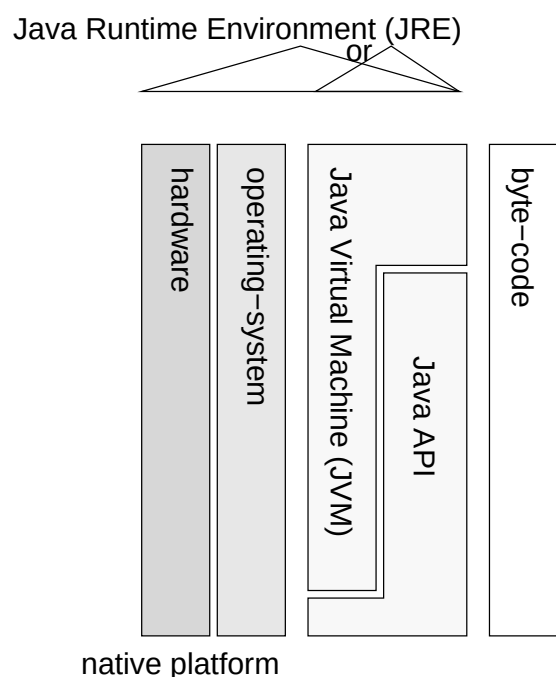
The Java model of a computer-system is given by:

Java CPU model This specifies a processor-type, the so-called Java Virtual Machine (JVM). This model-processor's functions are implemented by native software as a part of a platform-specific Java Runtime System. With this native translation software, Java processor-commands called "byte-code" can be run on any platform with such translation-software.

Java high-level object-oriented language Programs written in that programming-language are translated into byte-code by the Java compiler.

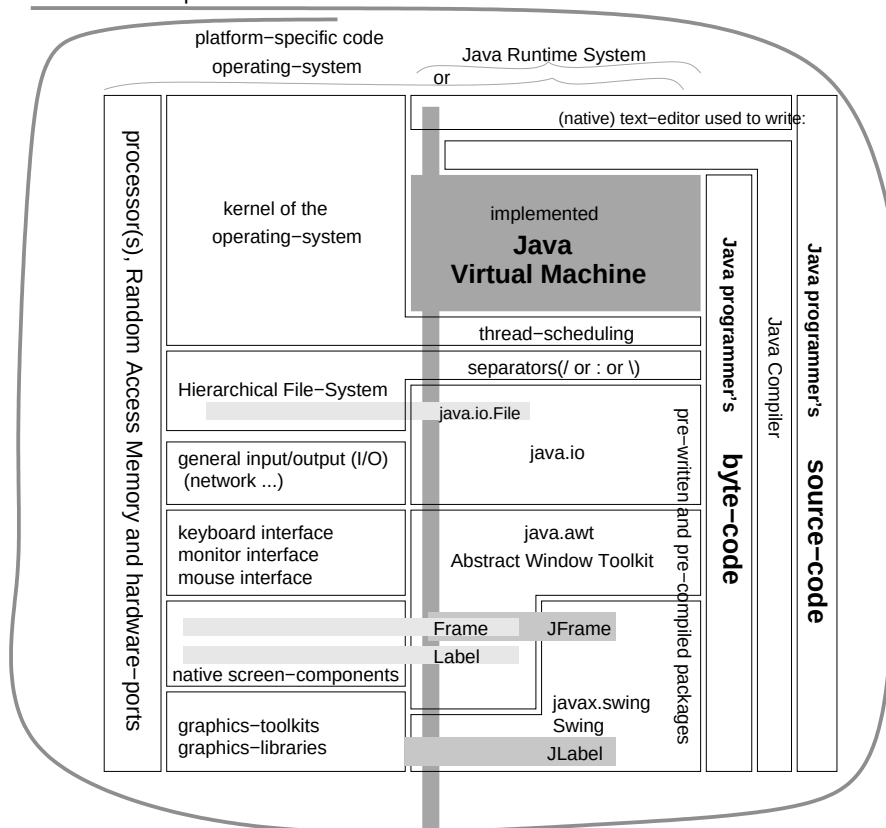
Application Programming Interfaces given by software libraries of standardized usage, as it is documented in the Java Platform API Specification [JPAPIS] available online from <http://java.sun.com>. For example, the Abstract Window Toolkit (AWT) and "Swing" support a Graphical User Interface (GUI) with keyboard-, mouse-, and monitor-windowing-system. Some of those software libraries use native code to keep contact with the native platform like the Abstract Window Toolkit with its native "peer-classes", other Java software libraries are said to be written entirely in Java and thus have to be fully portable, like the Swing-library. (Nevertheless Swing uses basic graphics routines of other Java libraries that use the services of platform-specific code of the Java Runtime System.) For more see the graphic below and page 548.

The use of the native file-system and threading-capabilities remains predominantly platform-specific as has been visualized in the graphic past the one below. The graphic below presents a layer-model of Java's virtual computer-platform:



A **Java Runtime Environment (JRE)** either is given by the software in which Java byte-code can be run or even by the entire virtual platform which lets the byte-code become runnable. A more refined model, with the focus on the Java Runtime Environment (JRE), is presented by the graphic below:

The Java Computer-Platform



As can be seen above, a programmer may write source-code, possibly even on a native text-editor, compile it with the Java compiler, and run the byte-code on the implemented Java Virtual Machine (JVM). Some pre-programmed packages, usually furnished as byte-code, may be of some help too. If the Java compiler itself has been written in Java, as *Sun's* Java compiler, then it also works platform independent. (The lower part of the graphic above is described more closely on page 548.) The Java Runtime Environment (JRE) can be seen as the platform above, taken without the Java compiler and the Java programmer's source- and byte-code.

Each specific type of computer-platform has to be given such a virtual Java platform equivalent, the so-called Runtime Environment. The software-files that have to be loaded or have been pre-installed, which also contain platform-specific pieces of software, are also called "**Java Runtime System (JRS)** for the specific platform" (Linux, Mac OS, Windows). The Java Runtime System (JRS) furnishes an implementation of the Java Virtual Machine (JVM) on a specific computer-platform together with support packages like the Abstract Window Toolkit⁹ (AWT, `java.awt`) and the Input/Output package (`java.io`). Such a Java Runtime System (JRS) is meant to produce a Java Runtime Environment (JRE) in an operating-system that supports a Hierarchical File System (HFS).

In the graphic above can be seen two cases of *platform dependence* which the Java design apparently couldn't avoid: If paths of the file-system are to be worked explicitly with, then they have to be given in the notation of the operating-system of the underlying, native platform. This marks one point at which the underlying platform shines through to the Java Runtime Environment (JRE). The other case is given by the original method of thread-scheduling which Java may reuse directly, for more see page 200.

Native Software and Portable Software: 1. Native software (programs and libraries) is written for a specific platform (=hardware+operating-system). Contrary to portable software which ideally can be used on different platforms alike.

2. In the context of a Java Runtime System (JRS): The platform-specific part of the Java Runtime System (containing the software of the implemented Java Virtual Machine (JVM) and supportive software-libraries) that ensures the runnability of Java byte-code. The portable parts of the Java Runtime System are many of Java's software-libraries (for example the Swing-package) which have been written in the Java language and most other Java programs. (Except those non-portable Java programs that use the Java Native Interface (JNI, not treated in this text) to incorporate native code.)

Usually, the word "native" is used for discerning platform-specific-code from (portable) Java byte-code.

The entire concept can be restated approximately as follows:

⁹The interaction of the specific platform with the Abstract Window Toolkit (AWT) is refined on page 548.

NEXT TO A VIRTUAL MACHINE, THE JAVA PLATFORM-DESIGN INCLUDES JAVA SOFTWARE-LIBRARIES (APPLICATION PROGRAMMING INTERFACES (APIS)) DESIGNED TO WORK PLATFORM-INDEPENDENT TOO. This intends to make Java programs truly portable between different platforms and operating-systems. That *overall* strive for platform-independence constitutes one of the important aspects of the Java design. The concept is that of a “Java Programming Environment”, instead of introducing just a new high-level programming language.

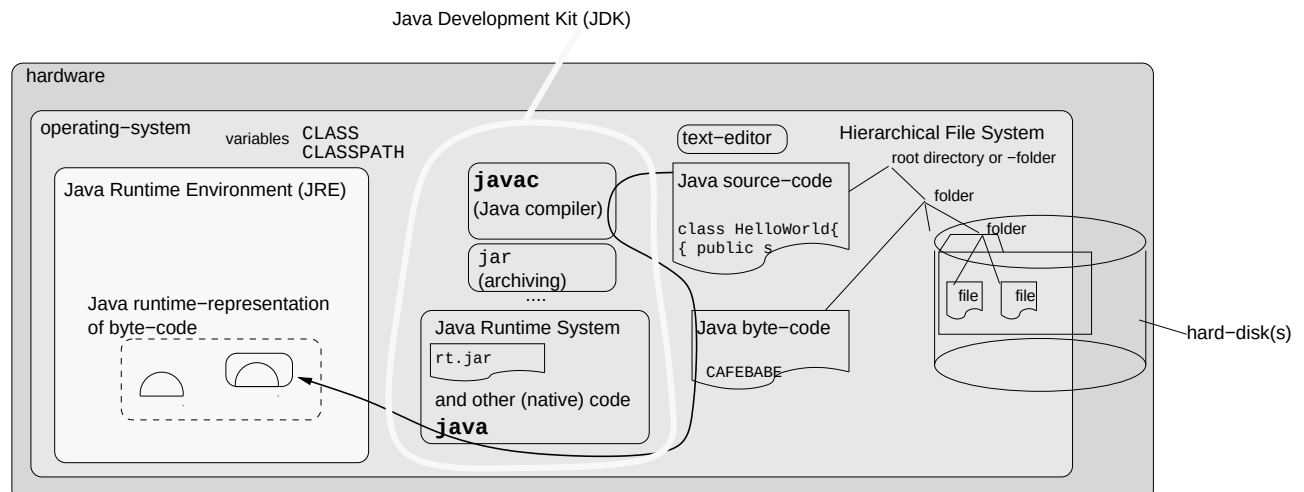
Java Programming Environment = Java Language + Java Software-Libraries

JAVA PROGRAMS WRITTEN IN THE JAVA PROGRAMMING ENVIRONMENT ARE RUN ON PLATFORM-SPECIFIC SOFTWARE ALSO¹⁰ CALLED **JAVA RUNTIME SYSTEM (JRS)**. THIS SOFTWARE RUNS AS A **JAVA RUN-TIME ENVIRONMENT (JRE)** DIRECTLY ON A NATIVE PLATFORM, OR WITHIN AN **EMBEDDED SYSTEM** (LIKE JAVA ENABLED INTERNET-BROWSERS), PRODUCING A VIRTUAL PLATFORM. For more on the Java Runtime Environment (JRE) see page 431.

To be able to even furnish a platform-independent compiler, Java uses a two-step translation-procedure. First, a Java application-programmer writes source-code. Using the Java compiler, this source-code is compiled into Java byte-code. Then this byte-code is interpreted into platform-specific code. For each type of native computer-platform exists a platform-specific Java Runtime System that includes a platform-specific implementation of the Java Virtual Machine (JVM).

All the necessary software that is needed to develop and run Java programs on a specific computer-platform comes in a Java Development Kit (JDK) for the respective computer-platform (Linux (and other Unix variants (Solaris, BSD)), Mac OS, Windows).

Java Development Kit (JDK) = Java Runtime System + Java Compiler (javac) + appletviewer + jar + other utilities



Java source-code can be written with almost any text-editor. The Java Development Kit (JDK) contains a Java compiler that transforms Java source-code into Java byte-code. The Java Development Kit also contains an archiving-tool (jar) which allows to aggregate many single files into one single big file. These are examples for the following general notion: Tools, also called utilities or commands, are pieces of software¹¹ of the Java Development Kit, enabling a user or an administrator to exert control over the resources and render services.

A central part of the Java Development Kit constitute the files of the Java Runtime System (JRS): — For example the JAR-file that contains the Java base-classes (rt.jar), Java’s central code-libraries. — Or, the (native) software that implements the Java Virtual Machine (JVM), which can be started by entering the “java”-command into the terminal-window of the operating-system. As part of starting the Java Runtime Environment (JRE), the “java”-command also triggers the transformation from files of Java byte-code into the Java runtime-representation of the byte-code, so that they are ready to be run (initial class-loading).

In the Java Runtime Environment (JRE) files of Java program-code, turned into a runtime-representation, become so-called classes (page 145), objects and packages (collections of classes with a common surname, on page 239). These notions are introduced later, here they have been depicted by a framed half circle (a class), a half circle (object), a dashed rectangle (package). The diagram above is reused on page 432 together with more information about the Java Virtual Machine (JVM) and its implementation in a Java Runtime Environment (JRE).

¹⁰Much like the low-level translators for higher-level programming-languages.

¹¹usually native, thus platform-specific

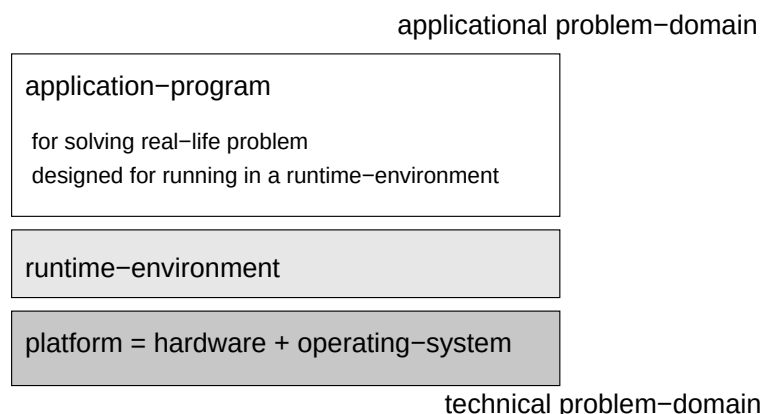
Base-Class also Core-Class: Pre-written Java-code (classes) furnished with the Java Runtime System (JRS). Those base-classes are part of the Java Application Programming Interface (API). Base- or core-classes are the central software-library of the Java Runtime Environment (JRE). Classes of the `java.lang`-package may be seen as arch-examples for base-classes.

Base- or core-classes usually have been collected into one single file within the Java Runtime System: In Java Version 1.2 and higher the file `rt.jar` usually contains the related files of byte-code. In other Java Versions or even other versions of Runtime Systems, the core-classes may be found in a file called `classes.zip` or even files bearing other names.

Nevertheless, many aspects of the Java design show a special care for platform independence. So, like large parts of the JDK are written platform-independently in Java (compiler, Swing-classes), the application-programmer too can produce Java programs to function truly platform-independent. This portability is thought to make rewritings in different dialects for different operating-systems unnecessary (C/C++ use different libraries in different operating-systems).

Runtime Environment: A PROGRAM THAT PRODUCES A FULLY-SPECIFIABLE CONTEXT (=ENVIRONMENT) for running other context-specific programs. Then, these context-dependent (non-native) programs are independent from the platform on which the runtime-environment runs, they are “portable”. The runtime-environment itself is given by a **NATIVE**, platform-specific, program that runs on a computer-platform and can acquire access to its resources. This program, the runtime-environment, has to be started on the platform to furnish the specific context needed TO RUN SUCH CONTEXT-DEPENDENT, NON-NATIVE, PROGRAMS. These (portable) programs then can concentrate on solving real-life-problems, which makes them so-called applications.

An operating-system can be seen as a runtime-environment which emerged from facilitating hardware-access and hardware-management. Runtime-environments and operating-systems are not meant for solving applicational or real-life-problems, they rather (and ideally) organize the computer’s resources or provide standardized programming environments to make application programming easier.



The notions “Java Runtime Environment” and “Java Runtime System” are often used synonymously! Although in this text, the notion “Runtime Environment” is used for the running virtual platform, whereas the notion “Runtime System” is used for the files of software, which are necessary to run the virtual platform on a specific type of native platform.

The name “**Java Runtime Environment (JRE)**” may stand for three different things:

1. **(running software)** The working software on a computer which makes it possible to run Java code/Java programs. A working example of the Java virtual computer-platform. Used in this text. — The Java Runtime Environment can load and run Java byte-code, manage memory with the → garbage-collector, handle exceptions with the implemented → exception-infrastructure. Usually the Java Runtime System begins its work after the end-user entered a “java”-command into a terminal-window of the operating-system. Multiple “java”-commands in multiple terminal-windows may run multiple Java Runtime Environments on one native platform.
2. **(specification)** The specification of the Java Virtual Platform. This includes the specification of the Java Virtual Machine (JVM) (the virtual processor of Java byte-code), the design of the Java core-classes (`java.lang.*`, ...) and other Java-related information needed to implement the Java Virtual Platform.
3. **(files)** The set of files which make it possible to install the software for running Java programs on a given computer-platform (Microsoft-Intel, Mac OS-PowerPC, Linux-Intel). Some of these files have to contain platform-specific, native, code. These files are also a part of the Java Developer Kit.
4. **(Altogether)** A computer on which Java code can be run.

Java Runtime System (JRS) 1. - 4. above. Here predominantly used in the meaning 3. and 4. Much in the sense that the Java Runtime System supplies the resources to produce a concrete, running implementation of a Java Runtime Environment. This includes the hardware, the software that is needed to make a file of Java byte-code a running program. This interpretation of the notion “Java Runtime System” is used predominantly in this text.

How can end-users and programmers control and see their interaction with the Java virtual platform? The following table lists some points of contact between a Java Runtime Environment (JRE) and the underlying computer-platform:

Operating-System (OS)	Java Runtime Environment (JRE)
-	- indicates the standard places of files.
“javac”-command compiles Java source-code into Java byte-code. CLASSPATH-variable	-
“java”-command into a terminal starts the JRE and Uniform Resource Identifiers (URI) entered into Internet-browsers start JRE and	- runs byte-code of applications.
Basic input- and output-functions for keyboard and screen	- run applet runtime-environment with byte-code of applets.
Resources like threads and the like	- furnishes a file of a package java.io (a code-library).
Native code	- are controllable from classes like System, Runtime or Process.
More advanced input- and output-functions for keyboard and screen	- referenced by Java identifiers assigned by factory-like-methods.
	- are given by the code in packages java.awt and javax.swing.
Files named java.security and java.provider	- furnish persistent data about security policies and encryption software.

For a shorter version of that table see also the list on page 642.

Java is designed to reach other basic fields: The Internet, microcontrollers and embedded computing.

Security: Any program, source-code or class-file, can be made downloadable in the Internet. Once downloaded and run, careless programs may disturb the platform on which they run. Basically this problem can occur with non-Java programs as well as with Java programs! BUT Java programs downloaded from the Internet, or coming from elsewhere (disk, . . .), are restrictable in their access to the Runtime System's resources (for example, the file-system or the printer): Java covers these problems by running so-called **applets** within secured applet-runners OR by running downloaded **applications** with an activated security-policy (see later in this text the part on security).

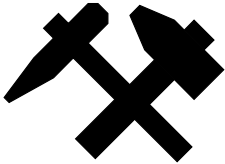
Microcontrolling: Mobile-phones/-computers, microwave-ovens, smart-cards, television-sets are examples for devices that contain microprocessors and often have a simple display and some means put them to use. Such devices usually work with programs written in BASIC or an assembly-language. The Java Runtime System (JRS) can be downsized to furnish most needs of such programming in naturally restricted environments.

Embedded Computing: Programs, especially applications, that temporarily activate and transfer (restricted) control over the system's resources are said to enable embedded computing. An operating-system represents the non-applicational prototype for embedded computing since it transfers control to every program that runs within, this even includes platform-emulators like the Java Runtime Environment (JRE).

Many of the previous concepts converge in the simple program which has been introduced on page 37: The source-code of the program can be written using a simple text-editor within the native operating-system. The outcommented lines at the end of the program's source-code do show input-lines and output-lines of some terminal-window in the native operating-system: An error-message, the changing of the current directory, a successful compilation (javac) and finally the run of the Java byte-code within the Java Runtime Environment (java).

The next chapter gives an impression of the necessary installation-procedures for having the javac- and java-commands available within a native platform's terminal-window.

Installing Java Development Kits (JDKs)



Installing Java Development Kits (JDKs) — The following chapter assumes to be working with a functional native platform (hardware + operating-system) for which exists a Java Runtime System (JRS). Depending on the COMPUTER-PLATFORM and the TIME and SPECIAL CARE taken, an installation procedure can be easy or disastrously time-consuming. Each paragraph, section or chapter which engages in describing technicalities of how to use a (Java-related) software-environment begins with one of the keywords “Organize” or “Install”. The icon, depicting hammer and chisel, marks installation procedures. The following has to be seen only as a coarse, probably even outdated, indication of how to install a Java Runtime System on a native platform. Up-to-date installation documentations can be found in the Java download- and installation pages (➡️➡️ <http://java.sun.com>) on the Internet.

Preparations: Generalities on (Downloading) and Installation

Preparations: Generalities on (Downloading) and Installation — As usual, but especially here, just pick out those informations that may help. What is needed to make **Java-applications** run on a computer?

Native Platform (computer-platform = hardware + operating-system) First there should exist enough space on a persistent storage-device, like a hard-disk. For further reference consult again the Java download and installation pages (➡️➡️ <http://java.sun.com>) on the Internet.

Java Runtime System Then there has to be installed a Java Runtime System written for that specific computer-platform. Java Runtime Systems contain implementations of the Java Virtual Machine (JVM) and other native code that is needed to produce a fully-functional Java Runtime Environment. Among that native code can be found a Toolkit-code which connects Java-commands with native commands, giving Java control over the native Graphical User Interface¹². A programmer only uses Java-specific commands of the Java Application Programming Interfaces (API). An end-user can create a Java Runtime Environment (JRE) and run Java byte-code therein by issuing the `java`-command in a terminal-window of the native operating-system.

What is minimally needed to **develop Java-applications** on a computer? A native platform and an installed Java Runtime System plus a

Java Development Kit (JDK) Usually Development Kits, either the ones that can be downloaded from *Sun Microsystems Inc.*, or bought somewhere else, contain a Java Runtime System together with a compiler (activated by the `javac`-command) and other tools (`jar`-command, ...).

Additional information provided from *Sun Microsystems Inc.* can be found in the JAVA DOCUMENTATION and the JAVA TUTORIALS of various kinds that are all available on the ➡️➡️ Internet, see the appendix on page 659. The original and up-to-date information about the class-libraries is given in the JDK (Java Development Kit) documentation that can be downloaded from ➡️➡️ <http://java.sun.com>. Further prototypes of programs and larger programs can be seen in the Java Tutorial which also can be downloaded.

The command

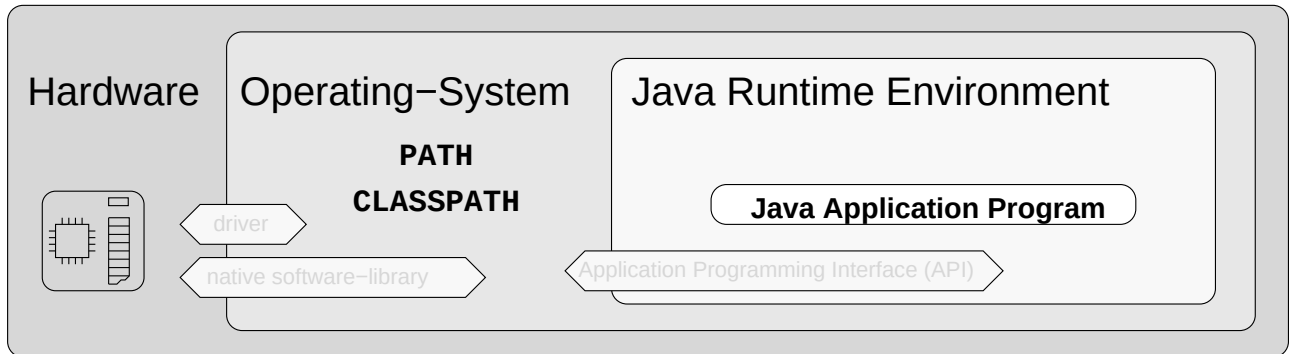
```
java -version
```

, issued in a terminal-window of the operating-system, tells if there has been pre-installed a Java Runtime System and prints out the version-number.

¹² Within the context of the Java Abstract Window Toolkit (AWT), an Application Programming Interface (API), the Toolkit-code, together with Javas peer-interfaces, accesses related platform-specific pieces of code, the so-called (native) peer-code.

PATH and CLASSPATH for Unix/Linux or Windows-based Operating-Systems

PATH and CLASSPATH for Unix/Linux or Windows-based Operating-Systems — Most operating-systems offer information about their state and configuration within ASCII-text-files (page 454) or alternatively by giving values of so-called system-variables. For configuration purposes, text-files may be editable and system-variables can be assigned values. Two variables of that kind are relevant for installing and working with a Java Runtime System: PATH for command-identification and CLASSPATH for code-identification. Both variables are collections of directory-paths in the file-system of the respective operating-system, as introduced around page 47.



PATH (for Commands)

Commands and their implementation as specific programs of the operating-system are the way of communicating from the operating-system with the Java Runtime System.

By issuing the terminal-command “`javac fileName.java`” the application-programmer compiles a given text-file “`fileName.java`” of source-code producing a so-called “class”-file of byte-code (“`fileName.class`”). And if that byte-code describes a Java-application, then that byte-code can be run by issuing the terminal-command “`java fileName`”. Application-programmer or user enter both commands into terminal-windows on the level of the operating-system of the (native) computer-platform.

The operating-system’s PATH-variable – a collection of paths in the native platform’s file-system – is used by the operating-system to search for those operating-system-specific programs (`javac` and `java`) that implement Java commands (an exception are the Mac OS versions with a number smaller than ten, where the programmer has to locate the command-realizing programs by her- or himself). — So new command-implementations can be introduced by either putting their code into a folder of the PATH-variable or by extending the PATH-variable suitably. The documentation of the Runtime System usually specifies how to change the PATH-variable of the respective operating-system. Or consider the hints in the following sections. Finally, properly (re-)editing the PATH-variable also makes it possible to choose one of several downloaded versions of Java Runtime Systems as the active version!

CLASSPATH (for Software)

Very much like the PATH-variable, the CLASSPATH-variable stores a collection of directories of the underlying operating-system. The operating-system’s CLASSPATH-variable designates additional parts of the given folder-file-hierarchy, where the compiler (`javac`) or the trigger for creating a Java Runtime Environment (`java`) may search for supplemental Java byte-code. The documentation of the Java Runtime System usually specifies how to change CLASSPATH-variables of the respective operating-system. Or, again, consider the hints in the following sections.

(Java Version 1.0.x/1.1.x) The Java versions below 1.2 need a non-empty CLASSPATH-variable to locate core code-files of the Java Runtime System.

(Java Version 1.2 and higher) With Java Versions 1.2 and higher, the Java Runtime System has been designed to be fully functional without depending on the contents of the CLASSPATH-variable. In these cases that variable is given as empty by default. Though the Java Runtime System does not depend on this variable any longer for loading core code-files, application-programmers can use it to let compiler (`javac`) or the trigger for creating a Java Runtime Environment (`java`) find application-related files:

After having been created, a Java Runtime Environment – or rather one of its (default) class-loaders – search for a class-file,

- (1. **Current Directory**) first in the so-called current directory,
- (2. **CLASSPATH**) then are searched all of the directories in the CLASSPATH-variable and
- (3. ... **Error**) finally a `java.lang.NoClassDefFoundError` is issued.

CLASSPATH-variable: A collection of directory-paths. Non-initial class-loaders of the Java Runtime Environment (JRE) search these directories and the current directory to find classes for loading (code- or class-loading is described beginning on page 433). How to see that variable and how to change it? See the following sections.

The operating-system's CLASSPATH-variable – a collection of various paths in the file-system – shows the Java Runtime Environment (JRE) where to search for additional Java code. That additional code is introduced by either putting it into a folder pointed to by the CLASSPATH, or by extending the CLASSPATH-variable with the file-system path that denotes the location of the additional code. Beginning with Java Version 1.2, the `ext`-folder, given by the path `path_to_Java_software/jre/lib/ext` also takes so-called extension packages (also see page 243).

Trick

For practical reasons it may be useful to put a folder onto the desktop that is defined to be part of the CLASSPATH-variable, then this folder will be actively searched by the Java Runtime Environment (JRE) for otherwise unknown classes. So a user or application-programmer can put those class-files inside that folder, which contain classes that are meant to be found automatically by the Java Runtime System (JRS). For examples of how the CLASSPATH-variable is used in the context of running Java-applications, see page 472 and within the context of installing packages, consult again page 243.

ORGANIZE: Apple Mac OS

ORGANIZE: Apple Mac OS — Apple's operating-system, Mac OS, had a Graphical User Interface (GUI) from its beginnings and did not need a terminal-window! On its introduction in 1984 even total beginners could use application-programs on an Apple computer relatively fast. The operating-system had the potential of making work on an applicational level almost purely intuitive. Beginning with Mac OS Version X (Mac OS X) the operating-system is Unix-based and thus there can be found a terminal-window! Furthermore, a Java Development Kit has been readily installed! Apple Mac OS versions below X may only support Java versions below 1.2! For more information see ➡ <http://www.apple.com/java>.

ORGANIZE: Mac OS 9.x: Installing the Java Development Kit (JDK):

The Mac OS Runtime for Java (MRJ) can be downloaded (about 8MB) from the Internet or found on a CD-ROM. The MRJ extracts itself and gets installed into the "Apple Extras" folder within a folder named "Mac OS Runtime for Java". In this folder also can be found an `AppleAppletRunner` with demonstration applets.

Programming in Java becomes possible with a second free software from Apple: The Mac OS Runtime for Java Software Development Kit (MRJ SDK). The MRJ SDK extracts itself and can be found installed inside the folder named "MRJ SDK" on the start-disk too.

ORGANIZE: Mac OS 9.x: Using the Java Development Kit (JDK)

Under Mac OS, the HelloWorld-program may be written with the omnipresent "SimpleText"-application and may be saved as a file named "HelloWorld.java". The "java"-terminal-command is implemented by the "JBindery" program in the "JBindery" folder.

javac: If entering the path to the source-code doesn't succeed ("error: Can't read: *fileName.java*"), then do prefix the source file(s) with the path (the path begins with a SLASH! AND folder-separators are slashes! Example: `/java/prototypes/HelloWorld.java`). If already existent classes are not found, then add the path, pointing to these class-files, to the system's CLASSPATH-variable.

Other Java tools (native utility programs) like `jar`, `javadoc`, `javah`, `javakey` and `rmic` are located in the "Tools-JDK Tools"-folder.

See <http://www.apple.com/java> for a free JVM for Apple (PowerPC-based) computers.

ORGANIZE: BeOS

ORGANIZE: BeOS — At the time of this writing there has been no official information about a JRS for BeOS. If still existent, see <http://www.be.com> for up-to-date information.

ORGANIZE: Linux

ORGANIZE: Linux — Linux (open source, <http://www.linux.org>), BSD Unix strains (<http://www.bsd.com>), Unix (from AT&T), Solaris (from *Sun Microsystems Inc.*, <http://www.sun.com>) are multiuser operating-systems that are very much alike. At the beginning of the 1970ies, Unix was written in the programming-language C (which provided similar functions as PASCAL and basic Java). Solaris and other Unix-like operating-systems were re-engineered, imitating Unix, by corporations that kept the source-code secret and licensed the operating-systems for money. Linux and BSD Unix are also re-engineered products but the source-code is publicly accessible under the GNU public license (<http://www.gnu.org>). The Linux development originated at the beginning of 1990 on personal computers which may explain its dominance on the sector of personal computers.

ORGANIZE: Linux: Getting the Java Development Kit (JDK):

On a Linux platform, download the Java Development Kit (JDK) from *Sun Microsystems Inc.* over the Internet (➡ <http://java.sun.com/products/>), thus acquiring the most up-to-date Java Runtime System (JRS). Or take the Java Runtime System from Linux-distributions (Redhat, SuSE) that follow the series-and-packet idea of the Redhat Packet-Concept. These Linux-distributions offer to install packets “java” (The Java Development Kit) of Series “d” (d for development), “javarunt” (The Java Runtime Environment) of Series “d” (d for development), and “javadoc” of the Series “doc” (doc for documentation).

Observe that a sole installation of the Java Runtime System does not furnish the capabilities of a Java Development Kit!

To see what has been installed use a Find Files utility (‘find /usr -name “*java*” -ls’ in terminal-mode or “Find Files” in a Graphical User Interface like KDE) and search for “*java*” or “*.class*”, “*.java*”. Most of the java-files may be found in subdirectories of the “/usr/lib/” directory. In that directory, search for directories like “java” or a directory like “jdk1.3” that indicates the version-number of the Java Development Kit (JDK) in use. Specific information concerning the operating-system and Java may be found in a README-file.

ORGANIZE: Linux: Using the Java Development Kit (JDK):

In Linux, the Java source-code can be written into a text-file with any simple text-editor and has to be saved into a chosen directory (for example “/home/userName/JavaCodeFiles/ComputerModels”) under a file-name with a “.java” suffix, like “HelloWorld.java”. In a Graphical User Interface (GUI) like KDE (see graphic on page 45) terminals can be opened from windows by searching for menu-choices like “File-Open Terminal” or “Tools-Open Terminal”. Then the Java terminal-commands can be given from a terminal-window (also called a shell).

javac HelloWorld.java compiles the Java program in the text-file HelloWorld.java creating a file of byte-code (HelloWorld.class). And the command

java HelloWorld starts a Java Runtime Environment in which that byte-code is run!

During the first try the compilation-command may cause an output:

```
javac: command not found
```

This should be remedied by extending PATH-variable of the Linux-system with a string of the following kind: “:/usr/lib/java/bin”.

(Command-Activation Linux) How to access system-variables in Linux? Type into the terminal-window: “\$PATH”, “echo \$PATH” or “env \$PATH”. Or read them out with “cat /proc/self/environ | tr \000 “\n””. The PATH-variable should look something like “/sbin:/usr/sbin:/usr/local/sbin/ ...” —

How to extend such a system-variable? Type into the terminal-window one of those three alternatives: “PATH=\$PATH:path”, “setenv PATH=\$PATH:path”, “export PATH=\$PATH:path” (no spaces around “=”!).

How to make such a setting persistent, that means making the setting constant over power-on power-off intervals of the computer? In a root account, the path changes could be entered as presented above in the `/etc/profile` or `/etc/bashrc` text-files and saved. If working as a non-root user, which is usually the case, then go into the home directory, choose the file-listing-mode that makes invisible files (indicated by a period(.)-prefix *.fileName*) visible and enter the changes into the `.bash_profile` or `.bashrc` text-files.

Another error-message may be of the following kind:

```
error: Can't read HelloWorld.java
```

Then a command like `cd /JavaCodeFiles/ComputerModels` changes the terminal-directory to the directory of the file `HelloWorld.java`.¹³ (**Class Search Linux**) If that doesn't help, then remember that the system-variable `CLASSPATH` of the native platform, underlying the Java Runtime Environment, stores a list of directories in which the compiler searches for Java source files (*name.java*) and in which the Java interpreter searches for Java byte-code (*name.class*). Java files located outside the current directory and are neither located inside one of the `CLASSPATH` directories won't be found by the Java compiler. How can the value of the `CLASSPATH`-variable be shown and changed?

```
# echo $CLASSPATH shows the value of the CLASSPATH-variable, which is usually empty with Java versions of Number 1.2 or larger.
```

```
# CLASSPATH=$CLASSPATH:path extends the CLASSPATH-variable (the colon ":" separates directories, the period "." marks the current directory, the directory of the terminal currently in focus and the tilde "~" marks the default-/user-/home-directory). This modification can be made persistent in the same way modifications to the PATH-variable has been made persistent.
```

If that doesn't improve the performance, then try to move the source-code into the `/root` directory or reinstall the Java packages. If that doesn't help, consult the support database of your Linux distributor or try to use their hotline.

Should the output be placed into a file with name `"out.txt"`? Then run the byte-code with the command `"java HelloWorld > out.txt"`

ORGANIZE: Windows

ORGANIZE: Windows: Installing the Java Development Kit (JDK):

The Java Development Kit (JDK) and its documentation (JDK-doc) should be downloadable separately from  <http://java.sun.com/products/>. Unpacking should be unproblematic.

ORGANIZE: Windows: Using the Java Development Kit (JDK):

In Windows, the Java source-code can be written with any simple text-editor (Start-Programs-Utilities-NotePad or ...-Editor) and has to be saved into a chosen directory under a file-name with a `".java"` suffix, like `"HelloWorld.java"`, beginning with Windows 98 the suffix may vanish. (For a first check just open the `"demo"` folder in the `"jdk version_number"` directory to run some pre-compiled applications or applets by double-clicking the appropriate icons.)

To get a terminal-window, also called DOS-window, onto the Windows-desktop, draw something like Start-Execute and enter `C:\WINDOWS\COMMAND.COM`.

```
c:\> javac HelloWorld.java compiles the Java program in the text-file "HelloWorld.java" creating a file of byte-code (HelloWorld.class) and the command
```

```
c:\> java HelloWorld starts a Java Runtime Environment in which that byte-code is run!
```

During the first try, the compilation command may produce the output:

```
javac: command not found
```

This should be remedied by setting the `PATH`-variable¹⁴ of the Windows system as to `c:/jdk1.2.2/bin`. If the following message appears,

```
error: Can't read HelloWorld.java
```

then the functionality should be assured by changing the directory into the one that contains the `HelloWorld.java` file. (the semi-colon `“;”` separates directories, the period `“.”` marks the current directory, as communicated by the table on page 48)

¹³See also the terminal-printout in the piece of source-code on page 37.

¹⁴How to access system-variables in Windows95? Draw Start-Programs-Utilities-NotePad and open the file `c:\autoexec.bat`, if there is no entry for `PATH` and `CLASSPATH` visible, then set those system-variables: Type into the file: `"set PATH = c:\jdk1.2.2\bin"` and into another paragraph, for example: `"set CLASSPATH=c:\jdk1.2.2\demo"`.

ORGANIZE: Development-Environments

ORGANIZE: Development-Environments — A simple development-environment consists of a text-editor, for writing the source-code, the Java compiler, for compiling the source-code into byte-code, and the Java Runtime System to run the byte-code on the native computer-platform. This simple development-environment, the Java Development Kit (JDK) together with a text-editor, can be seen in the graphic on page 54. Such a minimal configuration has been used in testing the source-code of this text. The Java programmer essentially controls this development-environment with three commands entered into a terminal-window of the operating-system:

text-editor: One class in a text-file <i>name.java</i> must be called <i>name</i> and be declared <i>public</i>
<code>javac name.java</code>
<code>java name</code>

So-called Integrated Development Environments (IDEs) usually install the Java Runtime System (JRS) automatically and make working with a large number of classes and objects easier. These commercial development-environments do closely work together with Java Runtime Environments (JREs) and often let the programmer choose comfortably between a range of different implementations of the Java Virtual Machine (JVM). And IDEs usually supply an integrated debugging feature. (Examples for development-environments are *Metrowerks*' CodeWarrior or *Symantec*'s VirtualCafe, *IBM*'s JBuilder)

Organize: Extending Java



Organize: Extending Java — Some extensions that are used in conjunction with the Java Development Kit have to be (downloaded and) installed separately. Java software comes in files. To be found by the Java Runtime Environment (JRE), these files have to be positioned in folders that will be searched by the Java Runtime Environment. This is always the case when these files are found in folders, pointed to by paths of the CLASSPATH-variable. Beginning with Java Version 1.2, some Java software has to be installed by being given as a *jar*-file and by being positioned into

an *ext*-directory (see page 243).

Or Java software like the Java Server Web Development Kit JSWDK or Remote Method Invocation (RMI) may require special installation procedures, like extending the PATH-variable to make new commands available in a terminal-window for

- starting the right helper-applications (mostly by typing commands into the terminal-window)
- in the right directory.

That may also include positioning (as links, as copies) the right files in the right folders or extending the CLASSPATH-variable. For the JSWDK example see page 405; for the RMI-example see page 414.

This ends the chapter about storing files of data on the native platform and about transforming these files into a working Java Development Environment, including a working Java Runtime System (JRS).

How Java Code is Run

How Java Code is Run — First check whether a Java Runtime System (JRS) exists on your computer-platform, for example by entering the command `java -version` into a terminal-window of the operating-system. If necessary, reconsider the previous chapter and the relevant section “Organize: Using the JDK”. — So assume that you have installed onto your computer a fully-functional Java Runtime System (an implemented Java Virtual Machine plus software library). Assume also, that the Java Runtime System (JRS) represents a part of a working development-environment. Such a development environment can be a Java Development Kit (JDK) from *Sun Microsystems Inc.* or any other product.

Compiling Source-Code

Compiling Source-Code — Assume further that you are given Java source-code by means of a disk or a CD-ROM or downloaded from an Internet-site, identify the file that contains Java source-code. Or use some text-editor to write a text-file¹⁵ that contains Java source-code:

fileName.java Name of text-file containing Java source-code.

Java source-code, to be compiled, is found in files with names suffixed by `.java`. The Java compiler can be told where the file is to be found by using an option and postfixing it with a system-specific path-name. The actual implementation of the user-command-interface varies as has been indicated in the previous section.

Generally a command “`javac -pathname[pathName] fileName.java”`, entered into the terminal-window of the native operating-system, activates the Java compiler, so that the text-file with the *fileName.java* is taken to be a file of source-code from which the Java compiler tries to compile a file of byte-code. (The optional value of *pathName* is given by a path inside the Hierarchical File System of the operating-system, the path pointing to the directory where the text-file can be found. If no *pathName* is furnished, then the Java compiler expects to find the file within the currently opened directory of the active terminal-window.) For more options furnished by the compiler and the interpreter, see the online-documentation, supplementing the ➡ ➡ Java Platform API Specification [JPAPIS]. — After the compilation has taken place, there should exist a so-called class-file, usually within the current directory:

fileName.class Name of a file containing Java byte-code, ready to run within a Java Runtime Environment (JRE).

This is the Java byte-code — to be found in files suffixed by `.class` — that can be run by the byte-code interpreter, the byte-code interpreter being an implementation of the Java Virtual Machine (JVM). The source-code is needed only for compiling the byte-code, but not for running it!

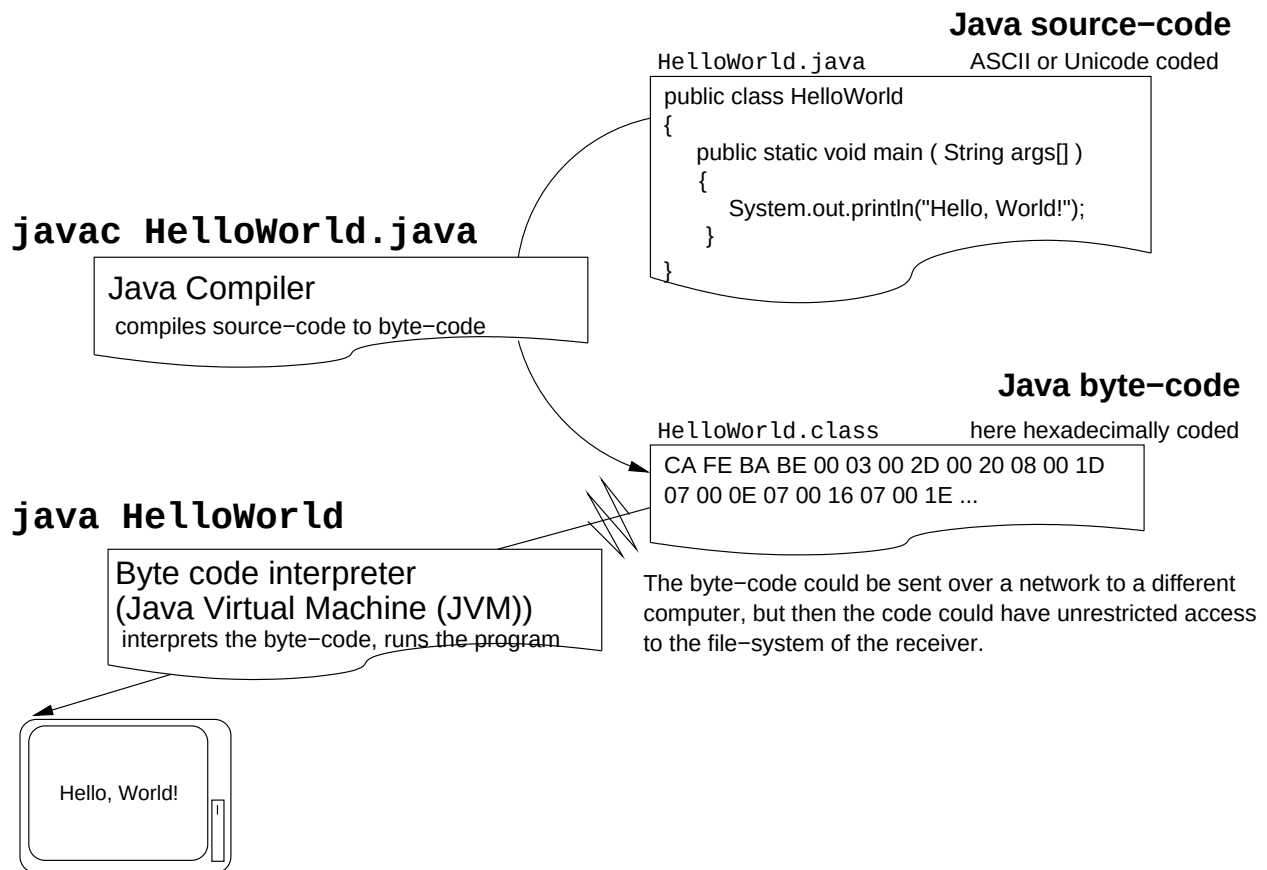
Byte-Code: Sequence of bytes, that codes the commands of a Java program to be run by an implementation of the Java Virtual Machine (JVM). Files of byte-code are produced by compiling Java source-code with the Java compiler. The byte-code is supposed to be fully portable, this means it should run the same in Java Runtime Environments (JREs) on different native platforms.

Running Programs/Applications

Running Programs/Applications — Regular Java programs, also called applications, can be run by issuing the command “`java [pathName] fileName transferString(s)”` in a terminal-window with a properly chosen present working-directory. This command starts a Java Runtime Environment (JRE) and triggers a call of the (compiled) `main()`-method. (Remember that the `main()`-method characterizes the source-code of an application.) (The *fileName* is used without any suffix!).

¹⁵Text-file means text written without any invisible control-code, except returns or tabulators.

The end-user can append strings (separated by spaces) to the command “**java [pathName] fileName**”, triggering a call of the `main( String args[] )`-method of the given program — with these strings as arguments. Thus, the user can transfer strings¹⁶ to the program, for more see page 72.



As can be seen in this example, the class-name (HelloWorld inside the text-file) and the file-name of the code (HelloWorld.java) to be compiled, have to be the same. Here in the source-code the keywords `public class` are followed by the program-name and the program-code, the actually programmed command: `System.out.println("Hello World!")`, can be found inside the so-called `main()`-method. The printed output “Hello, World!” generally appears in the terminal-window. CAUTION: All letters in commands or the program’s source-code are treated case-sensitive by operating-system or by the Java compiler.

Runtime System/operating-system	Standard output is where the information written into System.out goes:
Java Development Kit (JDK) /Unix, Linux	terminal-window
Metrowerks/Mac OS 9.x	Out-Window
Netscape Navigator/Mac OS 9.x	Navigator-Tools-Java Console
Internet Explorer/ Windows, Mac OS X	Log-file, terminal-window

The Java compiler also finds classes in directories of the system-variable `CLASSPATH`. The Java Runtime Environment (JRE), that is started by the “java”-command, searches for class-files that contain byte-code in the directory from which the “java”-command has been issued and then in the directories given by the `CLASSPATH`-variable.

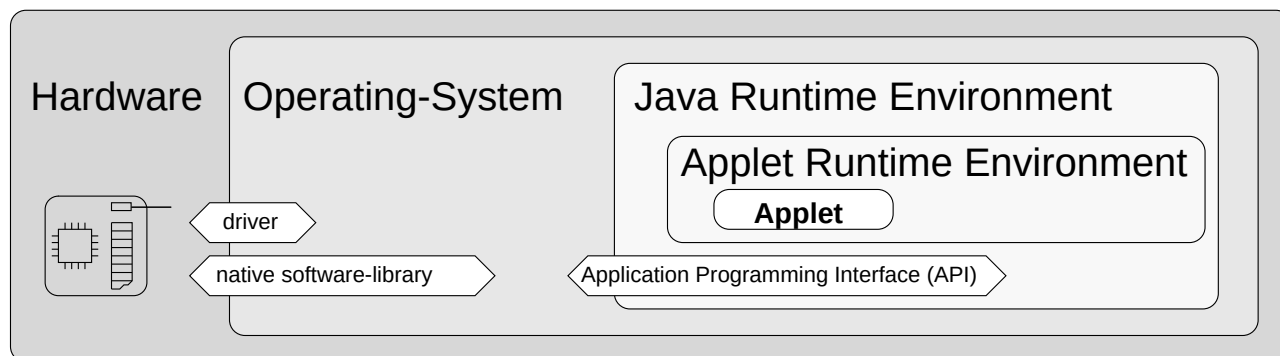
A programmer of Java-code may implement security measures, for example by establishing a security-manager, see page 462. Or, the user may restrict access to platform-resources by editing policy-text-files (page 458) and by activating options of the java-command (page 461). A way to run voluminous pieces of Java byte-code with the java-command is shown on page 326. The source-code of the “HelloWorld”-program also can be found on page 71.

¹⁶A string is a sequence of characters.

Running Applets

Running Applets — A key feature distinguishes applets from code loaded over a network by a client from a server¹⁷: With applets, security measures, like the accessibility of the file-system, are easily controllable or preset by the client! An applet is given by the byte-code compiled from a Java source-code that uses a special Applet-class (sometimes this kind of source-code is also called applet). The source-code of an applet is designed to run in a special “applet runtime-environment”. Such an applet runtime-environment (Internet-browser, appletviewer) *essentially* is just a security-restricted Java Runtime Environment (JRE).

Applet: An applet is a program having simple *externally controllable, by default, restricted access* to the resources of the platform it runs on. An applet is a Java program which has been specially (pre-)designed to work in an applet runtime-environment; this should make it exceptionally easy for an end-user to restrict the misuse of resources by the running applet.



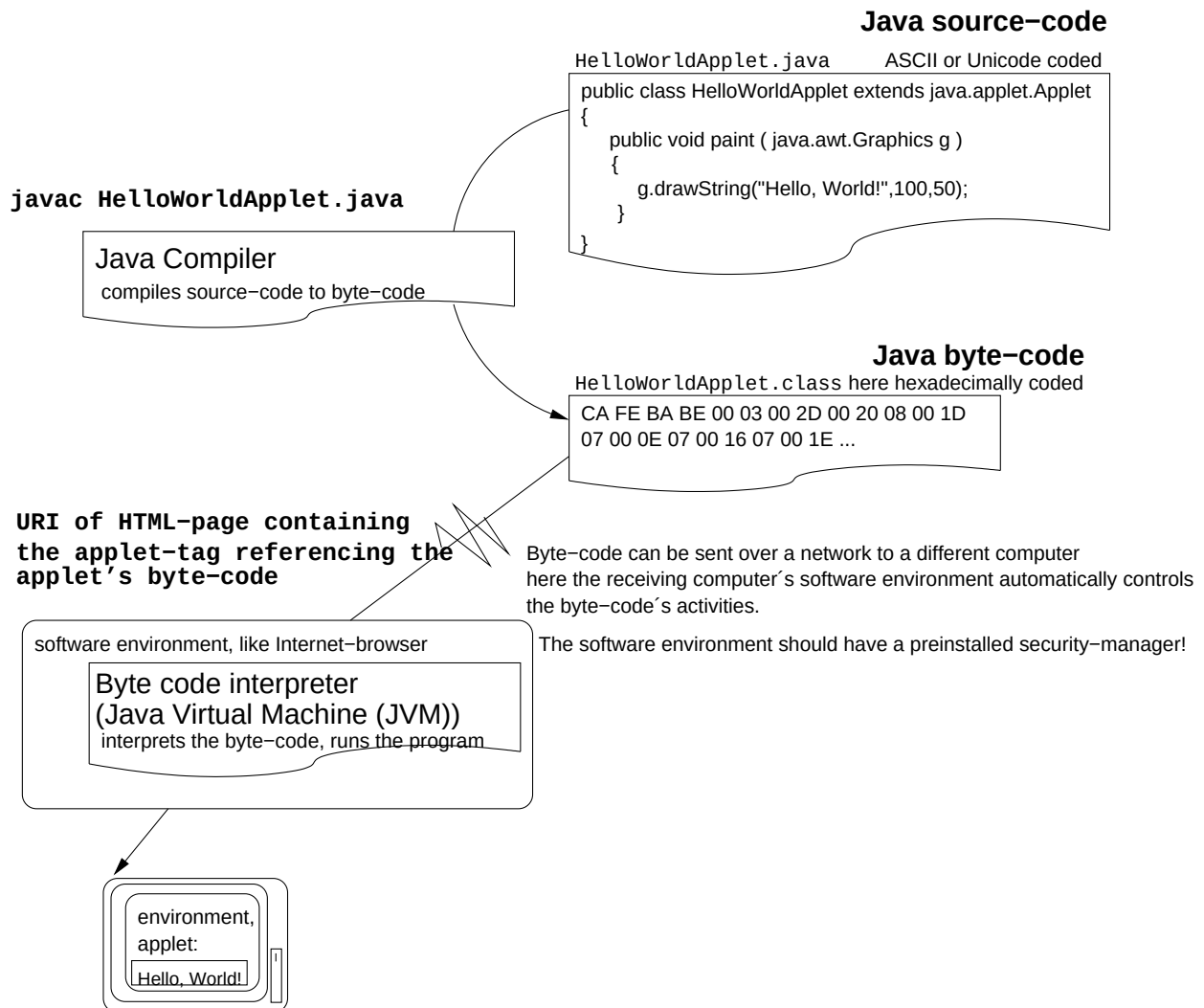
hardware + operating-system = native platform

hardware + operating-system + Java Runtime System = Java virtual platform

Applet runtime-environments are (Java-)programs which start a Java Runtime Environment (JRE) to run an applet AND almost always automatically provide additional input listening and extra security checks. The standard example of such programs on computer-platforms are Internet-browsers that are Java-enabled and therefore can run applets. Often only the virtual machine, the Abstract Window Toolkit (AWT) and some simpler Input/Output operations are implemented in these programs.

Apart from the different software-environment, applets remain autonomous programs. If the byte-code of an applet is to be downloaded over the Internet, then the applet is usually held small, to minimize waiting-time. What about sending byte-code of applications over the Internet? That's possible, but applets can be given easier security checks and need less additional code to handle in- and output routines. So applets furnish a built-in security and economize on code volume and can be incorporated into other software. That's their advantage over sending byte-code of applications right away over the Internet.

¹⁷The notions client and server are introduced beginning with page 275.



How can an applet be incorporated into a Java enabled HTML- or Internet-browser? By inserting an applet HTML-tag into the HTML-text:

```
<applet code = "http://www.abc.com/programs/theAppet.class "></applet>
```

How are applets started? By entering the Uniform Resource Identifier (URI) of the HTML-page into the command-line of a Java-enabled Internet-browser. As already mentioned, sending programs over the Internet to run them on a computer raises question of security! Properly designed applet runtime-environments, by default, deny any applet the access to the file-system and any applet has only limited access to the Internet, for example to bar applets from exporting data from your computer.

Applets are a special kind of Java program: Applets are run inside some applet runtime-environment (security-sandbox) within the Java Runtime Environment (JRE). Why? Before an applet is run, by default, the Java Runtime Environment has been restrained from granting freely access to system-resources (→ security-sandbox). This kind of security-environment, which automatically alleviates the user from any security fine-tuning is what basically distinguishes applets from any other (Java-)programs. Applets often are referenced by HTML-pages in Java-enabled Internet-browsers. Java Version 1.2 and higher make it possible for users to restrain Java → applications in a similar way. For one of the simpler examples see page 458. The name “**applet**” can be given to a variety of entities:

1. The **source-code** that contains a sequence of literals like “extends java.applet.Applet” or “extends javax.swing.JApplet”.
2. The **byte-code** compiled from such a piece of source-code.
3. A **file** containing one of the both items mentioned above.
4. The running piece of software and its **on-screen-representation**.

A related characterization of an applet is found beginning with page 397 in the context of networking. In the parts about security matters and about the single-user-interfacing more questions are answered in detail. More on the source-code (page 400) of the “HelloWorld”-applet can be found beginning with page 396. The basic functionalities of an applet are given by Java’s code-libraries that support the programming of a Graphical User Interface, for more see page 613. Around this page are also reiterated the distinctions between an application and an applet.

Part III.

Basic Programming I The Language and Utilities

Basic Programming I, The Language and Utilities — The term “Java” may be used to describe the entire virtual platform given by the Java Runtime Environment (JRE). Or “Java” may be called the entire Java Development Kit. But “Java” may also denote the Java programming-language. Basic elements of this programming-language are introduced in the following two parts of the text. Data-storage in variables and data-processing by expressions or successive program statements represents the core of most symbol-oriented programming-languages; especially in the case of the Java programming-language. In that context have been identified various strategies for structuring the source-code:

- Gathering functional entities of code in “(sub-)routines” or “methods”, to be called and – if necessary – recalled from different places in the program (Proceduralization).
- Constructing own extensible types of data-storage and -manipulation (Object-Orientation).
- Being able to run two pieces of code concurrently or parallel in time (Multithreading).
- Gathering multiple pieces of data (and code¹⁸) and to give them a common way of access (Aggregation).
- And finally storing code for reuse in a library (Packaging).

These concepts are to be illustrated by considering their implementation within the Java programming-language. At the end of this (large) part of the text, are introduced some Java code-libraries (pre-written Java packages).

Hello World Program

Hello World Program — A Java program is written as a text-file in a format given by a character encoding-scheme like ASCII, EBCDIC, ISO Latin or even Unicode-format (as introduced on page 34). Before compiling, the Java compiler translates the source-file into Unicode-format. Consider again the source-code of a typical HelloWorld-program: (The same program has already been introduced on page 37 with another name and slightly differing comments.) One-line comments follow the token “//”. Multiline comments are enclosed by the tokens “/*” and “*/”. For more on commenting see page 323.

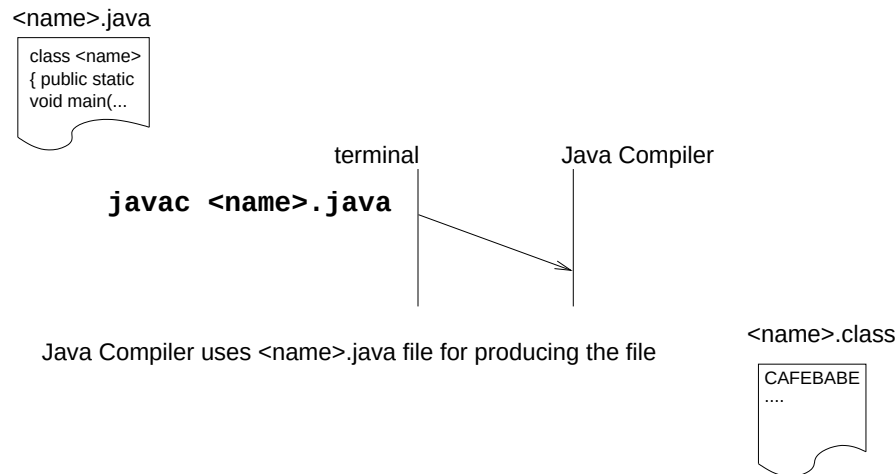
```
public class HelloWorld
{ public static void main( String unused[] )
  { System.out.println( "Hello World!" );
  }
}
/* Comments:
--- Try compilation and running without using the keyword "public".
--- Braces mark blocks of source-code.
--- The entire piece of code marked by the Java "class"-keyword
    is given the name HelloWorld.
--- The code to be run is found inside Java's main()-method,
    this is the code 'System.out.println(...);',
    that contains the string "HelloWorld!"
--- The modifier "public" tells that the main()-method
    is freely accessible.
--- The modifier "static" tells the Java compiler
    that the main()-method is associated with the class,
    and not with any object derived from this class.
--- The return type "void" tells the Java compiler that
    the main()-method does not return any value.
--- The main()-method is called automatically
    within the Java Runtime Environment (JRE) when the
    user issues the java-command below.
--- Space-separated strings, appended to the java-command,
    would appear as an array of name "unused".
--- Suggestion: Closing braces are positioned under opening braces,
    this ensures a better overview over block- and brace nesting.
*/
/*
marycosway@linux:~/JavaCodeFiles/Basics1/PrimitiveTypesMore > javac HelloWorld.java
marycosway@linux:~/JavaCodeFiles/Basics1/PrimitiveTypesMore > java HelloWorld
Hello World!
marycosway@linux:~/JavaCodeFiles/Basics1/PrimitiveTypesMore >
*/
```

¹⁸ Later: When gathering objects this means gathering the data of objects as well as gathering their methods.

As can be seen in the terminal printout above, the command “`javac HelloWorld.java`”, has to be issued in a terminal-window with a current directory containing the text-file with the source-code above. Then the Java compiler uses the source-code above (file-name: `HelloWorld.java`) to produce another file that contains the runnable byte-code (file-name: `HelloWorld.class`).

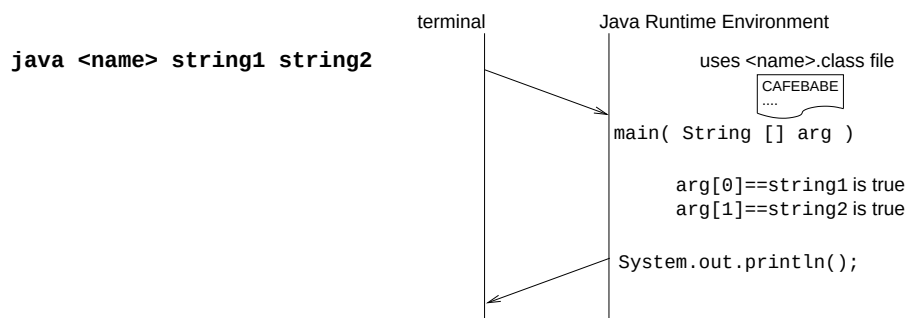
Java requires the name of the class that contains the `main()`-method to be the same as the name of the file that contains the source-code plus a suffix “`.java`”.

The Java compiler stores the file that contains the byte-code under the name of the class that contains the `main()`-method plus a suffix “`.class`”:



So the command “`javac HelloWorld.java`” produced the byte-code and stored it in a class-file. The command “`java HelloWorld`” establishes a Java Runtime Environment (JRE) and starts running this byte-code. The part of the source-code of the application that “does something” is given by the code inside the so-called `main()`-method. The user-initiated-event of entering the command “`java HelloWorld`” starts the Java Runtime Environment (JRE) AND invokes the (thus externally triggerable) `main()`-method. More on triggerable methods see page 127. Then, byte-code compiled from the line of source-code, “`System.out.println(“Hello World!”);`”, generates the output onto the screen¹⁹.

The `main()`-method has been declared with an expression “`String unused[]`” inside the parentheses (alternatively use “`String[] arg`” or the like). This indicates an expected transfer of a sequence of strings (an array) from the program-call into the program-code (Each of these strings must not contain a blank character.). Then, inside the program-code these strings are given as variables of type `String`. (For more about so-called arrays in Java see page 234 and especially for more on the `main()`-method see page 236.)



Java-applications run in a Java Runtime Environment (JRE) activated by the `java`-command. (Applets need an appletviewer-software or a Java enabled HTML-browser that incorporates or reuses a Java Runtime System (JRS). For applications that can also be used as applets see the remarks on page 619) For more on Java classes with a `main()`-method (Java programs) see page 141.

¹⁹CAUTION!: The short term `System.out.println()`, for printing out string-information, turns out as extraordinarily helpful, when exemplifying Java basics. But, as seen in the table following the graphic on page 66, the output triggered by this command may appear in different places on different platforms. This does violate “a little” the idea of portability of Java code. Rules that try defuse even such little violations, can be found under the name “100% Java” (<http://java.sun.com/products/archive/100percent/4.1.1/100PercentPureJavaCookbook-4.1.1.pdf>).

On the demonstration level of this text, such little violations shouldn’t bother beginners too much, later when developing code that has to be really robust under platform-changes it may be useful to try a little to adhere to the 100% Java rules.

Data-Storage — Data-Structures

Data-Storage — Data-Structures — Programming-languages generally relieve the programmer from scuttling tediously through individual bits or bytes. (Higher-level-) Programming-languages furnish variables, functioning as elementary storage-spaces for data, usually numbers or characters (the concept of a variable already has been introduced on page 40). When using higher-level programming-languages like Java, numbers and sequences of characters are simple elementary data-structures. Thus again, the programmer shuns coding or interpreting of individual bits or bytes. Most parts of that elementary symbol-coding (ASCII- or Unicode-format) happens fully automated and the Java programmer works only with texts and number-symbols.

Concept: Literal, Type, Name and Variable

Concept: Literal, Type, Name and Variable — Most programs manipulate symbols and characters and use pre-written support-programs to display the results on a screen or a printer. During their run, these programs usually can take commands via keyboard or mouse. Related data-storage works by using basic concepts in symbol-oriented programming:

Data in Form of Literals — What is stored in the computer memory?

The data in form of literals usually can be read by humans, like a string “A string of characters” or a number “123”. These literals are usually to be found in the source-code of a program. Putting literals into the source-code represents one of the simplest means of importing actual data into a computing-environment, apart from reading input-streams (see page 297) or receiving messages (see page 575).

Type or Class of Literals — How is it stored into the computer memory?

Remember the introduction of how bytes can be used to code data (on page 33); an extension of this idea is given in programming-languages when introducing the notion of a class or a type: The **type or class** of given data tells (**DATA TO NUMBERS**) what CODING-MECHANISM IS APPLIED TO TRANSFORM THE DATA INTO SEQUENCES OF BYTES AND WHAT DECODING-MECHANISM IS TO BE APPLIED TO THE RAW BYTES TO TRANSFORM THEM BACK INTO LITERALS. For example, the character-literal “a” can be ASCII-coded by the binary number $61_{16} = 97_{10} = 01100001_2$, the sequence of zeros and ones represents the raw byte, stored as different electrical potentials in the computer’s Random Access Memory. For coding the character-literal “a”, the Unicode-scheme uses *two* bytes with the first byte being a null-byte and the second represented by the ASCII-code. As another example consider the number-literal “123”, the number “hundred-and-twenty-three” is coded by the Java-type byte by simply using the associated binary representation of the number $123_{10} = 01111011_2$. Variables of type “byte” can be added, subtracted, multiplied and divided with a rest. Characters may be concatenated into Strings. These operations on the coded data demonstrate the second aspect of a type or class of given data: (**DATA-MANIPULATION**) The type or class of given data includes MEANS FOR DATA-SPECIFIC TRANSFORMATION also called “semantic manipulation”.

type, class

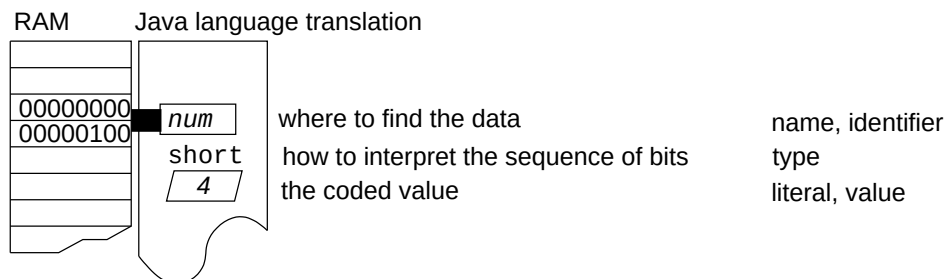
- Type: 1. How given information is coded into bit-sequences (encoding) *and* what means are available for manipulating that information (available semantic transformation).
2. (Java) A concrete class, a concrete interface, an array-type or a primitive-type.

Identifier, Name — Where is it stored in the computer memory?

The data is given by a sequence of bytes stored in the computer’s Random Access Memory (RAM). 
The programming-language uses a name to identify the RAM-address of the data stored and to get the

type-information. The name of such a piece of data usually stands for the address in which the first byte of the data has been stored. In RAM-addresses, the end of the data often is indicated by a null-byte. — For the programmer, the (variable)-name (or data-identifier) gives read- or write-access to the data. In the RAM the data of a given type usually waits for further manipulation, like displaying or printing (see graphic on page 35). Using identifiers, makes it unnecessary to reintroduce the data explicitly or “literally” each time the data is needed. — The following anticipates the data-coding-model of an integer-value “4” of type “short” which is coded by using two bytes:

Java variable of primitive-type



Variable = Type + Data-Literal + Name

A variable is given by the data together with its type-information and a name (variable-name or (variable-)identifier). That means, variables are given by the raw data-bytes together with the information of how to manipulate and how to retrieve the information they contain.

OUTLOOK: The range of validity of a variable’s name is also called scope, see page 122. An object, as introduced on page 131, represents a slight generalization and variation: A variable’s type can be seen specifically as an object’s class. A variable’s data-literal becomes generalized to the state of an object. Finally, the name of a variable, a part of the variable’s identity, becomes the identity-tag of an object, see page 161 (NOT the name of the object!). In Java, one object can be referred to by using several names, each of them reassignable into pointing to another object, see the graphic around page 143.

Variable = Reference, pointer or alias of an object of reference-type OR the object itself!

In Java both definitions are used, the first one for variables of primitive-type, the second one for variables of reference-type. Java-objects can be seen as collections of variables and methods for manipulating these variables.

Some special kinds of variables in Java are local variables (page 121), class- and instance-variables (page 148) and anonymous variables (page 159). End of outlook.

Concept: Declaration, Allocation, Instantiation of a Variable

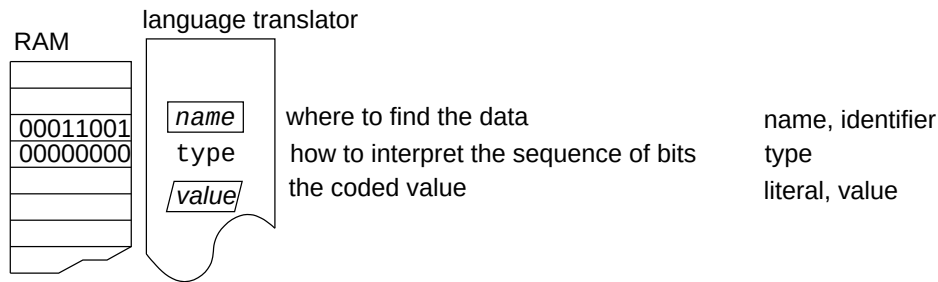
Concept: Declaration, Allocation, Instantiation of a Variable — Making a variable or object existent in a high-level programming-language requires at least three processes:

(Declaration) Introducing a variable-name (= identifier) for data is called declaration. Sometimes the type of the variable or object is defined too.

(Allocation) Then the computer reserves or allocates space in its Random Access Memory (RAM). Usually the type of data is now known to the high-level language. Sometimes the declaration means just to reserve a name and the allocation needs a separate command, like the “new” command in Java or C++/C.

(Instantiation) “Filling a variable or object” with data is called assignment of a variable or instantiation of an object.

Variable and programming language



In some cases one single expression can comprise of all three processes in one: Declaring, allocating and instantiating; as for example in the expression `short num = 4;`.

Other cases add a fourth process; an entire definition of the used type! See the definition of the type “Pair” in the second code-sample below.

Concept: Statically-, Dynamically-, Runtime-Typed Programming-Languages

Concept: Statically-, Dynamically-, Runtime-Typed Programming-Languages — Information is stored in a computer coded into sequences of bits or into sequences of bytes. Programming-languages make the data accessible by assigning a name to the byte-blocks in question. The format of storage and the associated methods for the manipulation of the information encoded by the data has been called the type of the data. Name, type and data are the defining notions of a **variable in a programming-language**.

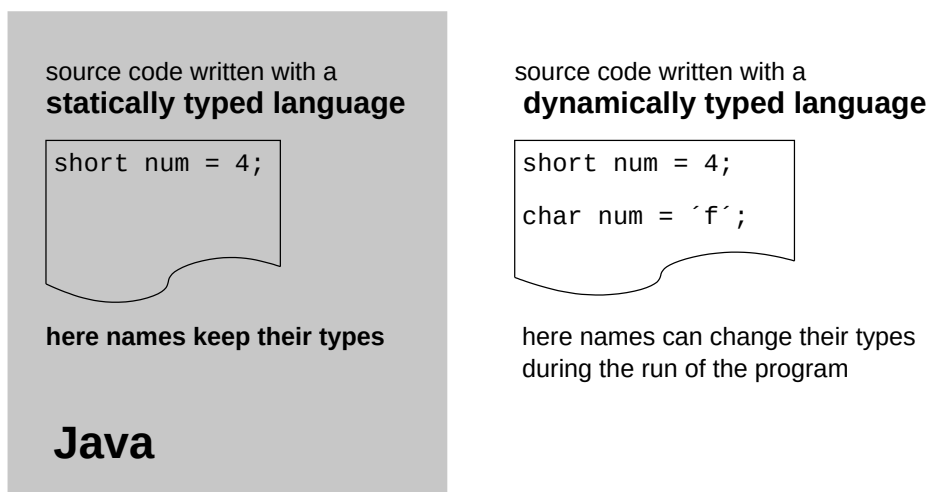
variable =
name + type +
data

In Java, name and type of a variable are fixed on declaration and are never changed during the run-time. This characterizes **STATICALLY-TYPED LANGUAGES**, like Java or C/C++.

In **DYNAMICALLY-TYPED LANGUAGES** things may evolve differently: There, variables, once declared, can change their types consecutively and thus contain data of different types during their existence. That’s the case in high-level languages like Smalltalk, Lisp or Mathematica.

But Java has other dynamic aspects concerning types: Loading classes on runtime means that Java can extend the set of known types during a program-run. This describes one aspect of Java’s late-binding capacities. C/C++ cannot do that.

A **RUNTIME-TYPED LANGUAGE** performs type-checks on runtime. Java checks type-casting at runtime, C/C++ does not.



Java: Declaration, Allocation, Instantiation: Examples and Outlook

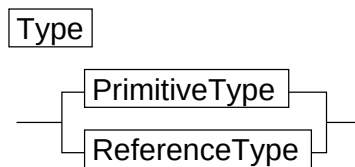
Java: Declaration, Allocation, Instantiation: Examples and Outlook — The formulation “Allocating space in a computer’s Random Access Memory” sometimes is used interchangeably with “creation of a variable” in a high-level language. As already indicated, with simple types, the processes of declaration, allocation

and instantiation can take place in one line of source-code (`int i=1;`). A program's source-code usually uses a declaration-part, where the variables are introduced, and another part of commands, where the contents of the variables are manipulated:

```
class VariableOfPrimitiveType
{ public static void main( String unused[] )
  { int i; // declaration
    i=1;   // instantiation
  };
}
```

For more see the chapter on variables of primitive-types beginning on page 83.

Remember the idea of a type (page 73) as a coding transformation (data to/from bytes) with additional means for data-manipulation (data to other kind of data, for example by incrementing an integer). Java furnishes two kinds of types:



Type := *PrimitiveType* | *ReferenceType* — *PrimitiveType* 84, *ReferenceType* 143

Programmer-defined types which are given by variables of reference-type are introduced here, but treated extensively later; the next paragraphs just give an example of the definition, declaration, allocation and instantiation of an object of reference-type. But first a note on the graphic above:

railroad-diagrams
or BNF-notation

Java language-constructs can be roughly described in **Backus-Naur form (BNF)** OR using **railroad-diagrams (syntax-diagrams)**, see the Appendix (page 655) for an exposition of the rules. At times, this structuralization can prove to be a GUIDING LINE when experimenting with own expressions. Railroad-diagrams or the BNF-notation of syntax-constructs just give an equivalent to a grammar-book of the programming-language. Knowing only *some* basic grammatic rules of a language often suffices to use the language well.

Furthermore, there are many constructs that are possible according to the syntax-structure, yielding unintelligible pieces of source-code; this stresses the character of the syntax-diagrams as NECESSARY RULES FOR CONSTRUCTING JAVA SOURCE-CODE. But the syntax-diagrams alone contain insufficient, rather very little, information for constructing valid Java source-code. The wealth of the sensible constructs is NOT treated in this text, but in the ➡☐ ➡☐ Java Language Specification 2nd Edition. In this text only the more common and simpler constructs are covered. Due to this classifying approach of the syntactical structures, many of the denotations are anticipated in the syntax-diagrams, this shouldn't deter the reader from following the systematically developed parts of the text.

In Java, programmer-composed types are always of reference-type. Such programmer-composed types require additional pieces of source-code which contain the actual type-definition. In the following piece of Java source-code four steps are fully discernable, namely: Type-definition, declaration, allocation and finally instantiation.

```
class VariableOfReferenceType
{ public static void main( String unused [] )
  {
    // type-definition below:
    class Pair
    { int x1, x2; Pair(){}; Pair(int x,int y){x1=x; x2=y;}; };
    // declaration (name)
    Pair coord;
    // allocation (new-keyword reserves space in Random Access Memory)
    // instantiation (allocation plus filling variables with data-literals)
    coord = new Pair();
    coord.x1 = 6; coord.x2 = 8;
  }
}
```


But not every variable of reference-type in Java needs that much coding. As a singular exception, `String`-variables are of reference-type and can be introduced as easily as variables of primitive-type (`String s = "Hello World!"`; see also page 145). Allocation for primitive-types happens automatically. Allocation-expressions, concerning primarily reference-types, are treated later.

A typical instantiation of a variable of primitive- and of reference-type face each other in the code fragment below:

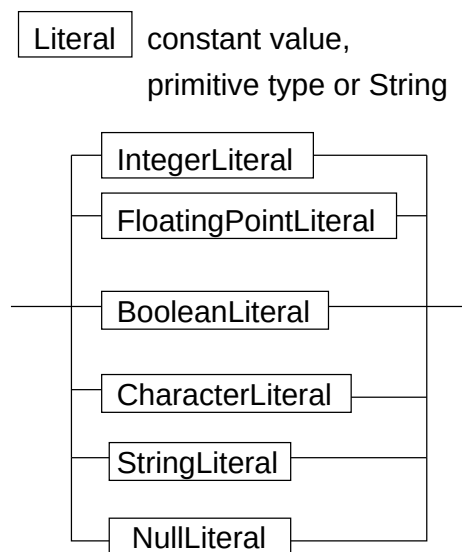
```
// declaration
PrimitiveType nameP;
ReferenceType nameR;
// instantiation of variable of primitive-type:
nameP = value;
// instantiation of variable of reference-type (an object):
nameR = new ReferenceType( value );
```

Declaration, allocation and instantiation of a variable of reference-type in Java has been treated extensively beginning with page 143.

Java: Literals

Java: Literals — The denotation of the primitive-types (`int`, `float`, `double`, `boolean`, `char`) is anticipated here. The following may be used as a mere illustrative introduction, of which the reader may skip uninteresting parts. But later, the sections below may serve as a quick reference.

value literal, value Java literals are sequences of Unicode-characters that represent a value of a primitive data-type or the content of a `String`-object. The following classifies the literals which are possible in Java²⁰:



Literal := *IntegerLiteral* | *FloatingPointLiteral* | *BooleanLiteral* | *CharacterLiteral* | *StringLiteral* | *NullLiteral* — *IntegerLiteral* 78, *FloatingPointLiteral* 79, *BooleanLiteral* 79, *CharacterLiteral* 80, *StringLiteral* 81, *NullLiteral* 145

The *NullLiteral* is given by the string “null” and introduced with reference-types on page 145. In the following sections, signs (+/-) of numbers are taken to act as unary operators.

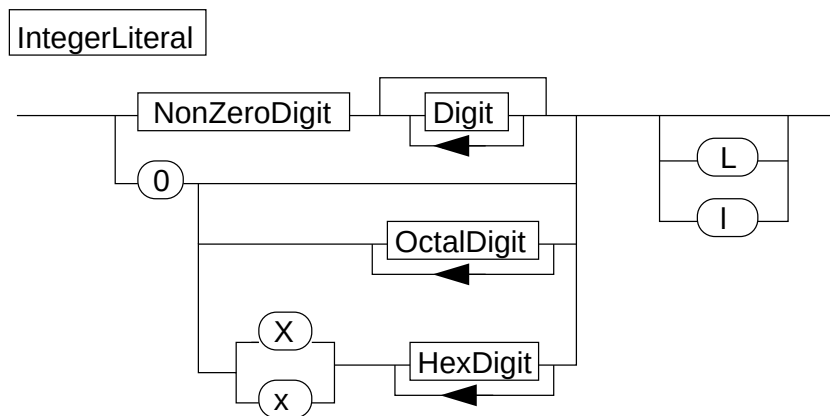
Integer-Literals

value literal, value Integer-literals contain programmer-readable integer-numbers of different types (for example `int` or `long`) and in some cases the numbers are given in non-decimal number-systems (hexadecimal or octal):

²⁰The Java-syntax, the prescription of how the Java source-code can be written, has been specified here by Railroad Diagrams (to be read from left to right) and independently by the Backus-Naur Form (BNF). If they are read in conjunction, these formulations should appear mostly self-explaining, otherwise reference page 655.

Default is type int (32 bit or 4 bytes)
 23233 type int decimal integer-literal
 45654L or
 45654l type **long** decimal integer-literal
 (better use only L to avoid confusing l (el) and 1(one))
 0123 type int **octal** integer-literal
 (leading zero)
 0x1ea or
 0X1ea type int **hexadecimal** integer-literal
 (leading 0x, with a=10,b=11,c=12,d=13,e=14,f=15)

Octal numbers are given to the number-base eight, hexadecimal numbers are given to the number-base 16 and decimal numbers are given to the number-base 10 (For more on number-bases see page 34); for more on types of integer-literals see page 85.



IntegerLiteral := *DecimalIntegerLiteral* | *HexIntegerLiteral* | *OctalIntegerLiteral*

DecimalIntegerLiteral := *DecimalNumeral* [*IntegerTypeSuffix*]

HexIntegerLiteral := *HexNumeral* [*IntegerTypeSuffix*]

OctalIntegerLiteral := *OctalNumeral* [*IntegerTypeSuffix*]

IntegerTypeSuffix := l | L (the character (el) in its lower-case version 'l' or its upper-case version 'L')

DecimalNumeral := 0 | (*NonZeroDigit* { *Digit* })

Digit := 0 | *NonZeroDigit*

NonZeroDigit := 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

HexNumeral := 0x <*HexDigit*> | 0X <*HexDigit*>

HexDigit := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | a | B | b | C | c | D | d | E | e | F | f

OctalNumeral := 0 <*OctalDigit*>

OctalDigit := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Floating-Point-Literals

value literal, value Floating-Point-Literals — Numbers which include fractions of their unit (one fourth of some unit can be written like 0.25) are called floating-point numbers:

Default is type double.

12. type double floating-point-literal

12.3 type double

.3f type float

Postfixes F or f indicate a type float (32 bit or 4 byte).

Postfixes D or d indicate a type double (64 bit or 8 byte).

1.2E3 or

12e2 type double floating-point-literal

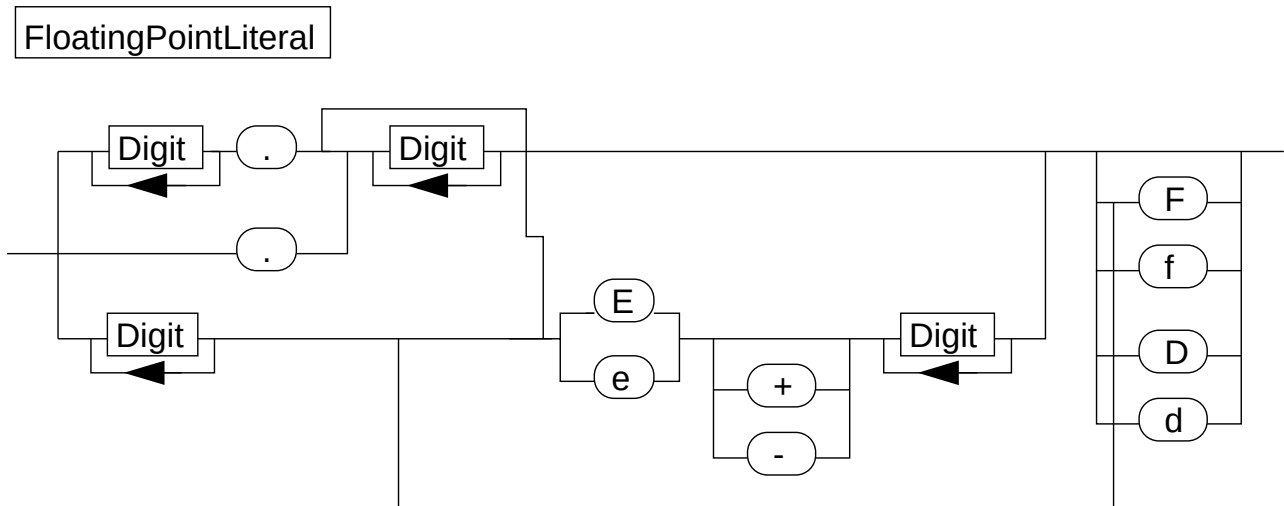
12e-3 type double

Postfixes with a letter e or E indicate an exponential-notation.

A decimal-point (a period (.)) or a letter (e, E) immediately mark a number as a floating-point-literal (of primitive-type double)! A letter (e, E) stands for “exponent of ten” and indicates the number of digits to shift the decimal-point:

$$1.2E3 = 1 \overbrace{200}^3 .0 \quad 12E2 = 12 \overbrace{00}^2 .0 \quad 12E-3 = 0.\underbrace{012}_{-3}$$

Literals of floating-point numbers are always given in the decimal number-system. For more on types of floating-point-literals see page 86.



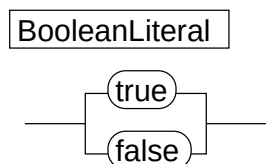
```

FloatingPointLiteral :=
<Digit> . { Digits } [ ExponentPart ] [ FloatTypeSuffix ] |
. <Digit> [ ExponentPart ] [ FloatTypeSuffix ] |
<Digit> ExponentPart [ FloatTypeSuffix ] |
<Digit> [ ExponentPart ] FloatTypeSuffix
ExponentPart := ( e | E ) SignedInteger
SignedInteger := [ ( + | - ) ] <Digit>
FloatTypeSuffix := F | f | D | d

```

Boolean-Literals

value literal, value Boolean variables store the information equivalent to one bit. The two possible values of boolean variables are represented in Java by two keywords: **true** and **false**.


$$\textit{BooleanLiteral} := \text{true} \mid \text{false}$$

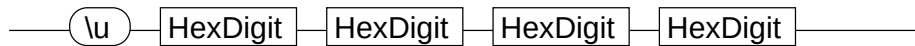
For more on boolean values see page 84.

Character-Literals

`<value>` literal, value Character-Literals — Remember that the source-code of a Java program can be found in a text-file of a character-encoding-scheme like ASCII, EBCDIC, ISO8859-1 (Latin-1) or even Unicode-format (page 34). Before compiling, the Java compiler translates the text-file of source-code into Unicode-format. This allows to work with Unicode-characters even in text-files of source-code that use a non-Unicode encoding-scheme. In non-Unicode text-files, Unicode-characters can be coded by the following

so-called escape-sequence `\u0000` - `\uffff` (the u stands for Unicode, the hexadecimal number codes two bytes).

UnicodeEscape



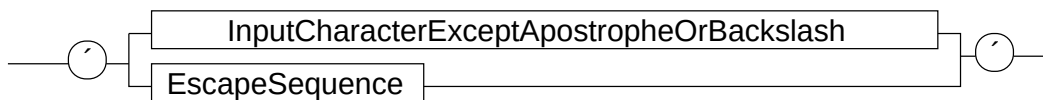
UnicodeEscape := `\u` *HexDigit* *HexDigit* *HexDigit* *HexDigit* — HexDigit 78

A character-literal can be encoded by a pair of bytes. These two bytes code a single letter using the Unicode-format (encoding-scheme presented at <http://www.unicode.org/charts> or see page 34). In Java, such literals are marked by enclosing apostrophes:

```

'a'          // the character-literal for the letter "a"
'\u03C9'
// above, the character-literal for the Greek letter omega: "ω"
  
```

CharacterLiteral



CharacterLiteral := `'` *SingleCharacter* | *EscapeSequence* `'`

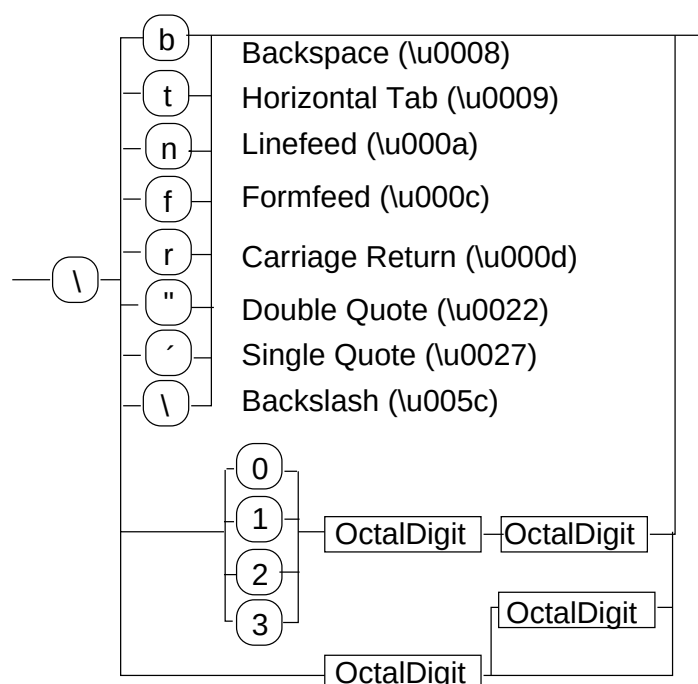
SingleCharacter := *InputCharacter* EXCEPT APOSTROPHE (`'` `\u0027`) OR BACKSLASH (`\` `\u005C`)

InputCharacter := *UnicodeInputCharacter* EXCEPT CARRIAGE RETURN (CR `\u000d`) OR LINE FEED (LF `\u000a`)

UnicodeInputCharacter := *UnicodeEscape* | ANYUNICODECHARACTER

In Java, escape sequences allow to write down non-printable control characters: Escape sequences are `\b` (backspace) `\t` (tab) `\n` (newline) `\f` (form feed) `\r` (return carriage) `\"` `\'` `\\` and UnicodeLetters coded with octal numbers within the hexadecimal range: `\u0000` - `\u00ff`. The compiler sees only the associated Unicode-character.

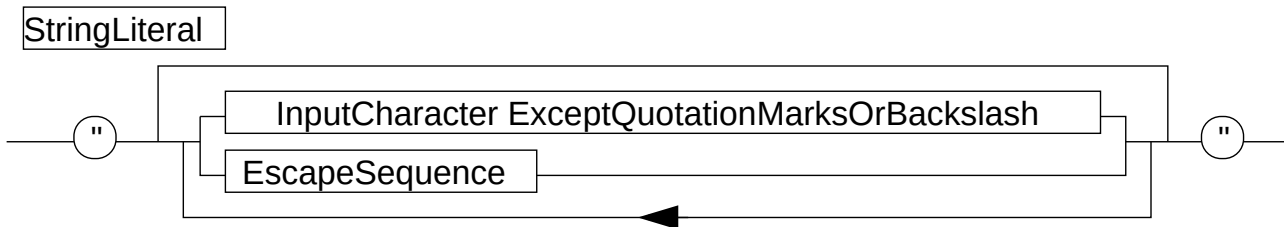
EscapeSequence



String-Literals

value literal, value A string-literal is a value that represents a sequence of letters.

```
"Hello World!"
"A \"string-literal\" with Escapes! \n"
```



StringLiteral := " { *StringCharacter* } "

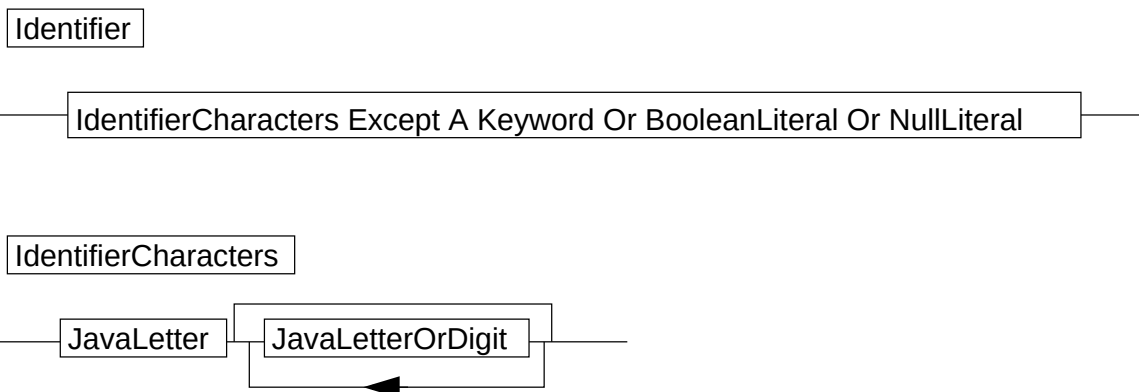
StringCharacter := *EscapeSequence* | *InputCharacter* EXCEPT QUOTATIONMARK (" \u0022) OR BACKSLASH (\ \u005C)

In Java, a string-literal is automatically treated as a *String*-object, there exists no primitive string-type in Java! And ONCE INSTANTIATED *String*-OBJECTS THEMSELVES CANNOT BE CHANGED OR MANIPULATED (like being trimmed or extended by concatenation)!

String-literals only can take the length of a single line in a text-file²¹, longer *String*-objects have to be constructed by concatenating *String*-literals to **new** *String*-objects. For more on *String*-objects see 257.

Java: Identifier, Name

name name, identifier Java: Identifier, Name — Variables²² usually²³ have to be given a name to make their values (or functions) referencable in their future. Names or identifiers basically are SEQUENCES OF UNICODE-CHARACTERS of ARBITRARY LENGTH. Being composed of Unicode-characters makes names or identifiers CASE-SENSITIVE. A name or identifier cannot be a JAVA KEYWORD, like *if* or *while*. case-sensitive names



Identifier := *IdentifierCharacters* BUT NOT A *Keyword* OR *BooleanLiteral* OR *NullLiteral*

Keyword := ONE OF THE FOLLOWING: *abstract*, *boolean*, *break*, *byte*, *case*, *catch*, *char*, *class*, *const*, *continue*, *default*, *do*, *double*, *else*, *extends*, *final*, *finally*, *float*, *for*, *goto*, *if*, *implements*, *import*, *instanceof*, *int*, *interface*, *long*, *native*, *new*, *package*, *private*, *protected*, *public*, *return*, *short*, *static*, *strictfp*, *super*, *switch*, *synchronized*, *this*, *throw*, *throws*, *transient*, *try*, *void*, *volatile*, *while*. (*false* and *true* are boolean-literals, *null* is called the null-literal)

IdentifierCharacters := *JAVALETTER* { *JAVALETTERORDIGIT* } — *BooleanLiteral* 79, *NullLiteral* 145

The method

²¹This can be seen from the definition above, because *INPUTCHARACTERS* are *UNICODECHARACTERS* without line-feed and carriage-return.

²²And later names of methods, classes and interfaces.

²³On page 71, the *String*-literal in the “HelloWorld”-program can be seen as a variable without a given name. Later (page 159) are introduced anonymous classes and -objects.

```
java.lang.Character.isJavaIdentifierStart( UnicodeNumber )
```

JAVALETTER,
JAVALET-
TERORDIGIT

returns true for a *UnicodeNumber* of a character that represents a JAVALETTER. A character being a JAVALETTERORDIGIT has a *UnicodeNumber* that makes the method

```
java.lang.Character.isJavaIdentifierPart( UnicodeNumber )
```

return the boolean-literal true. (The prefix “java.lang.” indicates the use of a software-library furnished together with the Java language²⁴.)

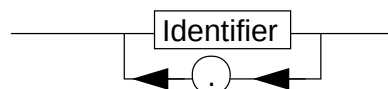
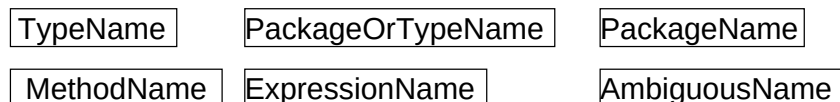
```
class IdentifierLetter
{ public static void main( String unused[] )
  { System.out.println( "JAVALETTER " + Character.isJavaIdentifierStart( '\u0041' ) );
    System.out.println( "JAVALETTERORDIGIT " + Character.isJavaIdentifierPart( 'b' ) );
  } }
/*
C:\WINDOWS\JavaCodeFiles\Basics1\PrimitiveTypesMore>javac IdentifierLetter.java

C:\WINDOWS\JavaCodeFiles\Basics1\PrimitiveTypesMore>java IdentifierLetter
JAVALETTER true
JAVALETTERORDIGIT true

C:\WINDOWS\JavaCodeFiles\Basics1\PrimitiveTypesMore>
*/
```

Be CAUTIONed of such traps as a lower-case Latin ‘a’ (\u0061) which has a different Unicode-number than the lower-case Cyrillic ‘a’ (\u0430), *both* may be represented by the same glyph! JAVALETTERS include the underscore (_) or the dollar sign (\$); within names and identifiers these characters are usually used by the Java system.

JAVALETTERS are the basis of any identifiers, which in turn form many kinds of names: *TypeName*, *PackageOrTypeName*, *PackageName*, *ExpressionName*, *MethodName* and the helper notion of an *AMBIGUOUSNAME*.



TypeName := *Identifier* | *PackageOrTypeName* . *Identifier*

ExpressionName := *Identifier* | *AmbiguousName* . *Identifier* (used in ARRAYACCESS on page 235)

MethodName := *Identifier* | *AmbiguousName* . *Identifier*

PackageOrTypeName := *Identifier* | *PackageOrTypeName* . *Identifier*

PackageName := *Identifier* | *PackageName* . *Identifier* (see also page 241)

AmbiguousName := *Identifier* | *AmbiguousName* . *Identifier*

Why discern these kind of names? Because there may be given a package, a type and a method with the same identifier. But this name has different meanings in different contexts of the source-code. (Such a duplication of a name should rather be avoided, to minimize confusion when (re-)reading source-code.)

²⁴When considering prefixes that end with a period (.), then the part that does not include the ending-period is also called qualifier: So, speaking of the prefix “java.lang.” has the same meaning as speaking of the qualifier “java.lang”!

Naming Conventions

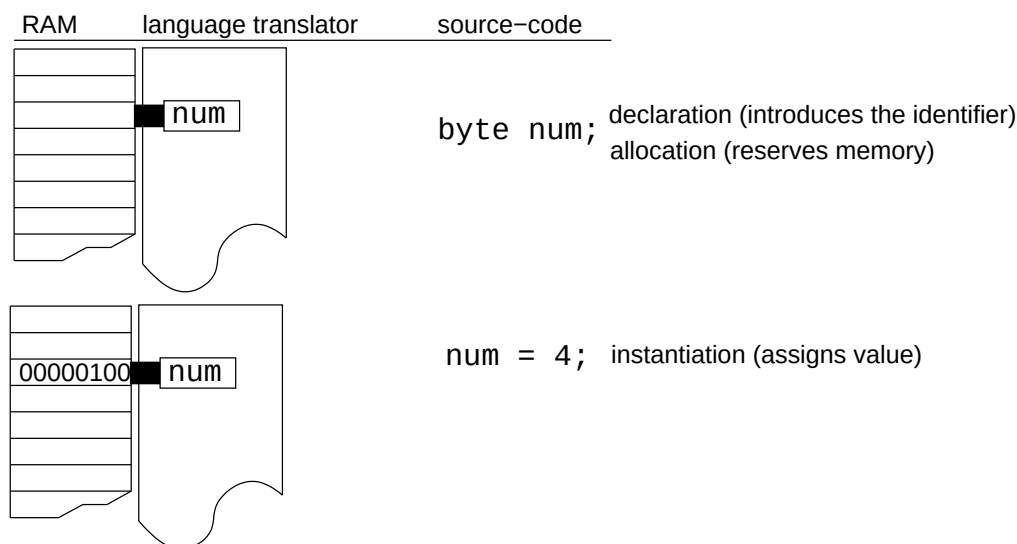
anIdentifierExample, _instance	First a lowercase letter, single words are discerned using an uppercase letter. Seldom instance-variables begin with an underscore.	variables, references to objects, names of methods
Pair, PairOfIntegers	First letters are uppercase. Single words are discerned using an uppercase letter.	classes or interfaces
MAX_VALUE	Names are written fully in uppercase letters and words are separated by underscore characters (_).	constants (final variables) See also page 182.

The dollar sign (\$) usually is used only in compiler-generated names.

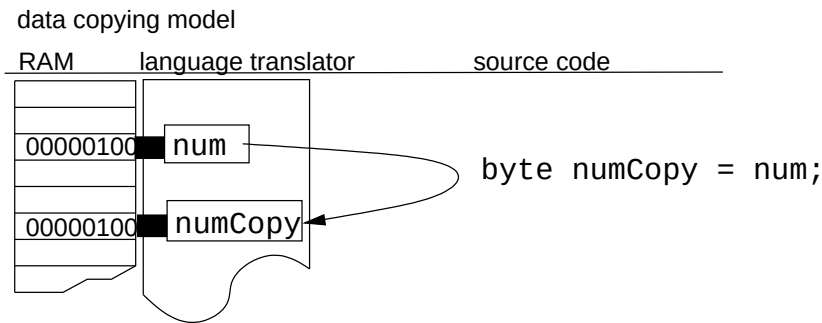
Java: Primitive-Types

type coding/decoding AND semantic manipulation Java: Primitive-Types — The declaration of primitive-types automatically reserves a part of the memory for data of that type. Java automatically assigns this part of the computer's Random Access Memory to the variable-name introduced in the declaration. Literals give the pieces of data, that can be put into the reserved part of the memory. Putting data into the variable's memory for the first time is called instantiation of the variable of primitive-type.

Variable of primitive-type:

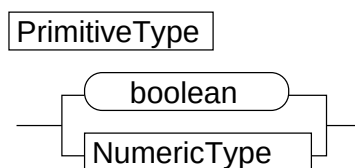


Variables of primitive-type have DEFAULT VALUES. Nevertheless when declared, variables of primitive-type can BE ASSIGNED A VALUE, a literal (num=4;) or the value of a pre-existent variable (num=numpreex;) ^{assignment by-value} Copying a piece of given data may be seen as a simple form of data-manipulation. This means to repeat the process of declaration, allocation and instantiation, whereby the instantiation is done by using the previously defined identifier. In the case of variables of primitive-type assignment and copying are the same:



(Copying may also be called cloning²⁵.) Remember the examples of declaration, allocation, and instantiation following page 75. A general form of declaring a variable of primitive-type is given below:

```
PrimitiveType name;
// declaration of variable of primitive-type:
// means instance-creation without assignment of value.
```



PrimitiveType := *boolean* | *NumericType* — *NumericType* 84

Logical Type

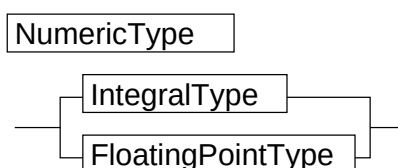
type coding/decoding AND semantic manipulation Logical Type — A variable of boolean type has two states, Java denotes these two states by the keywords *true* and *false*. Boolean values become arguments in branching statements like “*if(ok) then do this else do that*” (Branching statements are introduced beginning with pages 89 and 105.) Variables of boolean type are also used to solve problems involving so-called boolean operators like AND (in Java written *&* or *&&*), OR (in Java written *|* or *||*), XOR (exclusive OR, in Java written *^*; for more on the Java logical operators see pages 97 and 98)

```
// range: true, false
// default: false
// Codes one bit in the form 'true' or 'false'

// 'ok' is the variable-name, 'true' the literal of the value.
boolean ok = true;
int number = ok ? 1:0 // Converts 'true' to '1' and 'false' to '0',
// see page 99.
```

Numeric Type

type coding/decoding AND semantic manipulation A numeric type interprets given bytes as a number or as a single character (*char*); there are integral types in which the numbers can be manipulated without rounding errors and there are floating-point types with operations like addition or multiplication of limited precision.



NumericType := *IntegralType* | *FloatingPointType* — see below

²⁵A copying-process usually is called cloning when done with variables of reference-type; see page 144.

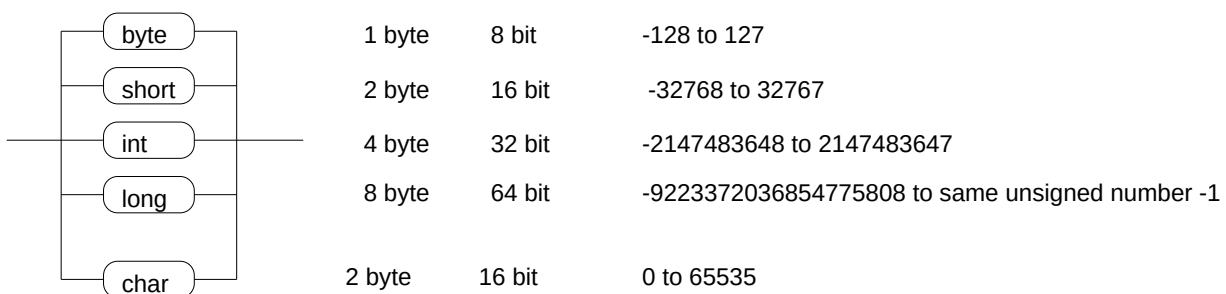
For each `IntegralType` exists a class extending the type `Number`, also called object-wrapper, furnishing constants `MIN_VALUE` and a constant `MAX_VALUE` which can be used for overflow control (see also page 257 on the class-type `Number`).

The floating-point types in Java are implemented according to the specifications of the Institute of Electrical and Electronics Engineers, "IEEE 754". This denotes an IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985 (IEEE, New York). The specification can be found under <http://www.ieee.org> or <http://standards.ieee.org> but unfortunately, at the time of this writing, the IEEE wants to be paid for sending a 100k PDF-file. (Possibly the specification exists somewhere else downloadable for free.)

Integral Types

type coding/decoding AND semantic manipulation There are five types for storing integer-values of differing ranges. A larger range costs more computer memory, considering this may become crucial when working with thousands of integer variables.

IntegralType



IntegralType := byte | short | int | long | char

```
byte numberByte = 128-1;
// range: two decimals, -128 ... 127
// default: 0
// signed integers, coded by one byte (8 bits)

char numberChar = 60000;
// range: four decimals, 0 ... 65535
// default: 0 or \u0000
// unsigned numbers, coded by two bytes (16 bits)
char c = 'a';
// or characters in Unicode-format

short number2Bytes = 30000;
// range: four decimals, -32768 ... 32767
// default: 0
// signed integers, coded by two bytes (16 bits)

int number4Bytes = 2000000000;
// range: nine decimals (10^9), -2147483648 ... 2147483647
// default: 0
// signed integers, coded by 4 bytes (32 bits)

long number8Bytes = 9000000000000000000L;
// range: 18 decimals, a 9*10^18 range (18 zeros)
// range: -9223372036854775808 ... 9223372036854775807
// default: 0L
// postfix "L" indicates a long-literal
// signed integers, coded by 8 bytes (64 bits)

number4Bytes = 1/0;
// produces an ArithmeticException during compile-time
```

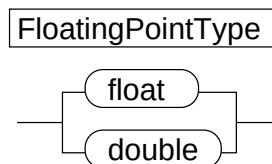
Java issues NO OVERFLOW/UNDERFLOW WARNING with *integers*. Only DIVISION BY ZERO TRIGGERS AN ARITHMETIC EXCEPTION message during compile-time or during runtime (for more on exceptions see page 280). The DEFAULT TYPE OF NUMBER-LITERALS IS “int”. Thus in some cases above, literals have been implicitly type-casted (int to short, int to byte). (For more on type-casting see page 88)

The ASCII-coding uses one byte to code the Latin character-set; this coding happens to be a special case of the Unicode-format which codes the character-sets of most languages with two bytes (making it possible to code at most 65536 characters). Variables of primitive-type char can have values represented by the coded characters: Expressions like 'A' + 'B' yield a number and char-variables can be compared ('A' < 'B') which proves useful for sorting.

OUTLOOK: Default values are taken by uninitialized member-variables of classes and objects (page 148). Local variables in methods have to be initialized (page 121).

Floating-Point Type

type coding/decoding AND semantic manipulation In mathematics, rational numbers are numbers which also may represent fractions of a given unit, for example one-and-a-half ($1\frac{1}{2} = 1.5$), one-eighth ($1/8 = 0.125$). — Number-types which allow to store integers together with fractions of the integer unit are called floating-point types. The range of the integer-part of the number AND the minimal interval of the smallest representable fraction determine the number of the bytes needed to represent a number of floating-point type. The existence of a smallest representable fraction provokes rounding errors.



FloatingPointType := float | double

Remember that exponentiated numbers ($35E2 = 3.5E3 = 3.5 \cdot 10^3$) or period-pre-, post-, or infix numbers (.1, 2.1, 3.) represent literals of the primitive-type double. Thus, the default type of floating-point-literals is given by the primitive-type double. Such a literal can be given the type float when postfixed by the letter “F” or “f”.

```

float fourbytes = 2E2f + 200f;
// range:      +/-3.40282347 E+38 in
// accuracy: 1.40239846 E-45 steps
// default: 0.0f
// postfixed "f" indicates a float-literal
// signed rationals, coded by 4 bytes (32 bits)

double eightbytes = fourbytes + 200.;
// range:      +/-1.79769313486231570 E+308 in
// accuracy: 4.94065645841246544 E-324 steps
// default: 0.0
// signed rationals, coded by 8 bytes (64 bits)

eightbytes = 0./0.; // gives NaN (Not a Number)
eightbytes = 1./0;  // gives Infinity
eightbytes = 1/0.;  // gives Infinity
                  // System.out.println( eightbytes );
  
```

Operations with floating-point numbers DON'T THROW EXCEPTIONS at all. An overflow indication is made possible by testing for constants given by associated wrapper-classes (java.lang.Float, java.lang.Double): POSITIVE_INFINITY or NEGATIVE_INFINITY or NaN (Not a Number): (see also page 257)

```

if( number == java.lang.Double.POSITIVE_INFINITY )
{ System.out.println("Overflow!"); }
float f = java.lang.Float.NaN;
  
```

(The prefix “java.lang.” or rather the qualifier²⁶ “java.lang” indicates the use of a software-library furnished together with the Java language.)

OUTLOOK: Default values are taken by uninitialized member-variables of classes and objects (page 148). Local variables in methods have to be initialized (page 121). (For type-casting see page 88) For later use with multithreaded procedures observe that the Java Virtual Machine (JVM) is specified having a 32bit bus-architecture and thus has to treat operations of 64bit variables (long and double) in two steps. In multithreaded programs this may make it necessary to operate on these variables within a machine-synchronized block of code (page 216).

Java: Declaration, Instantiation of Variables with Primitive-Type

Java: Declaration, Instantiation of Variables with Primitive-Type — Just for illustration, this section presents some ways to create variables of primitive-type in Java:

```
// Declaration and Instantiation: Variable of primitive-type:
PrimitiveType name; name = Value;
PrimitiveType name = Value; // same, but shorter
```

For more examples refer to the previous print-outs of Java source-code or consider the list below. A value is given by a primary-expression like a literal or a variable. The next lines list assignments of values to variables of primitive-type:

```
int name1 = 10; // two-byte integer
int name2 = 65L; // Long 64 bit integer implicitly type-casted.
int name3 = 2E3; // (2 times 10 to the power of 3 is 2000 integer!)
// Exponential notation for numbers only possible for primitive-types double, float.
int name4 = 02; // (octal numbers have a leading zero) octal numbers
int name5 = 0xA; // (leading 0x) hexadecimal numbers
// with A=10,B=11,C=12,D=13,E=14,F=15.
boolean name6 = true; // or false (default: false) truth values
float name7 = -3.1415E2
```

A fully-functional Java program (already introduced around page 76) that does only declare and instantiate a variable of primitive-type is given below:

```
class VariableOfPrimitiveType
{ public static void main( String unused[] )
  { int i; // declaration
    i=1; // instantiation
  }
}
/*
[localhost:~/JavaCodeFiles/Basics1/PrimitiveTypesMore] marycosway% javac VariableOfPrimitiveType.java
[localhost:~/JavaCodeFiles/Basics1/PrimitiveTypesMore] marycosway% java VariableOfPrimitiveType
[localhost:~/JavaCodeFiles/Basics1/PrimitiveTypesMore] marycosway%
*/
```

Java: Type-Casting I

Java: Type-Casting I — Remember that a type of a variable, apart from furnishing methods for manipulating the data, includes means for transforming data into a binary form (format on page 33, type on page 73). Here the SAME BINARY DATA is interpreted using two different primitive-types (byte and int), the respective values are likely to differ — as could be expected. This represents an elementary example about how the data of a variable can copied into another variable of differing type and the effects such a transformation might have.

²⁶A qualifier denotes a prefix separated by a period (.) from the to be prefixed entity.

```

/* Here the variable of primitive-type 'int' represents unsigned numbers,
   ranging from 0 to 255.
   The variable of primitive-type 'byte' represents signed numbers,
   the following program demonstrates that:
*/
public class IntByteCoding
{ public static void main( String unused[] )
  { for( int i = 0; i < 256; i++ )
    { byte b = (byte)i;
      System.out.println( "int " + i + "\t byte " + b );
    } // end for
  } // end method main()
} // end class

/* The piece of source-code given by the "for"-keyword
   and the following pair of parentheses and the following pair of braces
   make the computer revisit the piece of code in braces, here 255-times.
   The expression (byte)i transforms the 'int'-value into a 'byte'-value,
   the printout can be seen below:
*/

/*
[localhost:~/JavaCodeFiles/Basic1/PrimitiveTypesMore] marycosway% javac IntByteCoding.java
[localhost:~/JavaCodeFiles/Basic1/PrimitiveTypesMore] marycosway% java IntByteCoding
int 0    byte 0
int 1    byte 1
int 2    byte 2
int 3    byte 3
...
int 125  byte 125
int 126  byte 126
int 127  byte 127
int 128  byte -128
int 129  byte -127
int 130  byte -126
...
int 251  byte -5
int 252  byte -4
int 253  byte -3
int 254  byte -2
int 255  byte -1
[localhost:~/JavaCodeFiles/Basic1/PrimitiveTypesMore] marycosway%
*/

```

type-casting

Type-Casting: Assigning variables of different types to each other, or assigning a value to a variable of different type is called **type-casting**.

This means taking a variable (here `i`) with a definite type (here `int`) and value, and — by assignment to a variable (here `b`) of different type (here `byte`) — reinterpreting the binary data of the value according to the rules of the different type. As can be seen above, in most cases the value remains the same (`i = 0 ... 127`), but as can be seen above too, there are cases where the same binary data gives an entirely different result (`i = 128 ... 255`).

An example of assigning a value to a variable of different type already could be seen in the previous section, there numerous variables (“name1” to “name5”) of primitive-type `int` have been assigned literals of differing type. For more on type-casting see page 100.

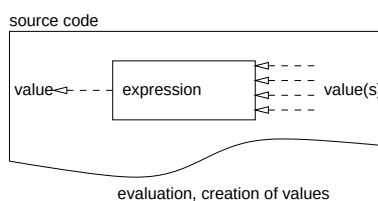
Processing Data

Processing Data — Declaring and instantiating a variable tells the computer how to code and where to save the data in its Random Access Memory (RAM). Expressions, statements and control structures tell the computer how to manipulate, how to transform, the data. The type-casting expression `(byte)i` which has been demonstrated in the latest program exemplifies this; actually even a simple assignment represents an expression. In that same program above, the `for`-keyword makes the computer multiply re-run the code-block which follows the `for`-keyword, this constitutes an example of a control structure.

As part of a runnable program the aforementioned structures, declarations and control structures, may reside inside the `main()`-method of a class:

```
class Program
{ public static void main( String[] cmdlineInput )
  { /* Declarations of identifiers */
    /* Statements */
  } }
```

Concept: Expressions, Statements, Control Structures



Concept: Expressions, Statements, Control Structures — In the source-code of most programming-languages, an **expression** usually forms a part of the source-code which returns a value. An expression takes inputs and gives an output; in that sense an expression resembles a mathematical function, although an expression does not have a name of its own. An expression evaluates information within a single syntactical construct, sub-expressions are parts of the embedding expression.

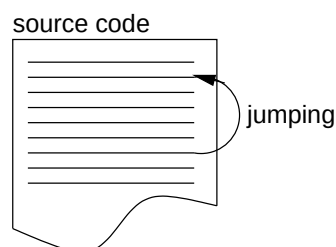
expression

statement

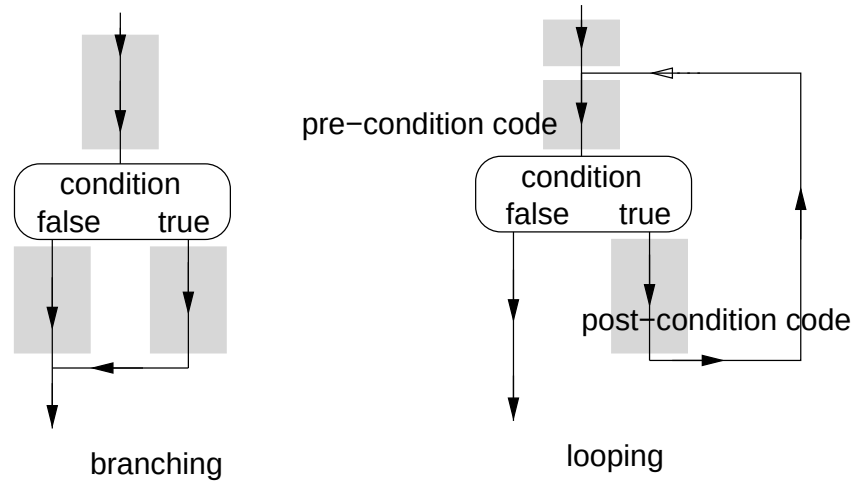
Statements approach information-management more in the form of sequencing commands. A **statement** is given by a part of the source-code that does not automatically return any values. Statements are single commands or ordered collections of commands.

control
structure

Control structures are statements, implementing branching-actions within the source-code. This includes omitting pieces of source-code or jumping back within the source-code, which can lead to looping, if done repeatedly.



Below can be seen clearly how branching can lead to an omission of code:



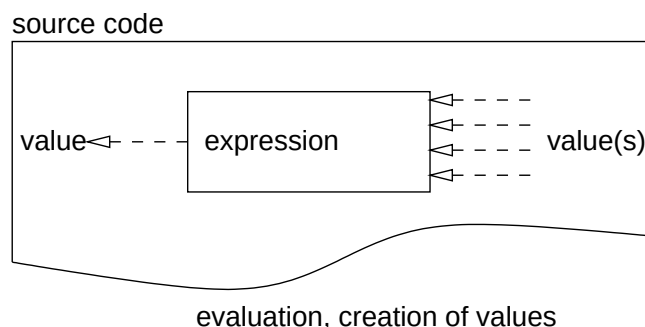
In the scheme above, observe that loops contain pre-condition and post-condition parts. Pre-condition parts are always worked through before the branching takes place. Pre-condition **code** defines so-called **pre-condition loops**. Pure pre-condition loops are given by **do-while** statements, see page 109. Post-condition parts are worked over only when the condition favors them; pure **post-condition loops** are given by **while-loops** and by **for-loops**, see page 108. Then, there can be imagined mixtures of both (**break**, **continue**, see page 110).

pre- and
post-condition
loop

Java: Expressions



Java: Expressions — Virtually all expressions produce a value²⁷. Expressions typically fetch, compute and/or store values; this is done using variables (names of primitive- or reference-type for given data) or literals. Expressions usually produce pure values (literals), variables of primitive-type, objects or arrays. Expressions become a value or are immediately associated with retrieving, calculating or storing values.

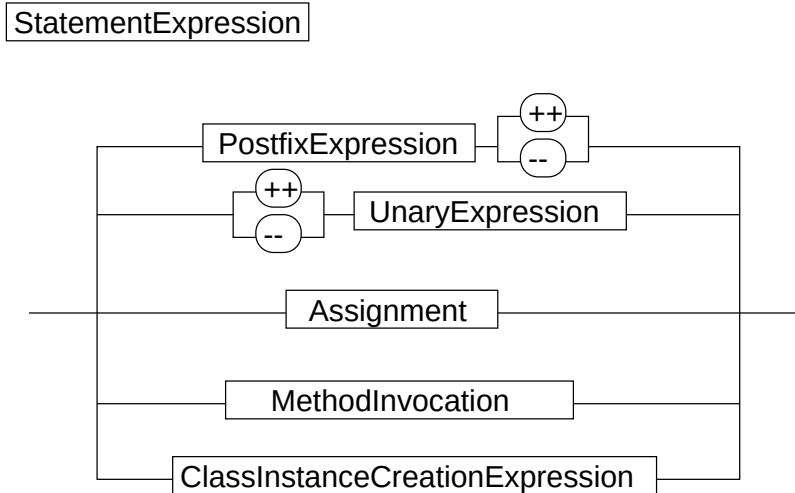


The syntactical structure of the Java language denotes an Expression to be a so-called StatementExpression:



²⁷The only expressions which return no values are method-inocations of void methods, which are so-called StatementExpressions at the same time.

Statement-Expressions include expressions that describe an assignment or the, to be introduced, method-involutions or object-creation:

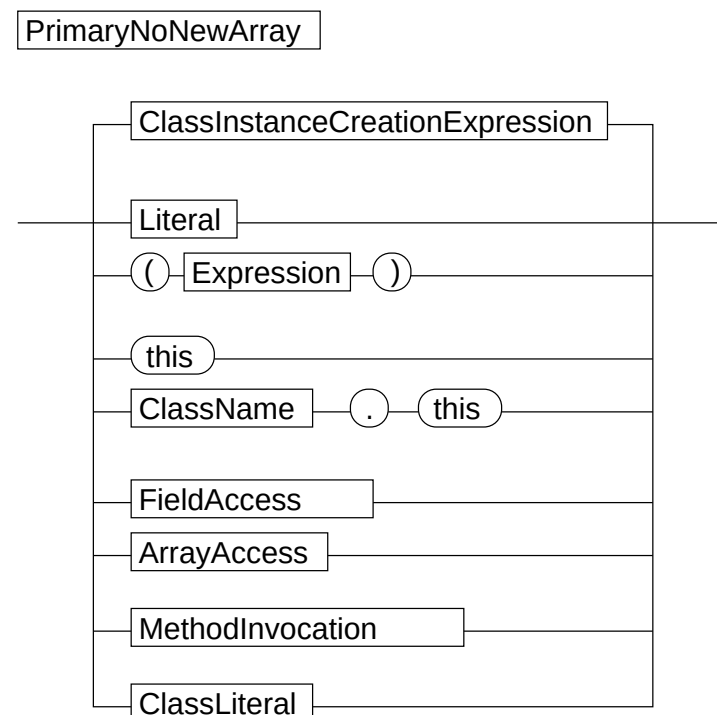
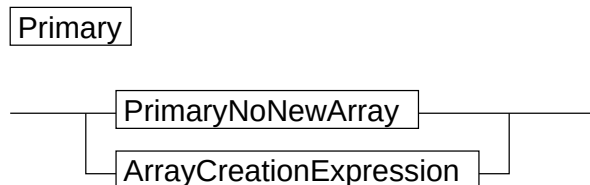


StatementExpression := *ClassInstanceCreationExpression* | *MethodInvocation* | *Assignment* | *Pre- / Post- / In- / DecrementExpression*

— *ClassInstanceCreationExpression* 151, *Assignment* 99, *MethodInvocation* 118

Pre-/Post- In-/DecrementExpression 92

Primary expressions, also called primaries, are parts of expressions considered to be of some elementary kind: Literals, field accesses, method involutions, array accesses, class-literals. Primary expressions fetch (Literal, ... Access, MethodInvocation) or create (ArrayCreationExpression) values, but usually they do not evaluate them.



PrimaryExpression := *PrimaryNoNewArray* | *ArrayCreationExpression*

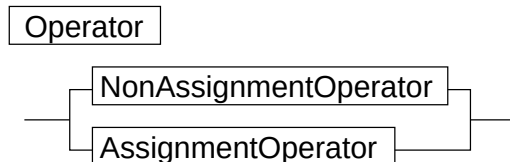
PrimaryNoNewArray := *ClassInstanceCreationExpression* | *Literal* | "(" *Expression* ")" | *this* | *ClassName* .*this* | *FieldAccess* | *ArrayAccess* | *MethodInvocation* | *ClassLiteral* — *ArrayCreationExpression* 234 ,

ClassInstanceCationExpression 151, Literal 77, Expression 90, ClassName = ClassType 146, FieldAccess 156, ArrayAccess 235, MethodInvocation 118, ClassLiteral 437 (Why is the preceding definition divided? See ArrayAccess on page 235.) The keyword `this` can be seen introduced beginning with page 166 and page 172. Primary expressions which produce a reference-type are prominent parts of FieldAccess expressions (page 156).

Operator Expressions — Calculate, Compare and Change



Operator Expressions — Calculate, Compare and Change — Operators change expressions. There are two kinds of operators found in Java:



Operator := *NonAssignmentOperator* | *AssignmentOperator* — AssignmentOperator 99

NONASSIGNMENTOPERATORS are `>`, `<`, `!`, `~`, `?`, `:`, `==`, `<=`, `>=`, `!=`, `&&`, `||`, `++`, `--`, `+`, `-`, `*`, `/`, `&`, `|`, `^`, `%`, `<<`, `>>`, `>>>`, their meaning will be clarified in the following sections.

Especially with variables of arithmetic type, using operators can produce unwanted results: An integral type overflow produces no exception! Dividing a value of integral type by zero gives the only possible `ArithmeticException` (page 280). A floating-point type overflow yields a not-a-number NaN, which can be tested by comparing the variable with the NaN-constant of the wrapper-class (page 86).

Increment/Decrement and Prefix Operators



Operators with one argument are called unary operators. Assume that the operator expressions below are valid with variables `x` and `y`.

```
- x // negates numbers of type byte, short, int, char to int and float to float
x++ // after having run current command a one is added
++x // before having run current command a one is added
x-- // after having run current command a one is subtracted
--x // before having run current command a one is subtracted
int i=2,j=2;
1+ i++; // gives 3 and i becomes 3
1+ (++j); // gives 4 and j becomes 3
i+(i-- )+i; // gives 3+3+2=8
int i=j=0;
System.out.println("i++ = " + (i++) + " then i= " + i +
                  " ++j = " + (++j) + " then j= " + j );
// gives: i++ = 0 then i= 1 ++j = 1 then j= 1
```

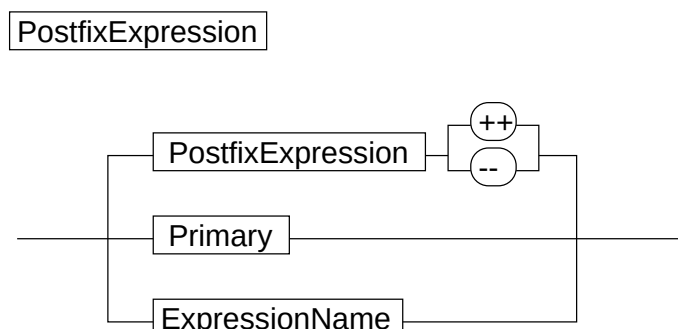
Here prefixing means incrementing or decrementing before using the value. Post-fixing means the value is used, then it is changed. This works with any primary-expression of `ArithmeticType`. Both operators are equal in precedence. An example of `-` and an remark on `-` the effective use of pre-deprecation can be found on page 127. The following definitions already have been visualized in an railroad diagram on page 91.

PreIncrementExpression := `++` *UnaryExpression*

PreDecrementExpression := `--` *UnaryExpression*

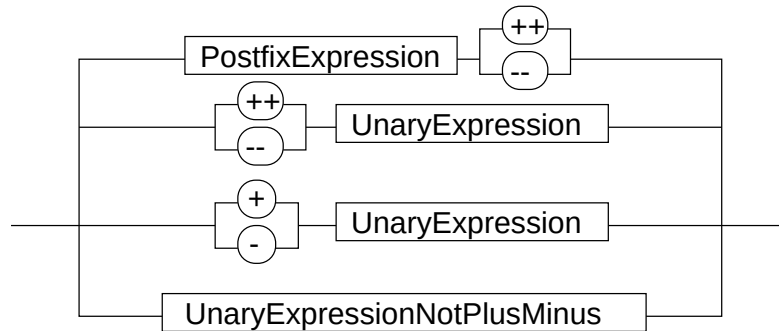
PostIncrementExpression := *PostfixExpression* `++`

PostDecrementExpression := *PostfixExpression* `--`



PostfixExpression := *Primary* | *ExpressionName* | *PostIncrementExpression* | *PostDecrementExpression*
 — *Primary* 91

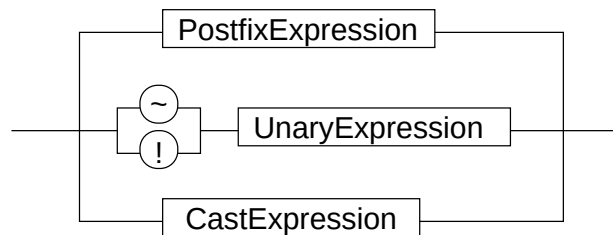
UnaryExpression



UnaryExpression := *PreIncrementExpression* | *PreDecrementExpression* | + *UnaryExpression* | - *UnaryExpression* | *UnaryExpressionNotPlusMinus*

Where the *UnaryExpression* returns a value of arithmetic type, the symbols '+' and '-' are taken as the unary plus- and minus-operators.

UnaryExpressionNotPlusMinus



UnaryExpressionNotPlusMinus := *PostFixExpression* | ~ *UnaryExpression* | ! *UnaryExpression* | *CastExpression* — *CastExpression* 101 (divided because of *CastExpressions* take no prefixed- or postfixed expressions)

boolean- and bit-wise negation

Where the *UnaryExpression* gives values of boolean type, the exclamation-point '!' becomes the boolean negation-operator (!true=false). — Where the *UnaryExpression* is of int-type, the tilde '~' becomes the bitwise negation-operator (~0 == -1 should give true) on integer data-type. See also page 97.

Arithmetic: Multiplicative Operators

Assume that the variables x and y are of some arithmetic type, then the expressions below are commented with their mathematical meaning:

```
x * y      // multiplication
x / y      // division
x % integer // REMAINDER of division with integer (modulo arithmetic)
```



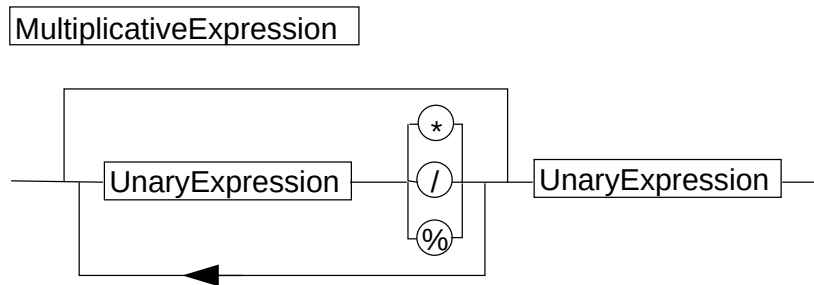
Operators with two arguments are often called binary operators. The binary multiplication- binary operator * usually produces the mathematical product of the two operands. The binary division- operator / usually produces the mathematical quotient of the two numbers. Operands of type integer give an INTEGER RESULT, WITH THE RESULT OF A DIVISION ROUNDED TOWARDS ZERO. The remainder-operator % subtracts the second value from the first, as long as there occurs no change of sign; then the rest is called the remainder, the value which is returned. For pure integer-data the remainder- or modulo-operator can be emulated: a%b = a - ((a/b)*b). For example: 28%5 = 28 - ((28/5)*5) = 3 (This works because the result of the division (28/5 = 5.6) is rounded towards zero (giving 5))

The multiplicative operators are equal in precedence and evaluated from left to right.

```

a=6; b=2;
System.out.println(a*b + " " + a/b + " " + (a+1)/b + " " + (a+1)%b );
// yields 12 3 3 1

```



MultiplicativeExpression := { *UnaryExpression* (* | / | %) } *UnaryExpression* — *UnaryExpression* see above

Arithmetic: Additive Operators



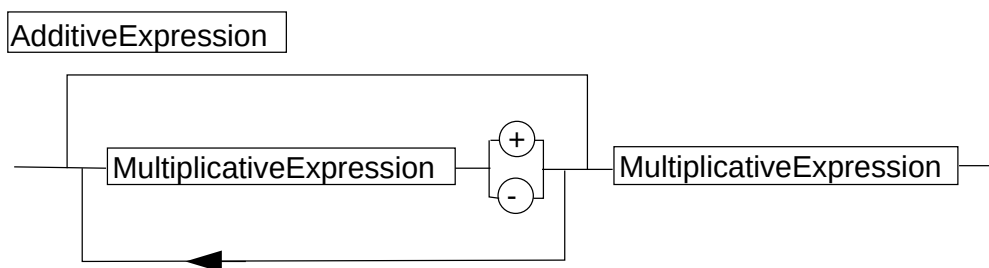
Assume that the variables *x* and *y* are of some arithmetic type:

```

x + y // addition of numbers or concatenation of strings
x - y // addition of negated value of a number, short for x + (-y)

```

Additive operators are equal in precedence and evaluated from left to right. For the illustration of string-concatenation see one of the following subsections.



AdditiveExpression := { *MultiplicativeExpression* (+ | -) } *MultiplicativeExpression*, — *MultiplicativeExpression* see above

Arithmetic: Roots, Exponentials, Logarithms and Powers



Elementary mathematical functions like addition and multiplication are implemented as electronic circuits on almost every computer, but there are other mathematical functions:

sqrt(double s) returns the (usually approximated) result of the mathematical expression $\sqrt{s}$, the so-called “square root of the argument *s*”. The square root of a number *s* is given by some other number *r* also named $\sqrt{s}$, which multiplied by itself (“squared”) returns the number *s*, in symbols: $r \cdot r = s$ or in another notation: $\sqrt{s} \cdot \sqrt{s} = s$.

exp(double x) returns the (usually approximated) result of the mathematical expression e^x , also written $\exp(x)$, the so-called “exponential of the argument *x*”. The exponential of a positive integer *x*

can be written as follows: $\overbrace{e \cdot e \cdot \dots \cdot e}^{x\text{-times}}$, where *e* denotes the Eulerian number as introduced in the footnote on page 257. For non-integer numbers is used the displayed formula in the definition of the exponential below. — Why is that exponential function so meaningful? This function incorporates a lot of special features which make it interesting for mathematicians (The exponential function happens to be its own derivative: $\frac{d}{dx} \exp(x) = \exp(x)$) as well as for advanced applications (The exponential function also represents a solution of one of the simplest differential equations which describes continuous growth processes, the growth(=derivative) being proportional or equal to the value: $\frac{d}{dx} f(x) = f(x)$ solved by $f(x) = \exp(x)$).

log(double y) returns the (usually approximated) result of the mathematical expression $\ln(y)$, the so-called “natural logarithm of the argument $x > 0$ ”. The natural logarithm also acts as the inverse-function ($x = \ln(e^x)$ and $y = e^{\ln(y)}$) of the exponential function above. (The exponential yields positive results for all rational numbers, therefore its inverse takes only positive arguments.)

pow(double x, double y) returns the (usually approximated) result of the mathematical expression x^y , in words: Rising the first argument x to the power of the second y . This means multiplying the first argument x with itself y -times, in symbols: $\overbrace{x \cdot x \cdot \dots \cdot x}^{y\text{-times}}$. This mathematical function (power function) also can be defined sensibly for non-integral values y . ($x^y := \exp(y \cdot \ln(x))$)

These functions can be used by prefixing them with the string `"java.lang.Math."`. Consider, for example, the following line of source-code: `int e = java.lang.Math.exp(1);` The qualifier “`java.lang`” indicates the use of a software-library (page 249) furnished together with the Java language. The class “`Math`” — a capital letter, by convention indicates a class — gathers mathematical functions.

How are these functions calculated by the computer? The `exp`-function, for example, can be calculated from successive additions and multiplications of the following kind until the result appears to be sufficiently precise:

$$\exp(x) = 1 + x + \frac{x^2}{2} + \frac{x^3}{2 \cdot 3} + \frac{x^4}{2 \cdot 3 \cdot 4} + \frac{x^5}{2 \cdot 3 \cdot 4 \cdot 5} + \dots$$

The results of the other functions can be calculated in similar ways. For more about these functions see for example <http://www.hypermath.com> or books on calculus.

Non-Arithmetic: String Concatenation



In Java, the only overloaded²⁸ non-arithmetic operator is given by the “+”-operator: From various string-objects can be produced ANOTHER string-object that is given by a concatenation of each single string-information:

```
"abc" + "def" + x // string-concatenation-operator
```

Overloading of functions or methods is discussed later extensively. But the gist of overloading already can be seen at this point: The binary operator “+” can be used with different types of variables: `String`, `int`, `float` (`"abc" + "def"`, `23+24`, `12.+1.8`) giving different types of results. For more on `String`-objects see page 257.

The concatenation-operator already has been used extensively in the `println()`-method, where variables of primitive-type and variables of reference-type are automatically converted into their respective `String`-representations.

Bitwise: Shift-Operators

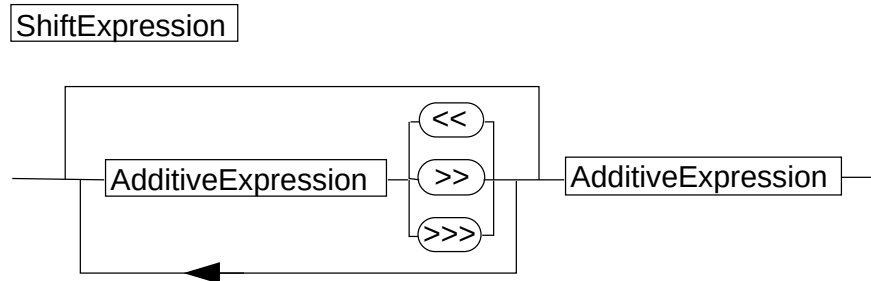


Bitwise shift-operators act on integer-types especially `int` (4 bytes) and `long` (8 bytes). The sequence of bits which stores the data of the integer variable `x` is shifted by an amount of `y` bits. The shift is done destructively, information shifted out gets lost and either zeros or ones are shifted in. The three operations have been listed below:

```
x>>y // (unsigned right shift)
      // fills from the side of highly significant bits with zeros:
      // 1011...1101 >>> 2 yields 001011...11
x>y // (right shift)
    // fills from the side of highly significant bits:
    // a leading '0' triggers filling zeros,
    // a leading '1' triggers filling ones:
    // 0011...1101 >> 2 yields 000011...11
    // 1011...1101 >> 2 yields 111011...11
x<y // (left shift)
    // fills from the side of least significant bits with zeros:
    // the highest significant bit remains as given!
    // 1011...1101 << 2 yields 11...00011100
```

²⁸For more on overloading functions or methods see page 124.

All evaluate from left to right and throw no exceptions. When shifting by multiples of four (bits), the effects can be illustrated well in hexadecimal notation (page 34), because one hexadecimal digit represents four bits of information! — For what are shift operators useful? Mainly in the (advanced) cases, where an integer-value is not seen by the programmer as an integer-number, but as an array of four single bytes: This happens when storing bitmaps of images or when working explicitly with formats or types on bit-level. The shift operator allows bits to be read out and to be written in individually. For a more concrete example see the source-code of a transformation utility (class “Transform”) on page 533.



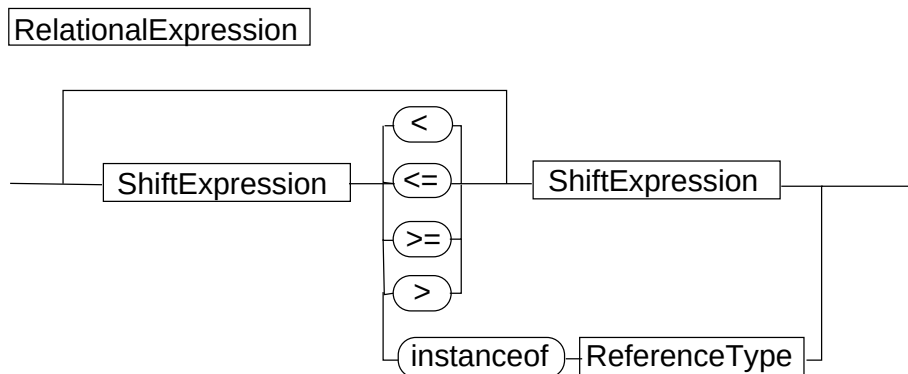
ShiftExpression := { *AdditiveExpression* (<< | >> | >>>) } *AdditiveExpression* — *AdditiveExpression* 94

Non-Arithmetic: Relational Expression



Smaller, larger, smaller or equal, larger or equal are operators that can be used with number-variables, the expressions then return a boolean value.

```
x < y
x <= y // NaN-expression as operand gives false,
        // -0.0 is the same as 0.0
x >= y
x > y
'a' > 'b' // yields false
```



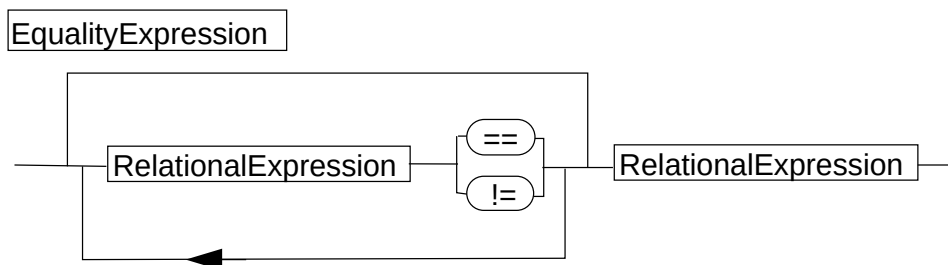
RelationalExpression := [*ShiftExpression* (< | <= | >= | >)] *ShiftExpression* | (*ShiftExpression* instanceof *ReferenceType*) — *ShiftExpression* 96, *ReferenceType* 143

(The definition above differs from the version printed in section 15.20 of the Second Edition of the  Java Language Specification, in that the denotation *RelationalExpression* is substituted by a *ShiftExpression*. This shouldn't change the meaning but reduce complexity.)

Un/Equality: Comparison Operators



Expressions, that test two expressions for equality (==) or inequality (!=), produce a boolean value and are usually found in branching statements (if, while).



EqualityExpression := { *RelationalExpression* (= = | ! =) } *RelationalExpression* — *RelationalExpression* see above

Avoid the use of the equality operator (==) with values of type float or type double, due to possible problems arising from rounding errors!

OUTLOOK: With variables of primitive-type these operators test the identity of the values (the description of a variable of primitive-type beginning on page 83 to be compared with the description of a variable of reference-type beginning on page 143). — With variables of reference-type the identity of the *references* is tested, having the effect that *different references* MAY point to the SAME values, then the test for INequality only concerns the identity of the pointers, NOT the identity of the values they are pointing to. Thus, the result of testing two variables of reference-type with the inequality operator (!=) may yield true though their two values are identical, for details see page 162).

Bitwise and Logical Operators (Single Character: &, ^, |)



There exists a part of mathematics that works with variables which can only take two states, usually denoted **true** or **false**, often also abbreviated to “one” or “zero”, respectively. This part of mathematics is called “**boolean algebra**” and the values of those variables are sometimes denoted “truth”-values. Since also mathematics has to take its inspirations from real-life observations, operations on boolean variables reflect the common interpretations of such conjugations as AND, OR, eXclusive OR or NOT. This can be seen by using these conjugations in simple sentences and comparing their truth-values with the outcome in the table below. “This door has been locked with two locks. If, the first key is available (true/false) AND the second key is available (true/false), then the door can be opened (see table).” (In the table below, anyone may substitute “1” by “true” and “0” by “false”).

Input →	→	→	→	→	→	→	x	y				
Output →		x AND y		NOT x						x OR y		x XOR y
		0		1			0	0		0		0
		0		1			0	1		1		1
		0		0			1	0		1		1
		1		0			1	1		1		0

Thus, conjugations of human language are given a rigid, less rich, meaning in mathematics and programming. Observe also, that in some cases, the outcome of the logical operations AND and OR is already determined by the value of one argument: If one argument in an AND-expression is “zero” or “false”, then the result is “zero” or “false”. If one argument in an OR-expression is “one” or “true”, then the result is “one” or “true”.

As already hinted at on page 84, in Java, the SINGLE-CHARACTER LOGICAL OPERATORS (&, ^, |) **FULLY EVALUATE BOOLEAN EXPRESSIONS**, even in those cases where one input-argument suffices to produce a valid output, as could be seen before with AND- and OR-operators. Working with variables or expressions of primitive-type boolean yields two-boolean-in giving one-boolean-out settings:

```
boolean x,y; x=true; y=false;
x & y      // logical AND (giving false)
x ^ y      // logical EXCLUSIVE OR (giving true)
x | y      // logical OR (giving true)
!x         // logical NOT (giving false)
```

The considerations above concern expressions of primitive-type boolean; whereas the same SINGLE-CHARACTER LOGICAL OPERATORS (&, ^, |) **WORK BITWISE ON INTEGER-VALUES**. This makes a two-integers-in giving one-integer-out setting, except for the unary operator “~”, the bitwise negation- or bitwise NOT-operator! (The exclamation point (!) denotes Java’s logical NOT-operator (also called boolean negation-operator) when working on expressions of primitive-type boolean.) — For example, the data-
bytes of integer-numbers can be negated bitwise:

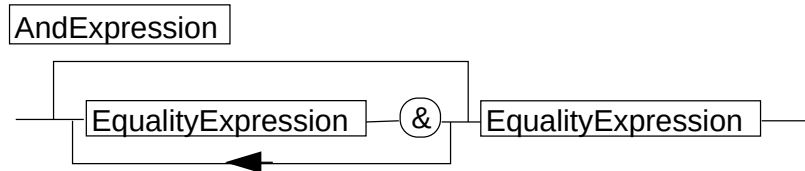
boolean- and bitwise negation

```

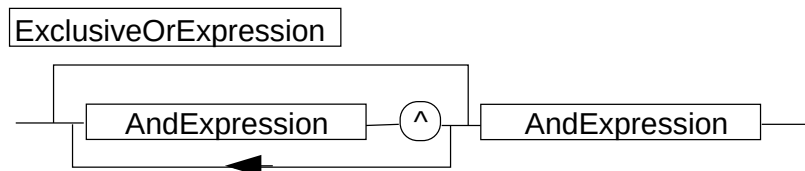
int x,y; x=0; y=2;
~x    // bitwise negation,
      // reverses each bit in the binary code of the integer.
      // ~0 == -1
      // ~10111000 = 01000111 (one byte of an arbitrary four-byte int-value)
x & y // bitwise AND
      // means the binary code of the integers
      // is manipulated bit by bit (giving 0)
x ^ y // bitwise EXCLUSIVE OR (giving 2)
x | y // bitwise OR (giving 2)

```

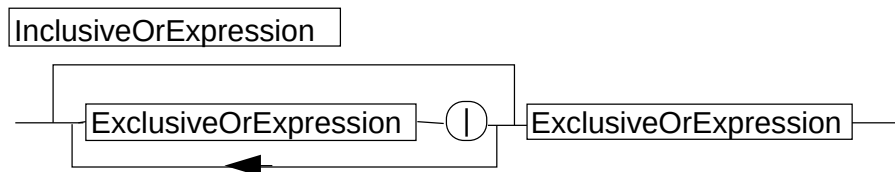
All these operators **ALWAYS EVALUATE BOTH OPERANDS!** They are sorted by precedence:



AndExpression := { *EqualityExpression* & } *EqualityExpression*



ExclusiveOrExpression := { *AndExpression* ^ } *AndExpression*



InclusiveOrExpression := { *ExclusiveOrExpression* | } *ExclusiveOrExpression*

Logical Operators (Double Characters: &&, ||)



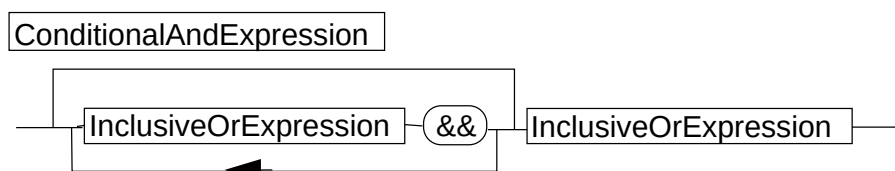
The doubled logical operators (&&, ||) always mean: Two-boolean-in gives one-boolean-out. The logical NOT-operator, also called boolean negation-operator, operates only on one value of boolean type that comes out reversed.

```

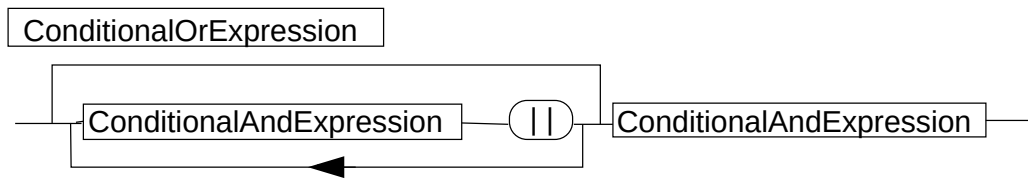
!x    // logical not, substitutes true and false and vice versa.
      //(Mainly used for decisions in loopings and branchings.)
x||y  // logical OR, if x is true, y is left unevaluated.
x&&y  // logical AND, if x is false, y is left unevaluated.
x==y  // logical EQUAL is a test on IDENTITY not equality!
      // See previous page 96.
x!=y  // logical NOT EQUAL

```

These twin operators are also called conditional boolean operators because the AND- and OR-operators **EVALUATE THE FIRST EXPRESSION AND STOP IF THE REST WOULDN'T ALTER THE OUTCOME**: If the first expression of an AND-clause is “false”, then the entire expression must be “false”. If the first expression of an inclusive-OR-expression is “true”, then the entire expression is “true”. This can be of some use when the second term becomes defined only if the first term has a proper value. The syntax is presented according to their precedence:



ConditionalAndExpression := { *InclusiveOrExpression* && } *InclusiveOrExpression*



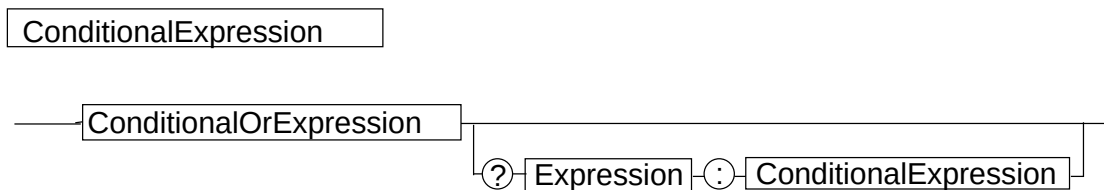
ConditionalOrExpression := { *ConditionalAndExpression* || } *ConditionalAndExpression*

Conditional-Operator

⚙ Anticipating the branching-statement associated with the keyword `if` (page 106), the following can be said: The conditional-operator represents a short form of an `if`-statement:

```
boolean ok = true;
int number = ok ? 1:0 // Converts 'true' to '1' and 'false' to '0'
```

Evaluates from right to left: `x?y:z?a:b` means `x?y:(z?a:b)`. Though this operator makes it possible to write compact code, its use can lead to some confusion. Therefore the following text uses this programming construct scarcely, if at all.



ConditionalExpression := *ConditionalOrExpression* [? *Expression* : *ConditionalExpression*] — *Expression*

Assignment Operators

⚙ As already mentioned, expressions transform data. A simple way of transformation is given by the “assignment”, especially in the case of a variable of primitive-type, where assignment amounts to copying a given value into the storage place of the variable (see page 83):

```
int i; i = 1;
```

Assignments evaluate from right to left: `x=y=z` means `x=(y=z)`, first the Java compiler treats `y=z`, then it treats `x=y`.

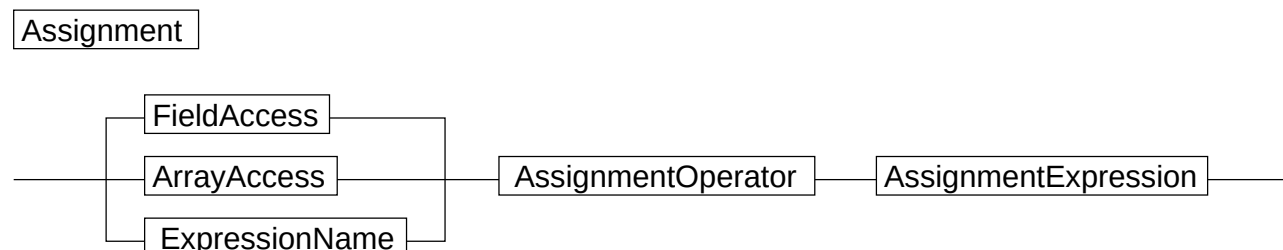
Compound assignment operators for primitive-types are given by the following expression(s):

```
x += y; // means x = x+y,
        // this works for +, -, *, /, |, &, ^, %, <<, >>, >>>
```

Such shortening may mainly help in confusing the reader of source-code; so for better readability of programs, compound assignment may be considered avoidable. (`x op= y` approximately means `x = (xtype)(x op y)`, where `op` represent a binary operator of the kind above)

The frequently used assignment-operator is given by the equal sign (=).

AssignmentOperator := = | += | -= | *= | /= | &= | |= | ^= | %= | <= | >= | >>=



AssignmentExpression := *ConditionalExpression* | ((*FieldAccess* | *ArrayAccess* | *ExpressionName*) *AssignmentOperator* *AssignmentExpression*)

OUTLOOK: Further remarks on assignment, concerning variables of reference-type, can be found beginning with page 143.

Type-Casting II (Casting of Primitive-Types)

Type-Casting II (Casting of Primitive-Types) — Remember the introduction to type-casting on page 88. Type-casting means that data is stored according to a certain type-specification and the resulting sequence of bytes is interpreted according to another type-specification. But, as could be seen, not every cast turns out to be sensible and possible.

Type-Casting implicit

In the mathematical operations above which have two arguments, any combination of `ints` and `floats` is allowed giving `floats`.

```
int + float // gives a float
int + long // gives a long
```

Implicit casting of primitive-types loses no information. Explicit type-casting may lose parts of the original data as can be seen below:

Type-Casting explicit

The characters “L” or “l”, postfixing integer-literals (giving them the type “long”), and the characters “F” or “f”, postfixing floating-point-literals (giving them the type “float”), produce explicit type-casts of number-literals. (The prefixes 0 (for octal) and 0x (for hexadecimal) only indicate the number-system used.) Explicit casting of variables of primitive-type may cause information to be lost, for example by cutting off significant bytes:

```
name2 = (newtype) name1;
byte integer = (byte) 3.1415; // gives the value 3
int name = (int)123456789012345L; // gives -2045911175
```

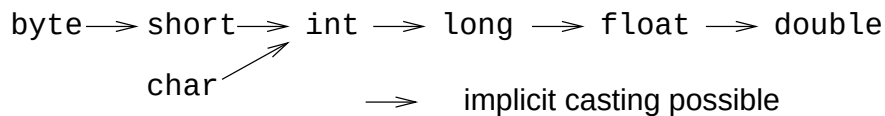
In the following table are considered two variables, `x` and `y`, of primitive-type, indicated by the captions of rows and columns. The proposed assignment

```
x = (primitive-type) y
```

and its implication can be read from the table: (“i” means implicit type-casting as described above and allows to discard the explicit form)

x y→	1 byte	2 byte	2 byte	4 byte - 32 bit	8 byte - 64 bit	4 bytes - 32 bit	8 bytes - 64 bit
↓	byte	char	short	int	long	float	double
byte	=	= (byte)	=(byte)	=(byte)	=(byte)	=(byte)	=(byte)
char	=(char)	=	=(char)	=(char)	=(char)	=(char)	=(char)
short	i	=(short)	=	=(short)	=(short)	=(short)	=(short)
int	i or see below	i	i	=	=(int)	=(int)	=(int)
long	i	i	i	i	=	=(long)	=(long)
float	i	i	i	i	i	=	=(float)
double	i	i	i	i	i	i	=

In a more compressed form the implicit casting can be visualized as follows:



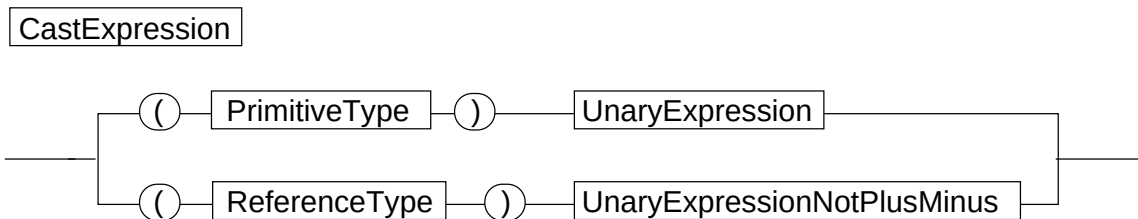
Remember the dangers of type-casting as demonstrated on page 88 and above with casting the long-literal. The effect described there (`signedByte = (byte) unsignedByte`) can be reversed with the following one-liner:

```
int unsignedByte = y < 0 ? y + 256 : y;
```

Generally autocasting (implicit casting), as well as explicit casting, have to be done with care as the line below illustrates:

(float)x/y is not equal to (float)x/(float)y

Explicit casting can be described within the syntactical structure of the Java language:



CastExpression := "(" *PrimitiveType* ")" *UnaryExpression* | "(" *ReferenceType* ")" *UnaryExpressionNotPlusMinus* — *PrimitiveType* 84, *UnaryExpression* 93, *ReferenceType* 143, *UnaryExpressionNotPlusMinus* 93

Type-casting is revisited for reference-types on page 169.

Precedence

Precedence — In the table below, operators are coarsely ordered according to the succession in which they are evaluated within an expression. That is meant by precedence. So a sub-expression of highest precedence always is evaluated first. In the case of same precedence, left-, right-, or non-associativity govern the evaluation: Arithmetic operations of same precedence, for example, are

left-associative, so `x+y-z` means `(x+y)-z`, abbreviated below by the symbol "→".

For example, the assignment and conditional-operators are

right-associative, so `x=y=z` means `x=(y=z)`, abbreviated below by the symbol "←".

Precedence	Operator	Associativity L/R	
1	(), []	NON	argument-list, array-element identification
2	new -keyword	NON	instance-creation
3	.	→	qualification-operator
4	++, --	NON	post-increment-/decrement-operators
5	-, +(unary), !, ~, ++, --, (type)	←	signs, logical and binary negation-operator, pre-increment-/decrement-operators, explicit type-cast
6	*, /, %	→	multiplicative operators, binary
7	+, -	→	additive operators, binary
8	<<, >>, >>>	→	bitwise shift operators
9	<, >, <=, >=, instanceof	NON	relational operators
10	==, !=	→	un/equality comparison-operators
11	&	→	logical/bitwise AND
12	^	→	logical/bitwise EXCLUSIVE OR
13		→	logical/bitwise OR
14	&&	→	conditional AND
15		→	conditional OR
16	?:	←	conditional-operator
17	=, *=, /=, %=, -=, <<=, >>=, >>>=, &=, ^=, =	←	assignment operators, single and compound

Expressions within parentheses and brackets, as well as within binary operators are evaluated in the following order (where the ring represents any binary operator):

$\boxed{1} \overset{3}{\circ} \boxed{2}$
 $\boxed{1} [\boxed{2}]$
 $\boxed{1} (\boxed{2})$
 $(\boxed{1}, \boxed{2})$

where the ring $\circ$ represents any binary operator

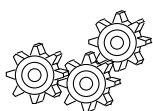
The more important rule about precedence may be given as follows: When in doubt about the preference, use a pair of parentheses to enforce the desired order of evaluation.

OUTLOOK: Precedences 2 and 3 make the following expression meaningful

`new String("abc").toUpperCase();`

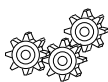
because the application of the new-keyword precedes the application of the qualification-operator (.).

Java: Statements, Control Structures and Programs



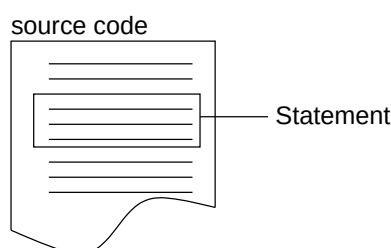
Java: Statements, Control Structures and Programs — A statement represents a piece of source-code, that manipulates information, which it preferably takes from variables and which it usually puts back into variables. Unlike expressions, statements do not return values explicitly. A sequence of statements represents a flow of control or (a part of) a thread.

Java: Statements

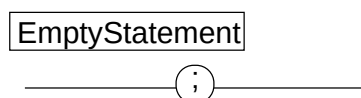


Statements in Java finish with a curly brace “}” or a semi-colon “;”. Statements do not return automatically any value. Though statements may contain expressions that associate them with retrieving, calculating or storing a value, statements also manage the flow of control (branching, looping).

statementexpr;

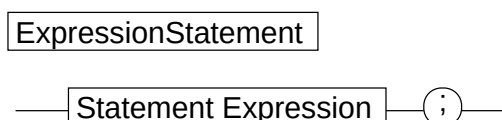


The empty statement does nothing:



EmptyStatement := ;

(One of the few non-trivial cases, where empty statements may be used, are the finalization- or update-parts of a for-statement as can be inferred from the syntax-diagram on page 109.) Some expressions can be made statements by adding a semicolon: Statement-Expressions are exactly those expressions which suffixed by a semicolon become statements. The expression `x=1`, swiftly can be turned into a statement by suffixing it with a semicolon: `x=1;`

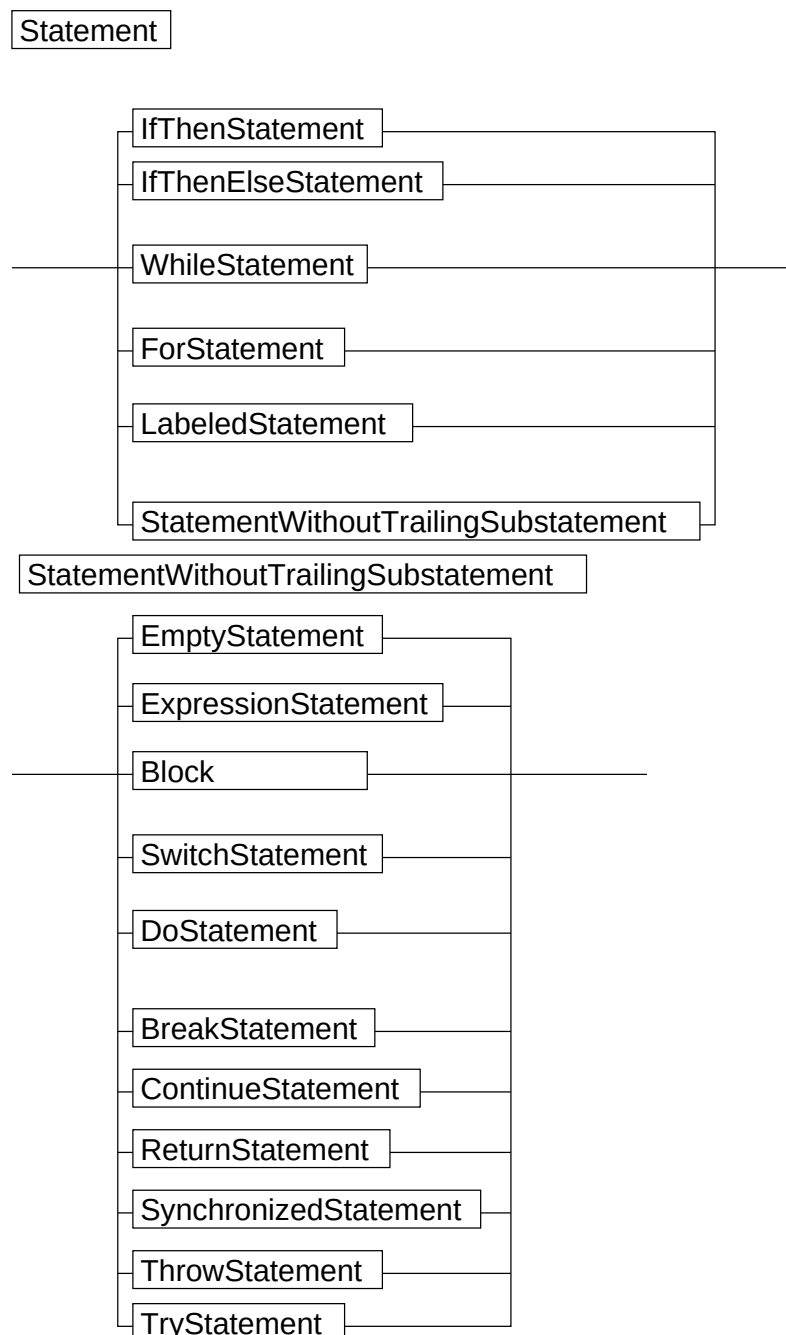


ExpressionStatement := *StatementExpression* ; — *StatementExpression* 91

If in doubt, whether something may be called a statement or an expression, remember: Statements end with a semi-colon, without being a value. Expressions ²⁹USUALLY end as a value or ³⁰USUALLY transform a value in a sequential way, without any alternative flow of control. Generally Java statements can be classified as follows:

²⁹A method-call of a method, which has been declared `void`, returns no value.

³⁰The conditional-operator `?`, though being an expression, offers an alternative to the standard flow of control!



Statement := *IfThenStatement* | *IfThenElseStatement* | *WhileStatement* | *ForStatement* | *LabeledStatement* | *StatementWithoutTrailingSubstatement* — *IfThen/ElseStatement* 106, *WhileStatement* 108, *ForStatement* 109, *LabeledStatement* 111.

StatementWithoutTrailingSubstatement := *EmptyStatement* | *Block* | *ExpressionStatement* | *SwitchStatement* | *DoStatement* | *BreakStatement* | *ContinueStatement* | *ReturnStatement* | *SynchronizedStatement* | *ThrowStatement* | *TryStatement* — *EmptyStatement* 103, *Block* 105, *ExpressionStatement* 103, *SwitchStatement* 107, *DoStatement* 110, *BreakStatement* 110, *ContinueStatement* 110, *ReturnStatement* 118, *SynchronizedStatement* 216, *ThrowStatement* 279, *TryStatement* 283.

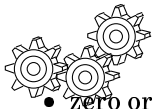
The Java statements above are divided into two sets (*Statement* and *StatementWithoutTrailingSubstatement*), this division and the following definition can be ignored at the first glance³¹.

StatementNoShortIf := *StatementWithoutTrailingSubstatement* | *IfThenElseStatementNoShortIf* | *WhileStatementNoShortIf* | *ForStatementNoShortIf* | *LabeledStatementNoShortIf*

The statements with a suffix “NoShortIf” always miss an ending of the kind ... *if*(...) ... rather they would have an ending like ... *if*(...) ... *else*

³¹Nevertheless this structuralization is chosen because the trailing sub-statement “else” like in the cases *if*(...) ... *if*(...) ... *else* ... or *if*(...) ... *while*(...) ... *if*(...) ... *else* ... might lead to ambiguous interpretations. Java takes such *else* ... blocks as belonging to the innermost construct. Marking blocks properly with additional curly braces resolves this problem anyway.

Block



Blocks are those parts of Java source-code that are enclosed in curly braces.

Blocks contain program-statements:

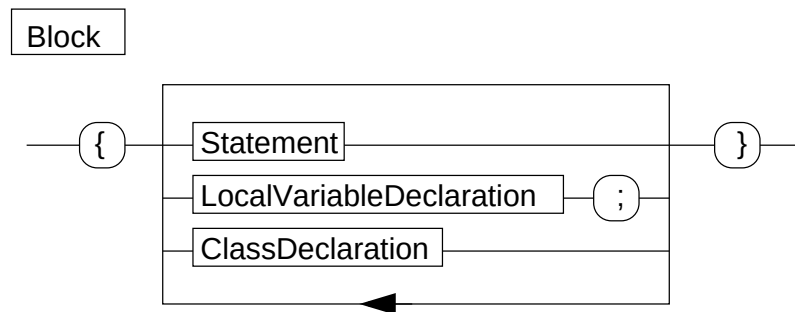
A block is given by a sequence of

{ ... }

- zero or more statements (method-calls, loops, branching),
- local variable-declarations, or
- local class-declarations

enclosed by curly braces. Blocks ONLY occur as parts of branching- or loop-statements or as the defining parts of methods or classes.

```
{ statement; localVariableDeclaration; } // block
```

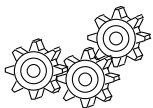


Block := "{" { *BlockStatement* } "}"

BlockStatement := *Statement* | *LocalVariableDeclaration* ; | *ClassDeclaration* — *Statement* 103, *LocalVariableDeclaration* 121, *ClassDeclaration* 146 (*BlockStatement* used in *SwitchStatement* 107 and *ConstructorBody* 150)

OUTLOOK on blocks: The idea of naming blocks characterizes the first step towards what is later called method, for more see page 115. And blocks of so-called anonymous classes can be found later in class-definitions; even in return-statements and parameter-lists, see page 159.

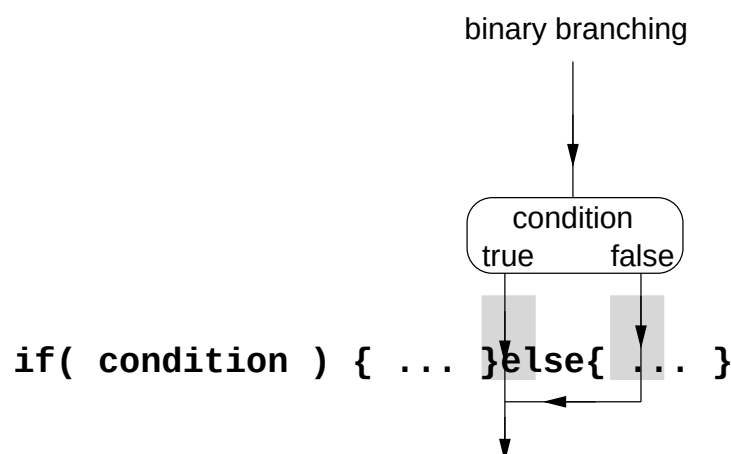
Java: Branching



Branching enables the (Java) programmer to select which block of code or statement has to be worked on next. Therefore, the programmer formulates a conditional expression that tells the computer which block of code or statement has to come next. In branching and looping all parentheses "(" with single entries contain expressions that return boolean values!

Binary Branching — If-Statement

In the graphic below the labels true and false have been switched (with respect to the graphic around page 89) to accommodate the overlay of the Java programming construct:



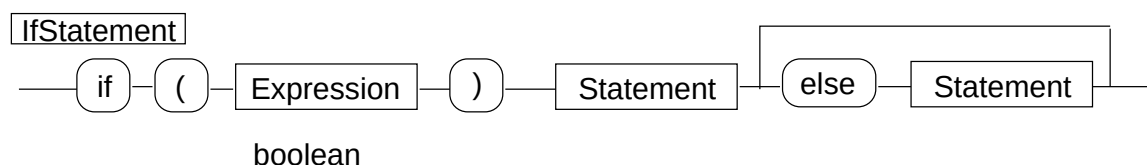
Binary branching essentially means jumping forward, ignoring code. The Java programmer can select between two basic code-alternatives:

```
if(x<y){...};
if(!x & y){...}else{...};
```

The expression in parentheses needs to be of primitive-type boolean. The statement can be identified as right-associative with respect to a counterpart of its own. But nevertheless it may be advisable to rigidly adhere to explicit bracing, even when it seems unnecessary: The version of nested if-statements — below on the right side — greatly improves the readability of source-code:

```
if(x<y) if(y<z){...} SAME AS  if(x<y) { if(y<z){...} }
```

The Java compiler gives an error-message about unreachable statements (`if(1>2){...}` gives such an unreachable statement). Contrary to unreachable if-statements, an if-statement with a *constant* truth-value (`if(false){...}`) allows so-called conditional compilation in Java, which can be recognized as equivalent to commenting out code!

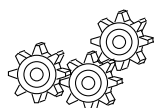


IfStatement := if "(" *Expression* ")" *Statement* { "else" *Statement* } — Expression 90, Statement 103 (Original: *IfThenStatement* := if "(" *Expression* ")" *Statement* — *IfThenElseStatement* := if "(" *Expression* ")" *StatementNoShortIf* "else" *Statement* — *IfThenElseStatementNoShortIf* := if "(" *Expression* ")" *StatementNoShortIf* "else" *StatementNoShortIf*)

The argument of the if-statement — the expression — has to return a boolean value.

```
double r = java.lang.Math.random();
boolean rLarger = r >= .5;
if(rLarger) System.out.println(r + ">=.5");
else System.out.println(r + "<.5");
// output for example 0.34930516006634516 <.5
if(false)
{ System.out.println("Another way of commenting out code"); }
```

Multiple Branching/Cascading — Switch-Statement



Multiple branching, like binary branching means jumping forward, ignoring code. A switch-statement uses a variable of the primitive-type `int` to jump to the statement associated with this number. Though the argument of the switch-statement is taken to be an `int`-value, values of type `char`, `byte` and `short` are implicitly type-casted.

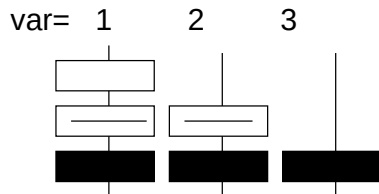
Below, the argument `var` is taken to be a variable of `int`-type:

<pre>switch(var) { case 1: <i>statement1</i>; case 2: <i>statement2</i>; case 3: <i>statement3</i>; };</pre>	<pre>switch(var) { case 1: <i>statement1</i>; break; case 2: <i>statement2</i>; break; case 3: <i>statement3</i>; break; };</pre>

Then there are basically two possibilities for the Java programmer: Either add a `break`-statement and skip the rest of the switch-block OR the next statement will be worked through, cascading down the remaining cases as long as no further `break` occurs. A program-run of the two example switch-statements is illustrated below:

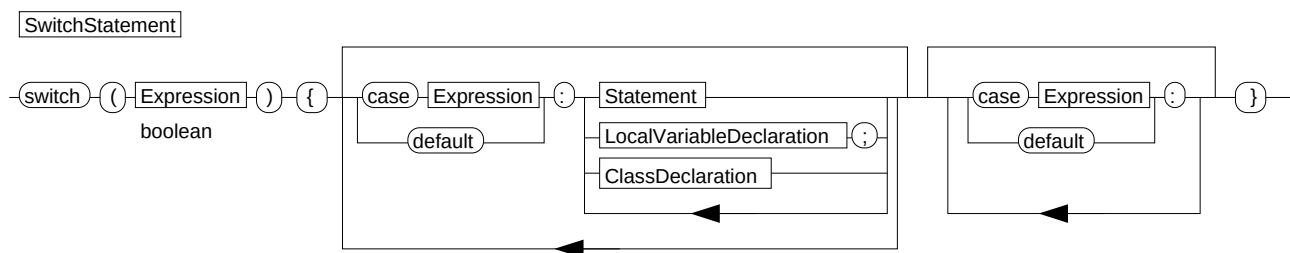
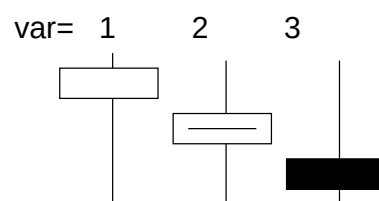
cascading

```
switch(var)
{
  case 1: 
  case 2: 
  case 3: ☒
}
```



selecting

```
switch(var)
{ case 1:  break;
  case 2:  break;
  case 3: ☒ break;
}
```

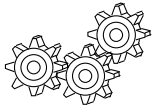

$$\textit{SwitchStatement} := \text{switch } (" \textit{Expression} ") \textit{SwitchBlock}$$
$$SwitchBlock := \{ \{ SwitchBlockStatementGroup \} \{ SwitchLabel \} \}$$
$$\textit{SwitchBlockStatementGroup} := \langle \textit{SwitchLabel} \rangle \langle \textit{BlockStatement} \rangle$$
$$SwitchLabel := \text{case } Expression : | \text{ default} :$$

- Expression 90, BlockStatement 105, LocalVariableDeclaration 121, ClassDeclaration 146

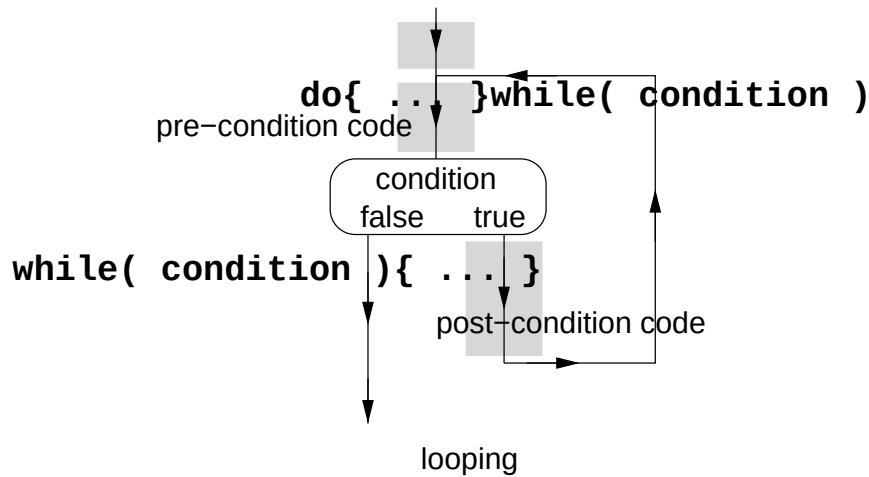
The Expression following the case-keyword has to be a so-called ConstantExpression, this means that the expression has no varying inputs, this kind of expression especially includes literals.

```
byte selector=2;
switch(selector)
{
case 1: System.out.print("case 1");
case 2: { System.out.print("case 2 in standard cascading switch and "); };
case 3: System.out.println("case 3 in cascading switch");
default :
}
/* output: case 2 in standard cascading switch and
    case 3 in cascading switch */
char select=2;
switch(select)
{
case 'A': System.out.println("case 'A'");
// because literal 'A' is of char-type
case 2: { System.out.println("case 2 in NON-cascading switch"); }; break;
case 3: System.out.println("case 3 in NON-cascading switch"); break;
default :
}
// output: case 2 in NON-cascading switch
```

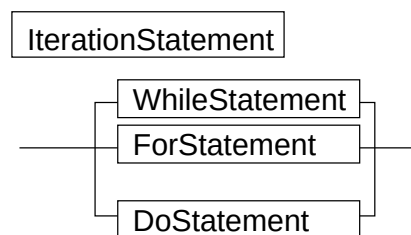
Java: Looping, Iteration



Contrary to binary- and multiple branching, looping essentially means jumping backward, repeating code-runs, usually with differing relevant values. To avoid a non-terminating loop, there has to be a terminating branch.

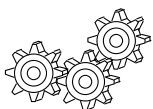


Loops in Java can be constructed with the help of three basic statements:



IterationStatement := *WhileStatement* | *DoStatement* | *ForStatement* — *WhileStatement* 108, *DoStatement* 110, *ForStatement* 109 This definition cannot be found in the Second Edition of the ➡☐ ➡☐ Java Language Specification [JLS], but here it gives a useful preview.

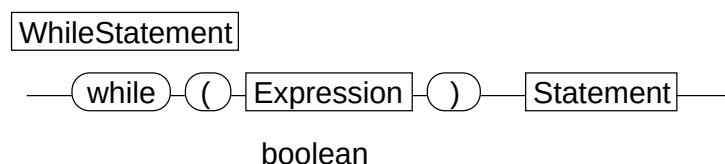
Post-condition Loops — While, For-Statements



Part of the while-statement below is given by a block of code. The expression in the parentheses has to return a **boolean-value**. If the expression in the parentheses evaluates to **true**, then the block is run. After the block has been run, the **boolean-expression** gets re-evaluated, if the expression re-evaluates to **true**, the block is re-run. This is done until the expression evaluates to **false**, then the code which follows the block is run.

```
while( booleanExpression )
{ /* statements */
};
/* code which follows the block */
```

If the **boolean-expression** always returns **false**, then the while-code is not touched!



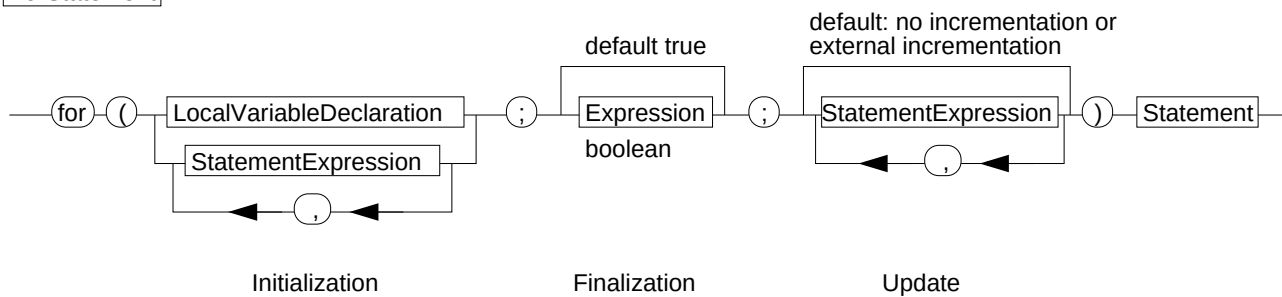
WhileStatement := while "(" *Expression* ")" *Statement* — Expression 90, Statement 103 (*WhileStatement-NoShortIf* := while "(" *Expression* ")" *StatementNoShortIf*)

The while-statement as well as the following for-statement produce *post-condition* loops: The branching decision is made *before* entering the loop block: (The “post” refers to the relative position of the code with respect to the condition.)

while, for: ———(true) [] ———(true) [] ———(false) ———

```
// case i = 3 NEVER touches the "block"
// --- a false condition skips the "block"
// --- post-condition looping
int i = 1;
while( i < 3 ){ i++; /* block */ };
or
for( int i = 1; i < 3; i++ ){ /* block */ };
for( int x = 0; x < 5; x++ ){ /* block */ } // loops five times over the block
```

ForStatement



The Expression needs to be boolean and is true by default.

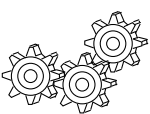
ForStatement := for "(" [*ForInitialization*] ; [*Expression*] ; [*ForUpdate*] ")" *Statement*

ForInitialization := *LocalVariableDeclaration* | *StatementExpression* { , *StatementExpression* }

ForUpdate := *StatementExpression* { , *StatementExpression* } — Expression 90, Statement 103, *LocalVariableDeclaration* 121, *StatementExpression* 91, (*ForStatementNoShortIf* := for "(" [*ForInitialization*] ; [*Expression*] ; [*ForUpdate*])" *StatementNoShortIf*)

One of many examples for a post-condition loop can be found in the source-code on page 390.

Pre-condition Loops — Do-Statement



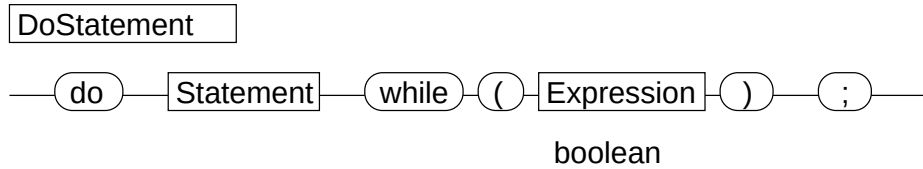
Here, *after* running through the block of statements, the computer decides whether to re-run it or not. In any case, the block of statements is run at least once! (Contrary to the post-condition loops, where the code-block may not be run at all.) Observe the `do`-keyword prefixing the block:

```
do
{ /* statements */
}while(expression);
```

Again, in any case, in pre-condition loops, the statement is worked over at least once:

do: ——— [] (true) ——— [] (true) ——— [] (false) ———

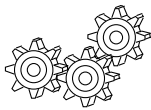
```
int i = 1;
do
{ i++; /* block */
}while( i < 3 );
// -- case i = 3 ALWAYS touches the "block"
// -- a false condition is detected after running the "block"
// -- pre-condition looping
```



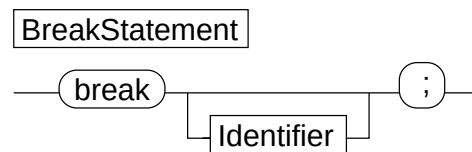
DoStatement := *do Statement while "(" Expression ")";* — Statement 103, Expression 90

For practical examples of pre- and post-condition looping see also their use in the program on page 297.

Restricted Jumping — Break, Continue-Statements — Pre- and Post-Condition Mix

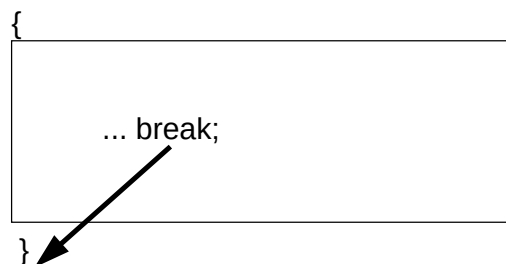


After *switch*, *while*, *do-while*, *for*-statements have finished, they transfer control to the statement that follows the respective branching- or looping statement. As already could be seen in presenting the *switch*-statement, the *break*-statement inside a *switch*, *while*, *do-while*, *for*-statement makes the computer resume work directly at that first statement past the *switch*, *while*, *do-while*, *for*-statement:



BreakStatement := *break [Identifier] ;* — Identifier 81

switch while do-while for



Breaks within an *if*-statement break the enclosing loop as indicated below:

```

while( true ){ if(endTheLoop) break; };
// The break finishes the seemingly infinite while-loop.
// And can turn it into a pre-/post-condition loop.
  
```

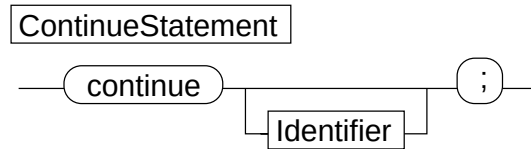
The *break*-statement allows turn pure pre- or post-condition loops into loops with pre- and post-condition parts. Here is given an example of a loop containing pre- and post-condition code:

```

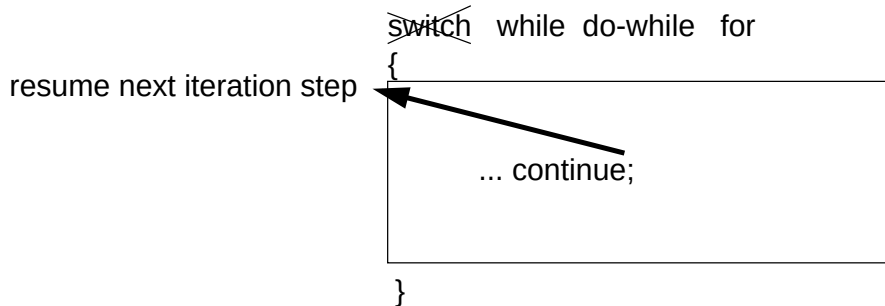
int i = 0; boolean ok;
while( true )
{
  /*pre-condition part:*/ System.out.print( ++i + " " ); ok = i > 5;
  if(ok) break;
  /*post-condition part:*/ System.out.println( ++i );
}
/* output:
1 2
3 4
5 6
7 */
  
```

The construct above allows to enter and leave the loop using the same code-block (pre-condition), and running it alternately the rest of the time with the post-condition block.

A *continue*-statement can only occur inside loop statements (*while*, *do-while*, *for* — NOT *switch*). The *continue*-statement aborts the current iteration-step and begins with the next one. In case of nested loops one of them can be labeled and becomes the starting point of the *continue*-statement.



ContinueStatement ::= continue [*Identifier*] ; — *Identifier* 81



Continue starts the next cycle of the loop!

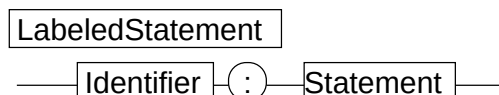
```

boolean skip;
for( int i = 0; i < 6; i++ )
{
    skip = ( i%2 != 0 ); // left associativity would allow to omit the parentheses
    if(skip) { System.out.print(i); continue; }
    System.out.print( " even " );
}
/* output:
even 1 even 3 even 5
*/
    
```

The optional identifier in the break- and continue-statements is used to break to a LABEL OF THE SAME label NAME IN THE IMMEDIATELY ENCLOSING METHOD OR ENCLOSING INITIALIZER-BLOCK:

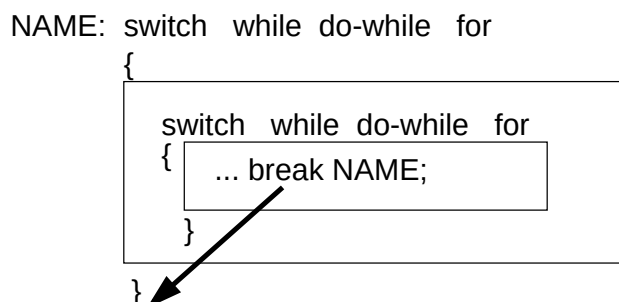
```
labelX: while(true){ if(isTrue) break labelX; };
```

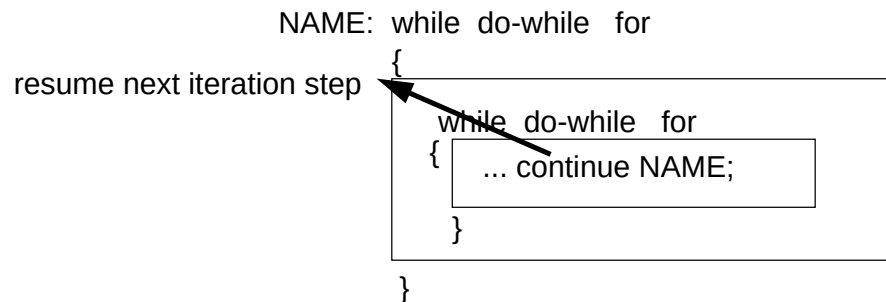
A labeled statement by itself doesn't affect the program flow, but a labeled statement is the sign for the break- and continue-statements where to divert the program's flow. After encountering a break labelX;-statement, that enclosing statement is searched which is labeled by labelX: Then, the label determines the block past which the running of the code is resumed.



LabeledStatement ::= *Identifier* : *Statement* — *Identifier* 81, *Statement* 103 (*LabeledStatementNoShortIf* ::= *Identifier* : *StatementNoShortIf*)

Labeled break-statements are useful for crossing multiple enclosing statements (switch, while, for, do) and labeled continue-statements for crossing multiple enclosing loop-statements (while, for, do). A labeled continue resumes work at the beginning of the labeled statement with its next iteration-step, whereas a labeled break resumes work at the first statement that follows the labeled statement! If possible, avoid these constructs.





The small program below uses the mechanisms mentioned above:

```

/* the do-loop is jumped from in two ways:
   1. as long as a random number is smaller than .9,
      the outer loop is re-run (while(true);)
   2. otherwise the loop is left with break;
*/
public class Escape
{ public static void main( String unused[] )
  { do
    { if( Math.random() > .9 ) { System.out.print("Success!"); break; }
      else { System.out.print("Caught "); continue; };
    }while( true );
    System.out.println(" ... Broke out");
  } // main()-method
} // class
/*
[localhost:~/JavaCodeFiles/Basics1/PrimitiveTypesMore] andreadipietro% javac Escape.java
[localhost:~/JavaCodeFiles/Basics1/PrimitiveTypesMore] andreadipietro% java Escape
Caught Caught Caught Caught Success! ... Broke out
[localhost:~/JavaCodeFiles/Basics1/PrimitiveTypesMore] andreadipietro%
*/

```

The `continue;`-command above initiates a jump to the `while(true)`-expression. Since this jump actually proves to be unnecessary, here the program works the same without the `continue;`-command!

Restricted Jumping — Try-Catch-Finally

With older programming-languages, runtime error-messages often cut off the entire running program. This situation could be improved with programming-languages that allowed runtime error-messages to be handled by the program; then the messages had to be treated at the point of their occurrence in the code. This practice often muddled the code:

```

open;
load;
close;

```

gives at least three possibilities for errors:

```

open;
if(openok)
{ load;
  if(loadok)
  { close;
    if(closeok)
    {"Task successful"}
    else{"Cannot close"}}
  else {"Cannot load"}}
else {"Cannot open"};

```

The exception concept — as implemented by the Java language — may be considered as presenting a clearer notation, because the branching-process remains implicit, the explicit source-code thus appears to be more message-oriented:

```
try
{ open;
  load;
  close;
}
catch(OpenException e){ "Cannot open";
catch(LoadException e){ "Cannot load";
catch(CloseException e){ "Cannot close";
finally{}
```

The handler-code becomes easily localizable (as can be seen above) and the original code is freed from distracting if-clauses that divert from the main thread of program flow.

Later, exceptions are implemented as classes to be able to use their extensible class-hierarchy, therefore the parameter-form of the catch-clauses had been chosen. An entire hierarchy of exceptions can be handled by using only one try-catch-clause. The try-catch-clause here is used in its elementary form, later try-catch-clauses introduce an entire messaging-infrastructure, see page 279.

Examples: Throwing references of self-instantiated objects on page 283. Opening files on page 298.

The remaining throw-, synchronized- and return-statements are introduced later.

Handler: A name for code that has supportive character, often comparatively small, but which is a part of the program like a service program, often like a plug-in or a subroutine. For example, the catch-clause of a try-statement forms the frame for the code of an exception-handler. See especially → Protocol-Handler and → Content-Handler.

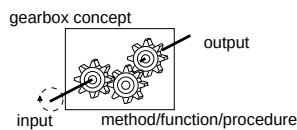
Exception: 1. A state in a program, that shouldn't occur under normal circumstances (division by zero, unexpected network shutdown), but is still recoverable within the program's context. Some languages, like Java, allow to add code that tries to recover the situation (for example, searches more information to circumvent the division-of-zero error or to search for the same information on other sources (hard-disk) than the network). Such recovering-code with auxiliary, supporting character is written into special exception-handling routines. For an example see page 283. Exceptions can be seen as error-related high-level relatives of → interrupts.

2. An object that usually contains the message about something that went wrong and re-routes the current thread to a piece of code that is supposed to handle the exception. In some worst cases the program-run even is terminated.

3. There are checked exceptions and unchecked exceptions in Java, see the text around page 280.

Structuring Code: Proceduralization

Structuring Code: Proceduralization — A recipe contains several commands for cooking a dish. A menu usually consists of several independent dishes and has to be cooked using several recipes. — A simple program just consisting of a sequence of commands, like those presented so far, resembles a recipe in this respect. Larger programs often resemble a menu, which may be structured like a collection of recipes. There, each recipe of the menu can be seen as a procedure, a semantic entity, that can be worked through individually when run in the proper context.

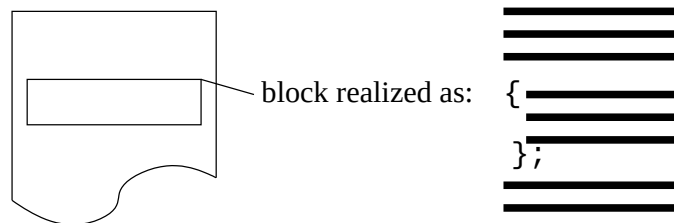


In a more mechanical interpretation the cogwheels given by expressions and statements may be bundled into a gearbox, that may take a special input and produces a special output. If input- and output-specifications are properly designed, then the inner mechanics may be changed or optimized independently. (This idea is also, rather abstractly, called encapsulation.) In the same way larger programs can be written as a collection of procedures to be called from

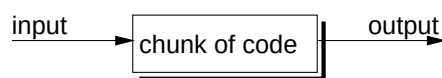
a main program, this is described below.

Concept: Methods are more than Named Blocks of Code

Concept: Methods are more than Named Blocks of Code — A chunk of declarations, expressions, statements and control structures inside a piece of source-code can be marked as a unit by declaring it a block. Thus, a so-called block or a so-called group are unnamed chunks of code.



If such a chunk of expressions, statements and control structures is multiply used in an application, or if that chunk has a problem-specific meaning, then it may be given a name. (Just like the typed and binarily-coded data is given a problem-related name in the process of data-declaration, leading to the notion of a variable on page 40 and on page 73.)



implemented as

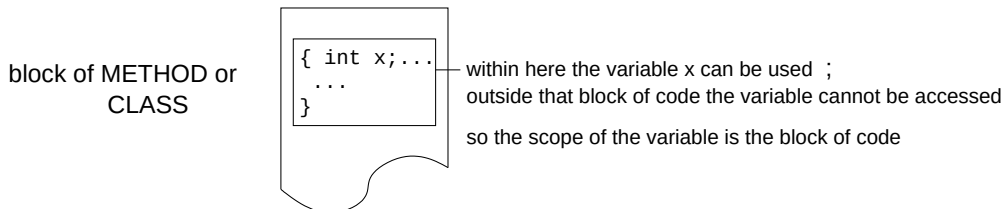
```
OutputType CodeChunkName(InputTypes inputnames)
{
    chunk of code
    return outputName;
};
```

to be called (multiply, if needed) from another part of the source-code by

```
outputVariable = CodeChunkName(InputValues);
```

The graphic above shows a typical implementation of the concept of a named block of source-code as it is found in several programming-languages. When defining the method, the specification of the input-types also is called parameter(-list); whereas the input-values of the method-call are called arguments. So there first comes the parameter-list when declaring or defining the method. After that, when calling the method, the call includes an argument-list of concrete values. Sometimes these notions are used interchangeably which can cause confusion; where possible, this text tries to clearly separate these notions.

As already hinted at in the introduction, another aspect proves to be useful: Isolating the proceedings inside the block from the outside. So that the only connection with the outside would be made by accepting input-arguments, and by returning a return-value. This would need variables to be defined locally and to refrain from changing variables declared outside of that block. Variables defined inside the block of code are to be taken valid only inside that block, thus introducing so-called local variables:



Then, in what sense are methods more than a named block of code? The **list of input-parameters**, and **local variables** make it possible to observe effects introduced later under the names: Scoping (where a variable is valid), recursion (when a method calls “itself”) and overloading (methods with the same name but differing parameter-lists). In Java, such specially defined named chunks of code are called methods. (In other languages such named chunks of code are called functions or procedures.)

For more on concepts of methods see the summary and outlook at the end of this chapter.

Java: Methods

Java: Methods — In Java, methods always are defined inside the block³² of a class-definition; that means methods are always declared in so-called class-bodies! An example gives the program below:

```
/* This piece of code does the same as the standard HelloWorld program.
   But the printing of the "Hello World!" string is encapsulated into a method. */
class HelloWorldMethod
{
    static void toEndUser( String s ) // method, output-routine, handler
    { System.out.println( s );
    }
    public static void main( String unused[] )
    { toEndUser( "Hello World!" );
    }
}
/*
Declaration, Definition of Methods:
--- The class contains two methods: The main()-method and the toEndUser()-method.
--- The main()-method takes several input-arguments indicated by [] of type String.
--- The toEndUser()-method takes only one input-argument, a variable of type String.
--- Both methods do not return any values and therefore are marked by the modifier 'void'.
Activation or Call of a method:
--- The method toEndUser() is called from inside the main()-method.
--- The main()-method is called when the end-user types
    "java HelloWorldMethod<RETURN>" into the terminal-window.
*/
/*
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization > javac HelloWorldMethod.java
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization > java HelloWorldMethod
Hello World!
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization >
*/
```

³²In Java, methods can only be declared as members of a class. Whereas in other programming-languages, methods may be declarable as sub-methods inside other methods!

For example, in a Java program/class there can be defined other methods beside the main()-method. But in Java there cannot be declared methods directly inside the main() -method; this can be achieved only by declaring an inner class (see page 157) that contains the desired method!

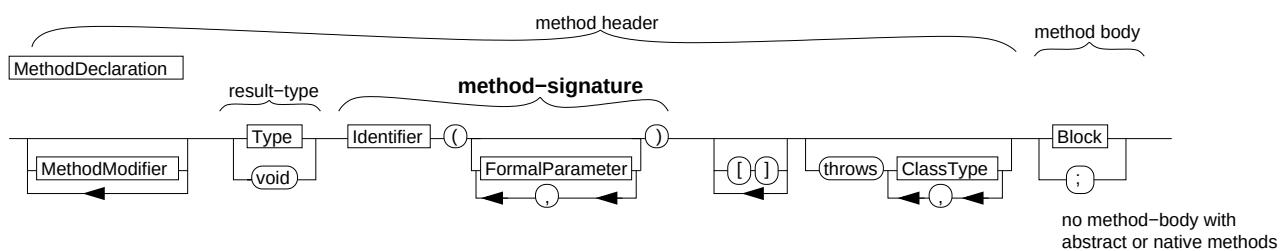
Java: Method-Declaration

Java: Method-Declaration — The following describes in a relatively general way how most of the methods are defined in Java (inside the block of a class!):

```
modifier(s) ReturnTypeNameOrVoid methodName( TypeName transferVariableName )
{
    TypeName localVariableName;
    StatementsAndControlStructures;
    return ExpressionOfReturnType ; // omitted if return-type of method is 'void'
};
```

In Java, variables defined inside a method-block (**local variables**) have to be assigned a value, otherwise the Java compiler complains. (Variables defined as members of classes are given default values, if not assigned explicitly.) The notion of a **non-local variable** denotes just a variable defined outside the current block of source-code; a non-local variable may be a local variable defined within the calling block of source-code. Locality as well as non-locality of variables *always refers to a given block of source-code!* assign local variables

So-called modifiers like public or static will be discussed later (see pages 171 and 175). The full description of a method-declaration is furnished below:



MethodDeclaration := *MethodHeader* *MethodBody*

MethodHeader := { *MethodModifier* } *ResultType* *MethodDeclarator* [*Throws*]

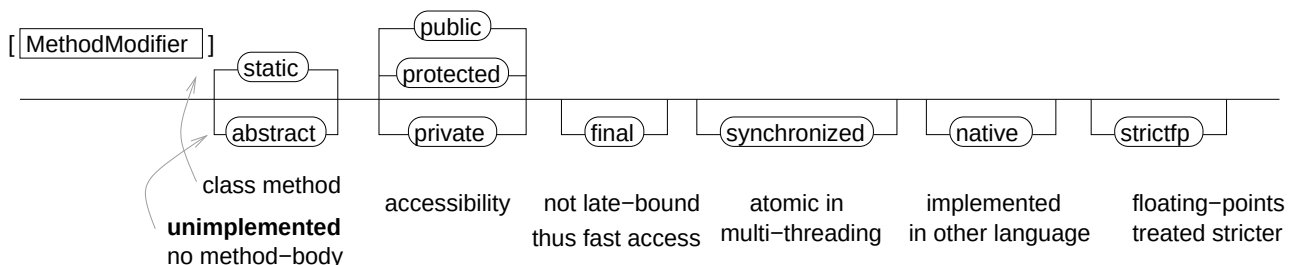
MethodBody := *Block* | ;

ResultType := *Type* | void

MethodDeclarator := *Identifier* "(" [*FormalParameter* { , *FormalParameter* }] ")"

Throws := throws *ClassType* { , *ClassType* } — Block 105, Type 76, Identifier 81, FormalParameter 118, ClassType 146

(some similarities are found in the ConstructorDeclaration on page 150 and the AbstractMethodDeclaration on page 184 and ClassDeclaration on page 146) — Name and parameter-list of a method taken together (the MethodDeclarator) is also called **signature of the method**. — The result-type describes the type of the variable returned by the method³³. Methods with a ResultType void do not return any value. Void-methods just manipulate data which has to be accessible using non-local identifiers; or void-methods perform output tasks, like printing out some text on the screen or the printer. signature of a method result-type: void



MethodModifier := ONE OF public protected private abstract static final synchronized native strictfp — for more about controlling scope with the public, protected, private-modifiers consider page 175.

Modifiers are keywords prefixed with an intermitting space “ ” to (method-) declarations³⁴. A range of, Modifier to be introduced, modifiers can be seen in the graphic above. For example, a practical trick for making method-calls run faster, is to use the final-modifier when declaring a method; this binds the method-final modifier

³³If the variable happens to be of reference-type (page 144), then a reference to the variable is returned, not the value of the variable!

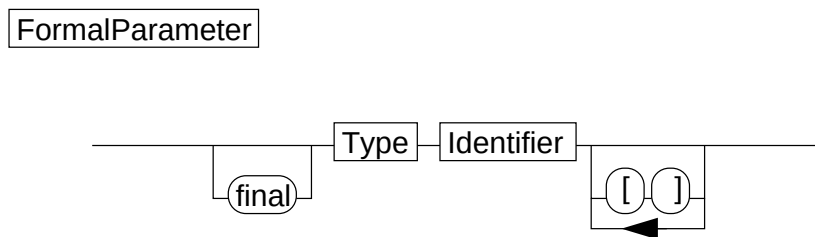
³⁴Qualifiers are class- or package-names prefixed with an intermitting period (.) to variable-names or other class-names.

name to byte-code at compile-time, not time-consumingly at run-time³⁵. For more see the remark on early- and late-binding on page 168. The modifiers `final`, `private`, `protected`, `public`, `abstract` and `static` get their meaning from the class-context of the Java programming-language which allows to realize an object-oriented structuring of source-code. Static methods, also called class-methods, are conceptionally introduced on page 133 and their Java realization can be found on page 148 and page 154. Abstract or “not-yet-implemented” methods are conceptualized on page 134 and are introduced as part of the Java-language on page 183. Accessibility of identifiers in general, also those of methods, is discussed later beginning with page 175.

The remaining three modifiers cover very different topics: When working with multiple threads — that means with multiple flows of statement running (quasi-)parallel — marking methods as `synchronized` makes them uninfluenceable by other threads (beginning with page 216). Native methods are not written in Java, but are written in another programming-language. Only if the native method is implemented for *all* possible native-platforms, then the Java program preserves its portability. Native methods are not treated in this text! Methods which are working with expressions of primitive-type `float` and `double` may be marked `strictfp` (short for strict floating point or FP-strictness) to assure a standardized treatment of floating-point values especially in situations of overflows³⁶.

strictfp
modifier

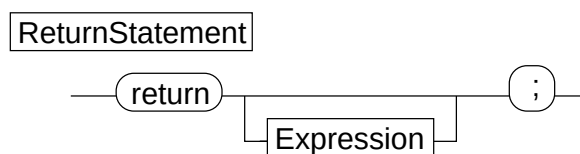
The item *FormalParameter* essentially represents the declaration of the parameter-list:



FormalParameter := [`final`] *Type* *Identifier* { "[" "]" } — *Type* 76, *Identifier* 81 (*FormalParameter* is used also in *CatchClause* on page 283.)

Method-parameters differ from a *LocalVariableDeclaration* (page 121) in that, with parameters the declaration and value come from the outside of the method's block. `Final` parameters take arguments that cannot be changed within the method's block!

The `return`-statement explicitly returns control from the current method (or from current constructor) to the next statement after the method-call statement. The optional expression contains the value to be returned with the method-call. In non-void methods the `return`-statement has to furnish an expression of the type which has been declared as return-type in the method-definition.



ReturnStatement := `return` [*Expression*] ; — *Expression* 90

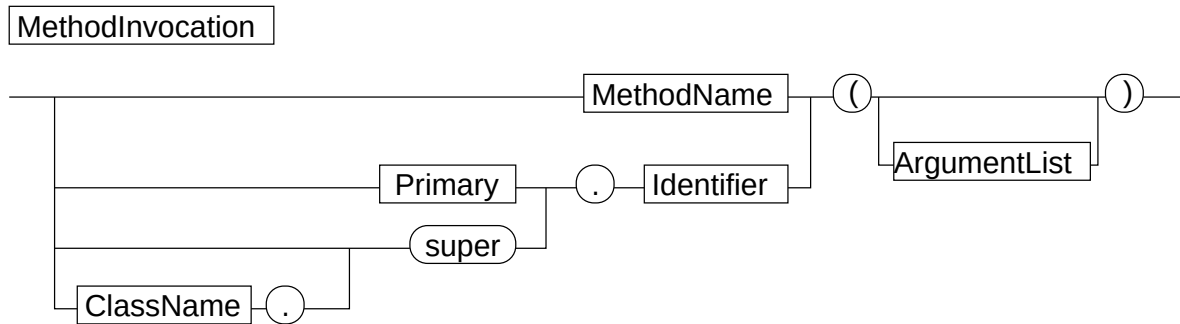
Multiple values may be returned as an aggregate, like an array or the like, see the chapter on aggregation beginning with page 233.

Java: Method-Call

Java: Method-Call — Finally, after the mechanics of positioning (inside a class) and defining a Java-method have been presented above, it remains to tell how a method can be called in Java. This is done by a method-call-expression, see also the modified HelloWorld-program above:

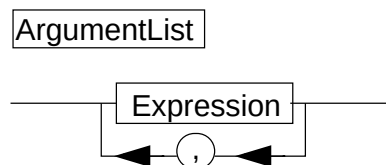
³⁵There occurs a disadvantage of course: If used in extended classes, such methods cannot be overridden. `Final` methods have to remain constant in the service they provide. Of course, if there is no need for the flexibility of contextually redefining a method, then methods can be readily declared `final`!

³⁶For more on the `strictfp` modifier see also the TechTip of April 10th, 2001.



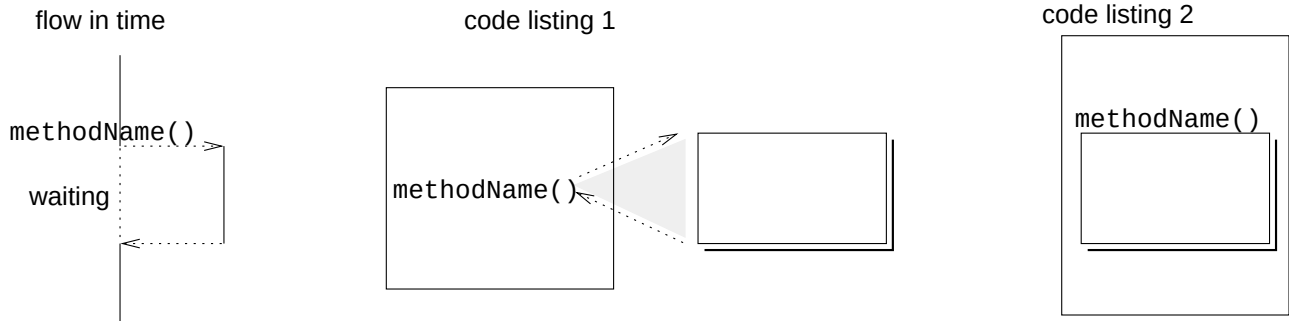
MethodInvocation := *MethodName* "(" [*ArgumentList*] ")" | *Primary* . *Identifier* "(" [*ArgumentList*] ")" | *super* . *Identifier* "(" [*ArgumentList*] ")" | *ClassName* . *super* . *Identifier* "(" [*ArgumentList*] ")" — *MethodName* 82, *Primary* 91, *ClassName* = *ClassType* 146 , *Identifier* 81, *ArgumentList* see below.

Above, in the simple case of the HelloWorldMethod-program an explicit method-call has been issued: Just by giving the method-name and letting follow a pair of parentheses containing an argument-list of one single String-object: `toEndUser( "Hello World!")`;



ArgumentList := *Expression* { , *Expression* } — *Expression* 90

Usually³⁷ method-calls let the calling code wait until the method's code has been worked through. Methods of that kind are called **blocking methods**. This blocking by a method has been visualized below in three different ways:



These mechanics of defining and calling a method in Java are accompanied by several other features (shadowing local variables, overloading, recursion) to be introduced in the remaining sections. Concrete examples of method-declarations and method-call expressions can be found in almost every piece of larger source-code in this text.

Concept: Method-Call Arguments — Transfer By-Value or By-Reference

Concept: Method-Call Arguments — Transfer By-Value or By-Reference — When a variable-name occurs in the argument-list of a method-call, what happens to the variable? Is the input copied and the copy manipulated, or is the input-variable only referenced? Here, referencing a variable means that the method's code can directly manipulate the original variable within the calling code. Both cases can be found:

When the method's input-argument is taken, and its value is copied into a local (parameter-)variable, then a change of the argument's value within the method's implementation-block doesn't affect the original

passing
arguments
by-value

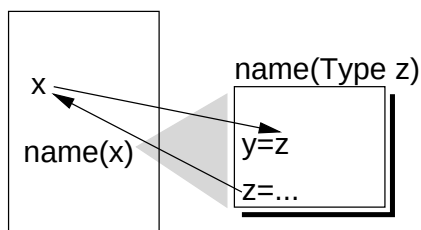
³⁷As introduced later, so-called non-blocking methods spawn their own thread, thus the calling-code need not wait until the method has finished its work.

variable outside. This copying input-procedure is called **passing the variable "by-value"** (In Java, VARIABLES OF PRIMITIVE-TYPE as arguments of a method-call are PASSED BY-VALUE.).

On the other side, the method's input-argument may be just a pointer which is used inside the method to access the original data. Then, the code of the method may change the data of the variable outside the method. If that's the case, then **the passing of the input-argument is called to be done "by-reference"** (In Java, VARIABLES OF CLASS-, INTERFACE- AND ARRAY-TYPE as arguments of a method-call are PASSED BY-REFERENCE.).

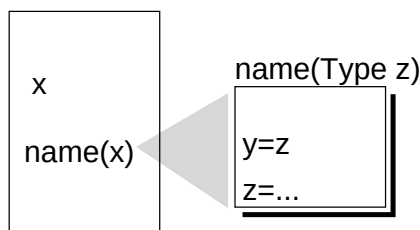
passing
arguments
by-reference

Call by reference



here the x value
is referenced
changes to the argument
affect the caller

Call by value



here the x-value
is copied
all changes remain local
if not officially returned as output

Of course, any values changed by-reference violate the idea of the method's input-data given by arguments and the method's output-data given by a return-statement, thus mellowing the idea of encapsulation of the procedures within the method.

In the following example the variable of class-type `java.lang.StringBuffer` is anticipated to illustrate the differing effects of calls by-value and calls by-reference in Java.

```

/*
--- A method-call, with arguments given by-value, copies the data, then any
    changes to that data, inside the method, are confined to the method-block.
    (The character-value changes to 'x' only within the method.)
--- A method-call, with arguments given by-reference,
    makes the method change the original data.
    (The StringBuffer-variable changes its value also outside the method-block.)
*/
class CallsByValueByReference
{
    public static void main( String unused[] )
    {
        char byValue = 'v';
        StringBuffer byReference = new StringBuffer( "referenced" );
        // Thus, before the method-call: byValue='v' and byReference="referenced"
        theMethod( byValue, byReference );
        System.out.println( "data AFTER method-call, by-value: " + byValue +
                           " and by-reference: " + byReference );
    }

    static void theMethod( char ch, StringBuffer s )
    {
        ch = 'x';
        s.append(" and thus changed.");
        System.out.println( "data during method-call, by-value: " + ch +
                           " and by-reference: " + s );
    }
}

/*
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization > javac CallsByValueByReference.java
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization > java CallsByValueByReference
data during method-call, by-value: x and by-reference: referenced and thus changed.
data AFTER method-call, by-value: v and by-reference: referenced and thus changed.
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization >
*/

```

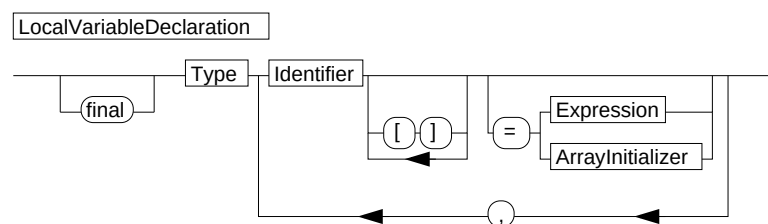
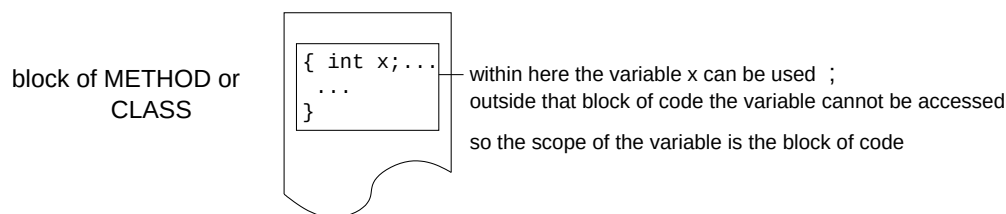
SUMMARY: Remember that read-only access of arguments can be guaranteed within a method-definition by declaring the associated parameter `final`. Otherwise, input-arguments of Java-methods may be

changed inside the implementation-block of the method, this has different consequences for variables of primitive- or reference-type: (PRIMITIVE-TYPE) The method copies the values of arguments of primitive-type, thus subsequent changes to those values are valid only within the block of the method. This makes any argument of primitive-type a sort of local variable. Input-arguments of primitive-type, whose values are to be changed, have to be returned explicitly by (re-)defining the method using a `return`-statement, which returns the changed input-arguments. (REFERENCE-TYPE) If the input-argument is of reference-type, then subsequent changes of that variable within the implementation-block of the method affect the variable outside (This aspect is revisited on page 143, as reference-types are introduced).

OUTLOOK: If an input-argument of reference-type has to be treated as a local variable, then it has to be cloned (page 191) and its copy has to be referenced by the code of the method-block³⁸.

Java: Local Variables - Scoping and Shadowing 1

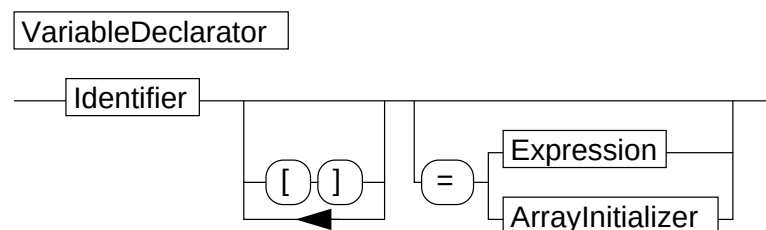
 Java: Local Variables - Scoping and Shadowing 1 — As already mentioned in the section about concepts some page before, methods allow the Java programmer to declare variables or identifiers within their block of code: Those so-called local variables are only valid and accessible within that block of code. **METHOD-ARGUMENTS OF PRIMITIVE-TYPE BECOME IMPLICIT LOCAL VARIABLES**, but the DECLARATION OF LOCAL VARIABLES ALSO CAN BE DONE EXPLICITLY:



LocalVariableDeclaration := [final] Type VariableDeclarator { , VariableDeclarator }

Local Variable: 1. A variable defined inside a method. Such a variable is only accessible from within the method's block.
2. Java, see 1. and add: Such a variable becomes accessible (→ scope) inside the method's block, from the point of its definition to the end of the method's block. Unless assignment is done implicitly — by being a variable of the method's parameter-list —, local variables of primitive-type have to be explicitly assigned a value; otherwise, beginning with Java Version 1.2, the Java compiler should complain. Initialize the local variables!

Variables defined in class- and interface-bodies are called either member-variables or fields. On the level of class-definition, the corresponding notion to a local variable can be seen as an inner class (page 157).



VariableDeclarator := Identifier { "[" "]" } [= VariableInitializer] (used in FieldDeclaration on page 147)

VariableInitializer := Expression | ArrayInitializer — Type 76, Identifier 81, Expression 90, ArrayInitializer 236 (VariableInitializer is used in ArrayInitializer)

³⁸Argument-passing by-value and by-reference has been considered also in the TechTip of Oct. 9th 2001.

So the variable-declarations inside a method-block are valid only *inside* the same block of code, the same is true for the arguments of a method. Local variables only can be given one modifier, the `final` modifier, which makes them a constant.

```
methodBlock{final double PI = 3.1415;}; // OR
methodBlock{final double PI; PI = 3.1415;}; // blank-final, assignable once
```

Final local variables are constants. If the `final` variable is not initialized immediately on declaration, it can be initialized exactly once and then cannot be changed any more. Until its first initialization such a `final` variable is called “**blank-final**”.

Scope



The **scope of an identifier** is the part of the source-code in which the use of the identifier (alone³⁹) suffices for retrieving data. For example, local variables have a scope limited to the block in which they were defined.

The concept of scope extends to identifiers in general, class-names, method-names and file-names, as can be seen later. Again, giving the scope of a variable tells the programmer in what part of the source-code the variable-name can be used to access the variable's data.

Remember that the declaration of LOCAL VARIABLES IN METHOD-BLOCKS, has to be done BEFORE working with the variables⁴⁰:

```
// block of some method (or static- or instance-initializer)
{ ...
  l=1; // returns an compile-time error: Undefined Variable: l
  ...
  long l;
  ...
}
```

Otherwise, in Java, every DECLARATION (within a class- or interface-block) IS VALID THROUGHOUT THE BLOCK IN WHICH THE DECLARATION OCCURS, see also page 155! The necessity⁴¹ of this feature is demonstrated strikingly in the callback-program on page 127.

It can be called good programming practice with methods to strive to work only with local variables and with explicitly transferred variables. And avoiding to manipulate non-local variables from a method (transfer by-reference). This practice is called information-hiding and is part of encapsulation of the object's internal functioning. Thus minimizing points of contact in code-design gives the code a better, clearer structure, which makes writing or re-writing or pure application of the code easier⁴².

Scope: 1. The scope of an identifier gives the area in which the sole identifier, without any additional qualification, can be used to access its data. The locations inside a network, a computer-system or a piece of source-code where an identifier is known. The scope of a → Uniform Resource Location (URL) are all protocols and computers that are part of the Internet.

2. (Java) See 1. and add: In Java, the scope of most identifiers is given by the class-block in which they have been declared. Identifiers, declared in a method-block — making them local variables —, have a scope given by the remaining block (the same is true with identifiers in blocks of static- or instance-initializers). See also the graphic below. Additionally, so-called modifiers (`private`, `protected`) give the Java programmer a limited control over changing the identifier's scope (page 175).

(3) Later, the notion of “scope” will be supplemented by that of an “access-area”, see also → Accessibility (page 172). Whereas a → name-space represents a collection of identifiers usually having identical scope.

³⁹Alone means, unqualified by some sur-name, for example: `String` instead of `java.lang.String`.

⁴⁰Anticipatingly, the same applies to local variables defined in blocks of static- and instance-initializers (pages 152 and 152, respectively).

⁴¹This makes the so-called prototyping (C,C++) or forward declarations (Pascal) unnecessary in Java.

⁴²Some languages (Pascal) make it possible to access a method's parameter explicitly globally or non-locally by declaring the parameter to be “variable”. Then, accessing the object globally gives the less time- and memory-costing alternative, but this makes the function not self-contained, which in turn can lead to unwanted complications! — Using “variable” parameters is called parameter transfer by-reference. Like in Java, the changed value of reference-type (in Pascal declared “variable”) is “returned” automatically, because the access by using the reference needed no copying of the values; the changes were written directly into the original data. — Contrary to this, the standard way of parameter transfer in Pascal is called transfer by-value. In practice this often requires the system to make a copy of the transferred object, which can take time and memory. This disadvantage in turn favors the access-by-reference in Pascal. — With Java or Pascal, this means either resource-consuming clarity (transfer by-value, explicit returning) or resource-saving direct access (access-by-reference, no encapsulation).

```

class className
{
    ...
    memberVariable
    ...
    methodDeclaration(parameterList)
    {
        ...
        localVariable
        ...
    }
    ...
}

```

scope of member-variable

scope of local-variable

More about scoping and accessibility in Java can be found beginning with page 171.

Name-Space: A name-space is given by a set or collection of names or identifiers, usually having the same scope, for accessing → resources. A name-space can be flat or hierarchical. A → scope is the area of validity of a single name. So, the notion “name-space” actually may specify two things:

1. A set of names or a common method to generate names. (Examples: The way to use and build absolute path-names in hierarchical file-systems, the area of validity extends over the entire platform (page 47). The name-space given by Uniform Resource Locators (URLs), which expands the area of validity over the entire network of computers. (page 364))
2. An area of validity of a given set of names. (Examples: The name-space defined by a package as introduced on page 239. The name-space given by class-loaders, described as a package-equivalent, beginning with page 436.)

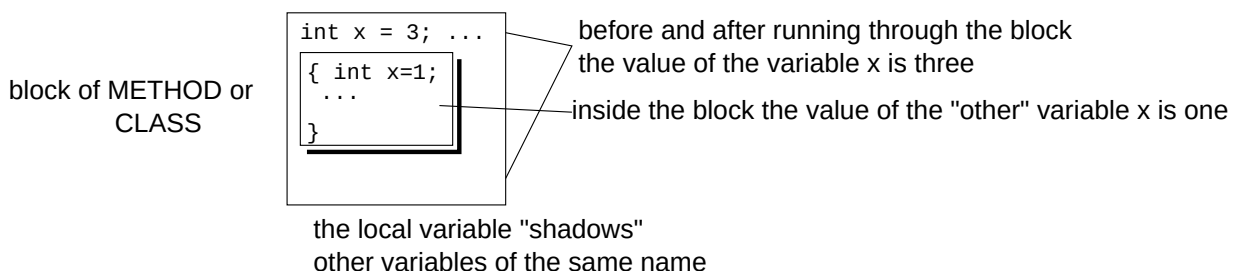


Examples for name-spaces are: The block of a method hosts the name-space of all the method's parameters of primitive-type and of all local variables. The Hierarchical File System (a hierarchical name-space, page 46), the package notation of loaded classes (a flat name-space that looks hierarchical, page 239), the Domain Name System of the Internet, page 360.

Shadowing Variables with Methods

Shadowing Variables with Methods — What happens if a variable, within the source-code that issues a method-call, has the same name as a local variable inside the method's block? (The original variable is also called non-local variable with respect to the block of the method.)

non-local
variable



Non-local variables, that have the same name as local ones, cannot be accessed with that name from the local block. This property of the Java language can be reformulated: If the identifiers of local and non-local variables, with respect to some method- or class-block are the same, then **the local variable shadows the non-local variable**. Shadowing becomes useful when letting a method call “itself” as presented in the next section; more scoping and shadowing can be seen with classes and objects (page 175).

local variable
may shadow a
non-local one

Nevertheless a shadowed non-local variable can be accessed by qualifying its name with the class-name of the class in which it has been declared. The elementary form of un-shadowing an identifier is presented in the following program:

```

public class ShadowStatic
{ static int x=3;

    static void theMethod( int arg )
    { int x=1;
      System.out.println( "non-local x = " + ShadowStatic.x );
                        // qualification with class-name
      System.out.println( "local x = " + x );
    }

    public static void main( String unused[] )
    {
        theMethod(x);
    } }

/* output: (same if method is declared and called without an argument)
           (same if method-block substitutes the block of the main()-method)
*/

/*
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization > javac ShadowStatic.java
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization > java ShadowStatic
non-local x = 3
local x = 1
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization >
*/

```

Above, the statements about blocks and scope refer **ONLY TO BLOCKS OF METHODS OR BLOCKS OF CLASSES**. Simple blocks of source-code {...}, even if they follow if-, switch-, while-, for-, do-keywords, MAY ALLOW A SORT OF LOCAL VARIABLE-DECLARATION BUT THOSE VARIABLES HAVE NO SHADOWING CAPABILITIES and thus exhibit the following peculiarities: First:

```
{ { int i; /* i valid only in this block, not outside */ } }
```

— Second:

```
{ int i; { int i; /* gives an "already defined" compilation-error */ } }.
```

The compiler-error has the form: `i is already defined in main(java.lang.String[]) ... 1 error`. Declarations in if-, switch-, while-, do- and for-statements are local with respect to the enclosing method (or class). So re-declaring a variable in one of those (if, switch, while, do and for) blocks has no shadowing effect but instead the Java compiler complains.

OUTLOOK: The class-notion allows to introduce a kind of local type: Local- or inner-classes, on page 157. Scoping for classes — telling where an identifier is valid — is treated extensively in the section beginning on page 171. More on shadowing in relation to class-extension can be found on page 168.

Concept and Java: Overloading, Recursion, Callback

Concept and Java: Overloading, Recursion, Callback — The previous sections should have shown that a method is more than a named block of code: Arguments can be seen as being transferred by copying (primitive-types) or by handing out a reference (class-, interface- and array-types). Variables can be defined locally with respect to the method-block, which ensues a shadowing phenomenon with non-local variables of the same name. These properties of methods are the basis for other useful effects that are introduced below: Overloading, recursion and callback.

Overloading Methods

Overloading Methods — THE JAVA COMPILER IDENTIFIES METHODS BY CONSIDERING THEIR **NAME AND THEIR PARAMETER-LIST**. Name and parameter-list are called the **signature** of the method. Therefore, one can define different functions with the same name, but different parameter-lists. This is called **overloading a method**. The return-type is not part of the method's signature and therefore irrelevant in that context. So, different methods can have the same name but differing parameter-lists (differing means different types of arguments or a different number of arguments). Different methods with the same name, the same parameter-list and different return-types usually cannot coexist.

signature of a
method

overloading a
method


```

    // prints a string:
    static void print( String s ){ System.out.print( s ); }
    //
    // prints an array of strings:
    static void print( String [] s )
    { for(int i=0; i<s.length; i++) System.out.print( s[i] ); }
    //
    // this prints everything it gets except primitive-types:
    static void print( Object o ){ System.out.print( o ); }
    /* The argument o takes almost any variable of reference-type
       and prints the available information. */
    //
    // This collection of print()-methods prints any information it can get.

```

The methods, as defined above, are clearly overloaded, because they all have the same name and different parameter-lists.

```

/* Three methods with the same name are defined below,
   their signature differs only in the parameter-list.
*/
public class Overloading
{
    static int method( int x ){ return x; }
    static int method( float x ){ return (int)x; }
    static int method( double x ){ return (int)x; }

    public static void main( String unused[] )
    {
        System.out.println( method(1) == method(1.1) );
    } }
/* The first and the third methods are used,
   because literals like 1.1 by default are taken to be a 'double'
   and literals like 1 are taken to be an 'int' by default.
*/
// The print-out yields 'true', because here casting means truncating.
/*
marycosway@linux:~/JavaCodeFiles/Basic1/SCProceduralization > javac Overloading.java
marycosway@linux:~/JavaCodeFiles/Basic1/SCProceduralization > java Overloading
true
marycosway@linux:~/JavaCodeFiles/Basic1/SCProceduralization >
*/

```

Overloading Methods: This means to declare a method with the name of an already declared method, but with a differing parameter-list. Thus the Java compiler still can discern which method is meant on processing method-calls. Overloaded methods are different methods defined having the same name but being given differing parameter-lists.

Another example of overloading methods can be seen when considering Container-classes for programming single-user-interfaces: In the list on page 553, can be found multiple methods of the name `repaint()`, each defined with a different parameter-list. In a similar list on page 556, occur multiple methods named `add( . . . )`, each with a different list of parameters!

Recursion of Method-Calls

Recursion of Method-Calls — THE NAME OF THE METHOD IS A VALID IDENTIFIER IN THE BLOCK OF THE SAME METHOD! Therefore, inside the method-definition – that means inside the method-block – the same method may be called. Actually each of these calls produces a new resource-consuming internal context of this method, but on the level of the source-code, this looks just as if the method can call “itself”.

Successive method-calls of this kind have to terminate at some level! Otherwise, if the calling is not terminated by a condition, then the successive calling of the method would continue stacking internal context on internal context, until some buffer overflowed.

The calling of a method within its own method-block is called **recursion**.

recursion

Recursion: 1. In mathematics: Inside the definition of a function is used the function itself, usually with an added termination condition. For example the function $f(n) := n \cdot (n - 1) \cdot (n - 2) \cdot (n - 3) \cdot \dots \cdot 3 \cdot 2 \cdot 1$ can be described recursively by $f(n) = n \cdot f(n - 1)$ plus the termination condition $f(1) = 1$.

2. In computing: Same as in mathematics, inside a method-declaration is used a call of the method itself! A termination-condition avoids a buffer-overflow or an `OutOfMemory-error`.

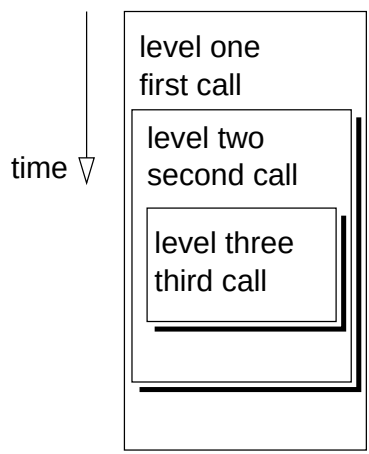
Usually, recursion provides an elegant way to avoid loop construction, but recursion may not always seem easily readable. A recursive method can be immediately identified when finding its own method-call within its defining block of code:

```

static void recurseTo(int depth)
{ System.out.println( "Entered level " + depth );
  if( depth < 3 ) recurseTo(depth++); // termination and recursion
  System.out.println( "Leaving level " + depth );
}
// initial call
recurseTo(1);

```

Observe that the variable “depth” is incremented only at one point in the method; it is never decremented. That is not necessary, because the variable “depth”, as a member of the method’s parameter-list, is a local variable shadowing non-local variables with the same name of some previously called level. Thus, a copy of the respective local variable persists on each level and when returning from a recursion-level the respective value of the local variable is reused:



Below is presented a fully functional piece of source-code with a recursion. The recursive method presented above can be easily compared to its counterpart below to identify the essentialities of such a simple recursion: Method-call within method-declaration, termination procedure and initial call and especially here: Incrementation of counter on call or in termination-condition.

```

public class Recursing
{
    static void recurse( byte x )
    {
        if( x < 3 ) x++; else return;           // termination
        System.out.println( "-> to level " + x );
        recurse( x );                          // calling (another copy of) "itself"
        System.out.println( "<- from level " + x );
    }
    public static void main( String unused[] )
    { recurse((byte)1);                        // first call
    }
}

/*
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization > javac Recursing.java
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization > java Recursing
-> to level 2
-> to level 3
<- from level 3
<- from level 2
marycosway@linux:~/JavaCodeFiles/Basics1/SCProceduralization >
*/

```

OUTLOOK: A very different kind of recursion is hinted at on page 309 in the program listing. There the content of the following chapter about defining types is used to define types recursively, leading to tree-structures.

OUTLOOK: Later, a variant of recursion is used with classes that represent screen-windows (page 545): They can be given a static main()-method - and thus be made a program - and can instantiate themselves from that main()-method. This means the class instantiates itself from within a static method; this is possible because the name of the class is a valid identifier within static blocks of code.

Callback of Methods

Callback of Methods — Here is given a case where two methods are calling each other. (This demonstrates that the method identifiers are known in the entire block of the class, independently of the order of the method-declaration⁴³!) A callback-method does not return via the `return`-command or by reaching the end of the method-block, but this kind of method returns by calling another method:

```
public class Callback
{
    static void ping( int x )
    {
        System.out.println( "ping " + x );
        if( x <= 0 ) return;                // termination
        pong( --x );                        // call
    }
    static void pong( int x )
    {
        System.out.println( "pong " + x );
        ping( --x );                        // callback
    } // return NOT used here and method leaves only by calling ping()!

    public static void main( String unused[] ){ ping(5); } // first call
}
// What happens if ping(x) or/and pong(x--) are used and why?
// Answer: Non-termination, due to non-effective change of counter!

/*
marycosway@linux:~/JavaCodeFiles/Basic1/SCProceduralization > javac Callback.java
marycosway@linux:~/JavaCodeFiles/Basic1/SCProceduralization > java Callback
ping 5
pong 4
ping 3
pong 2
ping 1
pong 0
ping -1
marycosway@linux:~/JavaCodeFiles/Basic1/SCProceduralization >
*/
```

Here also has been presented an example of the effectiveness of pre-deprecation-operators: In the case of `pong(x--)` and `ping(x)` the callback does not terminate, because the method-call with post-deprecating the variable `x` can be translated to `pong(x); x=x-1;`, that means: No effective deprecation in the *called* code!

Concept: Triggerable-Methods Or Who calls the main() -Method of an Application?

Concept: Triggerable-Methods Or Who calls the main()-Method of an Application? — The `main()`-method receives its call within the Java Runtime Environment (JRE) by an external trigger, namely the command “`java nameOfByteCodeFile`” which a user enters into a terminal-window. A call from inside the Java code is allowed, but discouraged. Nevertheless, a Java application⁴⁴ can be integrated into another application just by calling its static `main()`-method! (page 630)

OUTLOOK: Later will be encountered various service-methods of that kind:

The `java.lang.Object.finalize()`-method is called automatically by the Java Runtime Environment (JRE) before the object is erased from the computer’s Random Access Memory. This method is introduced in the next chapter on page 152.

The `read()`- or `write()`-methods of wrapped streams have to be defined, but then usually are called by their counterpart-methods of the wrapping streams (see the sections around the program on page 304).

When having been properly registered, the `actionPerformed()`-method is called automatically when mouse-movements, received by the native operating-system, trigger the Java Runtime Environment (JRE) into “throwing” event-objects, for more see page 544.

⁴³If one function calls another one, and the called-function calls back the first function, then some languages (Pascal) need a previously given hint about the existence of the function that is called back. Furnishing this information is called prototyping there (in short: some programming-languages need prototyping to allow cross-calling of functions, JAVA DOES NOT NEED PROTOTYPING).

⁴⁴Java application means: Inside the source-code has been defined a `main()`-method. A tool like the `java`-command can start a Java Runtime Environment (JRE) and thereafter automatically triggers a call of that `main()`-method.

The `paint()`-method of the `java.awt`-package, and within the `javax.swing`-package the `paintComponent()`-method, react to changes on the desktop triggered by mouse-movements or keyboard-inputs (or to the programmer's call of the `repaint()`-method). For more see page 576.

The `run()`-method associated to a `Thread`-object has its code processed within a thread of its own, only when triggered by a programmer-issued call of the `start()`-method of the `Thread`-object. A direct call `run();` produces no thread! See the graphic past the one on page 205.

The `service()`-method of a servlet responds to calls channeled from outside its server-environment, for more see page 407.

The `check...()`-methods of the `java.lang.SecurityManager` are called by the Java Runtime Environment (JRE) to get directives for accessing system resources (page 462). The same is true for the `checkPermission()`-method of the `java.security.AccessController` (page 453).

Later, next to triggerable methods, will be introduced streams (page 295) and sockets (page 273) for triggering a program reaction – from outside the program or even from outside the Java Runtime Environment.

Triggerable Method: Any method can become a **triggerable method**. For a programmer, triggerable methods are called indirectly or implicitly from some other part of the runtime-environment. Usually a programmer is free to also call a triggerable method directly, but in most cases this approach has been disadvised. Triggering a method-call means that this method is not called from within the current piece of source-code. A triggerable method usually reacts to actions that happen outside the program domain: The “java”-command, after starting an entire Java Runtime Environment (JRE), activates the `main()`-method. – Before an object inside the Java Runtime Environment is erased from the computer's Random Access Memory, the Java Runtime Environment (JRE) calls its `finalize()`-method. – A mouse-movement, or a pressed key of the keyboard, trigger the calls of associated triggerable → handler-methods.

Triggerable methods may also be called “event-activated” methods, when an external event triggers the method-call; for example when a mouse-movement causes a `paint()`-call within the Java Runtime Environment. But sometimes, triggering may just mean invoking a simple, delegated, relayed method-calling-chain; for example when the programmer invokes the method `repaint()`, thereby filing a paint-request in the event-queue of the event-dispatch-thread, which eventually lets the Java Runtime Environment call the related `paint()`-method.

Summary and Outlook: Proceduralization

Summary and Outlook: Proceduralization — Expressions are high-level programming constructs that return a value. Non-void methods return values too. — Blocks of code begin and end with curly braces; and variables declared within such a block are only valid within that block. — Methods can be seen as named code-blocks with the ability to define local variables. These local variables can shadow variables of the same name which exist outside the method-declaration-block (non-local variables).

The following list of features makes it possible to introduce concepts like recursion, overloading and callback: Local variables, the validity of the method-name within the defining block of the method, the fact that methods are discerned by their signatures (= name and parameter-list), and the fact that method-declarations are valid within the entire block where they are declared, independently of the place inside that block where they have been declared!

Recursion (a method-call can be issued from its own defining source-code), **Overloading** (methods of the same name but differing parameter-lists can coexist) and **Callback** (methods that call their caller) rely on the aforementioned features of Java methods.

In Java, the `main()`-method characterizes a Java program that can be run autonomously within a Java Runtime Environment (JRE). The `main()`-method is a very special method in another respect: A `main()`-method is virtually never called from within a Java program; an end-user may use a terminal-window to enter the command `java nameOfByteCodeFile space optionalSpaceSeparatedStringArguments`, to trigger — among other things like creating the Java Runtime Environment — a call of the `main()`-method of the program given by `nameOfByteCodeFile`. This gives also the first example of such external triggering of method-calls.

(Blocking Method): Usually, calling a method lets the calling-code wait until the method has finished its task and returns; then the calling-code resumes its work. The **method is also said to be blocking**. All the methods encountered so-far were blocking methods.

OUTLOOK: Non-blocking methods are methods, which let the program continue running after their method-call has been issued. The calling-code does not wait for the method to return, see page 231 in the context of multithreading.

(blocking)
method

Blocking Method: The standard idea of calling a method implies that the calling program stops until the method's code is worked through and the method returns. Such a method is also called a **blocking method**, because the calling program has to wait until the method returns, then the calling-code proceeds with the code past the method-call. See also the visualization given by the graphic on page 119. — In contrast to this, a $\rightarrow$ non-blocking method is worked through in a thread of its own; here the calling-code does not wait for the method to return! See also page 231.

Generally the concept of named blocks of code with local variables has the advantages of:

Code Reuse: A control-structure that realizes looping reuses code. Calling chunks of code by name supports a kind of non-localized code reuse.

Proceduralization: Large pieces of code become more easily manageable, if they are inspectable piece-wise, like a menu composed of several recipes.

Information-Hiding: If the chunks of code have designated input- and output-variables and locally defined internal variables, then the inner workings of the code-chunks can be changed (optimized, developed) without changing the calling code. This concept is also called encapsulation and leads, when documented, to a so-called (Application) Programming Interface (page 43).

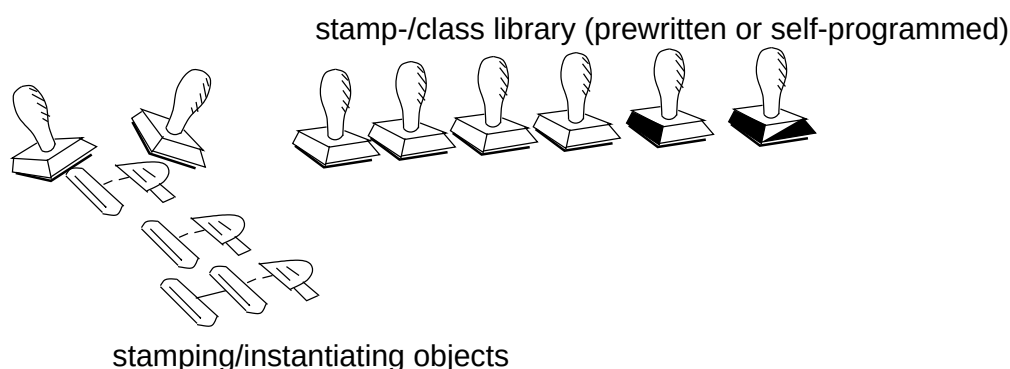
Structuring Code and Data: Object-Orientation

Structuring Code and Data: Object-Orientation — Methods and, if those methods properly encapsulate their functionality, the associated Application Programming Interfaces (APIs) are useful for structuring source-code. If properly designed, even large pieces of source-code, which have been structured with the means of proceduralization, do work and can be maintained. Nevertheless there are many (real-life) programming problems that are open to another way of structuring: Object-Oriented Structuring of source-code.

Historically the object/class-model emerged from trying to simulate physical real-life objects and their interactions⁴⁵. There, classes characterize sets of objects with common properties; this approach can be seen as the more general one.

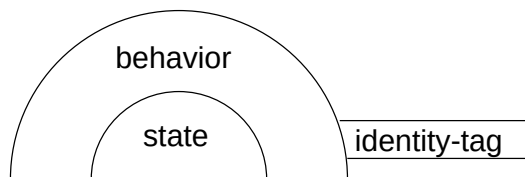
Another approach starts by developing programmer-defined types: A number may be thought as being characterized by its data AND the methods for its manipulation, like addition and subtraction, multiplication and division.

Since real-life objects often can be seen as displaying a considerable complexity, their relevant properties have to be isolated. That process of model-building is also called abstraction. Object-Orientation has a considerable conceptual overhead, but a readily approachable aspect can be shown with the stamp-analogy: Classes within object-oriented structuring are like stamps (which the programmer can carve or take from a library). With these classes the programmer can instantiate objects, much like the stamps can be used to produce several prints, into which it is possible to write data by hand. There even may be little stamp collections, which appropriately combined (“extended”) produce an new type of printed matter.



Concept: Object/Class-Model

Concept: Object/Class-Model — This section just introduces the central notion of the Object/Class-Model: An object is defined as an entity having a state, a behavior and an identity⁴⁶.



⁴⁵Nygaard, Kristen and O. J. Dahl. *The Development of the Simula Languages in History of Programming Languages*. Academic Press. New York, NY. 1981.

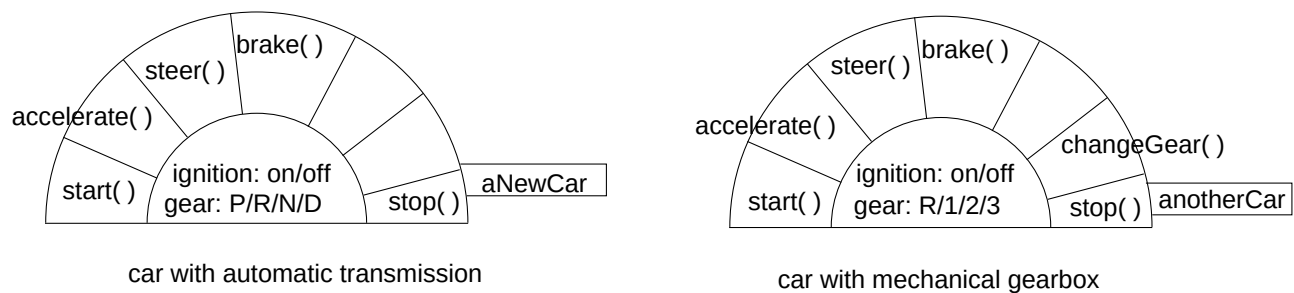
⁴⁶In object-oriented programming-languages the “state” of an object usually is given by variables and the “behavior” is given by methods. Though the identity often may be thought as being given by a variable-name, this may not be true when variable-names are pointers, that may change their pointing-direction! This is the case in Java; there, an identity-tag has to be retrieved by a special `hashCode()`-method, as introduced on page 161. CAUTION: Variable-names in Java do not fix the identity of an object!

In the graphical representation the identity-tag often is omitted, because of self-evidence. But the identity gives the means to discern two objects with the same state and behavior. Two objects with the same state and behavior are called equal (but they are not the same due to their different identities).

Later, the object-model is supplemented by an equally generally defined class-model, see page 140.

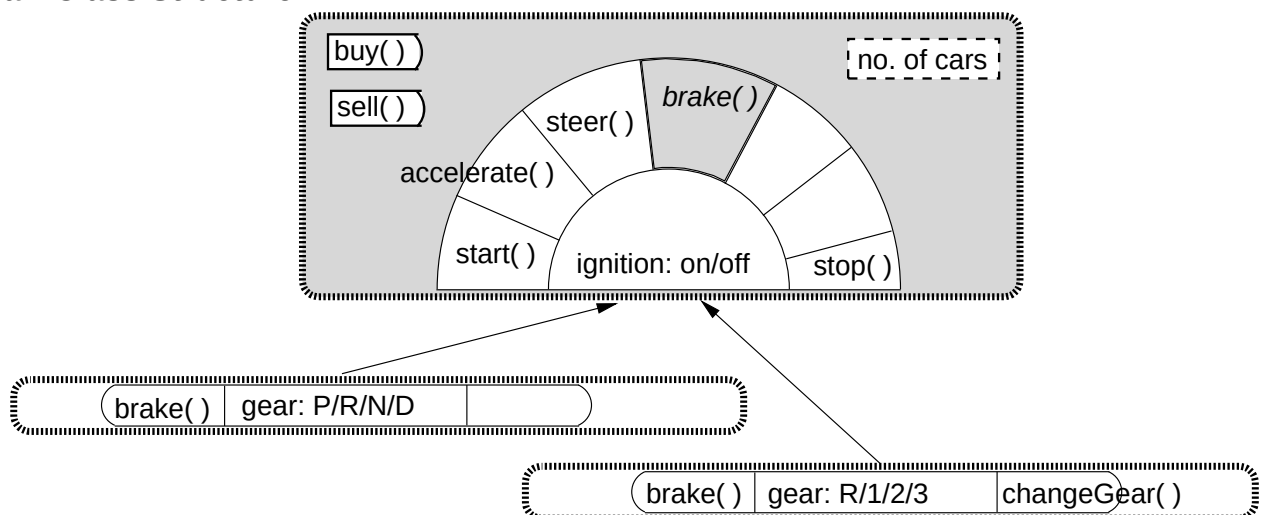
Example: Object, Class and Class-Hierarchy

As an example of such a general object-notion may be taken a new car coming from a production line: Although the cars look alike, the car under consideration is a unique one, it has an identity. It has also a state describable by the gauges for gasoline, motor oil, braking fluid, gear box fluid, cooling water, battery voltage, engine on/off, lights on/off, blinker right on/off, blinker left on/off, warning blinker on/off and other dashboard indicator positions. The behavior of that car, starting the engine, driving forward, driving backward, changing the direction of driving is described by more or less complex sequences of actions as found in the handbook of the car (or to be learned by driving it). The graphic below presents two objects which can be seen as simplified representations (so-called abstractions) of a real car:



One car may have an automatic transmission. Another car may have a mechanical gearbox, this changes the possible state of the car (possible gear-positions are 1/2/3 instead of P/N/D) and this even extends the behavior of the car by an explicit complex changeGear()-procedure. So far, the structure of the above objects can be overseen easily. This view may change in cases where ten or hundred cars have to be considered, abstracted in the way above. If more cars are to be collected, then the different structure of the objects would become clearer by isolating common states and behaviors to get the pure structure of the collection of objects:

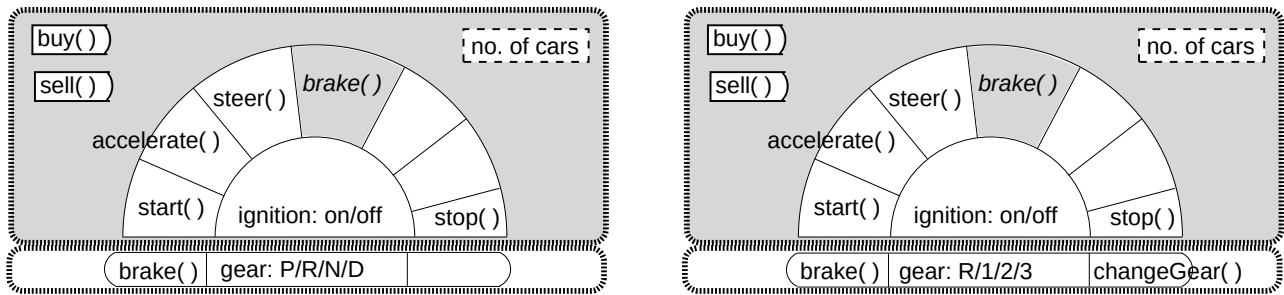
Car- Class Structure



(The various features, like rounded frames and grayed backgrounds, of the graphic above will be described successively in the following paragraphs.) Of course, the pure structure above does not relate directly to any individual object, but any given individual object can be classified fast and easily by using the structure above. The structure just collects all the different features used to describe the two kinds of cars above. The iglu-shaped part of that analysis describes a sort of basic car structure giving no information about what gearbox-system such a car contains. The extension of that structure produces two new structures: One of those extensions describes a car with an automatic transmission, and the

other describes a car with a mechanical gearbox. The rounded frame with the iglu-like figure inside represents a structure, called a **class**. That **class can be extended** by adding one of the little frames:

class
class-extension



The resulting new classes above can be used as templates for producing objects (Car-objects⁴⁷ with automatic transmission and Car-objects with a mechanical gearbox). The frame around the iglu-shape, or around the burger-like shape indicates that it is a class or part of a class-structure. Producing objects from a class is also called **instantiating the class** or **generating an instance of the class** or **deriving the object from the class**. This instantiating process in a way resembles the stamping-act: The class is used as a stamp for printing out one or more objects.

instantiating a class

The structural analysis of the Car-objects produced a **class-hierarchy** given by the abstract Car-class, without any gear-specification, and the two extended classes, above: A class for cars with a mechanical gear-box and a class for cars with an automatic gearbox.

class-hierarchy

Buying a car can be seen as instantiating the car in your living area. Selling removes it. Buying, selling or scrapping the same car more than once in the time of your ownership would be unusual. So buying, selling, scrapping can be seen as actions that are part of the behavior of the Car-class rather than reformulating them for individual Car-objects (The sales-man or the car-crunching machine are not integral parts of the car's state or behavior!). These behaviors can be assigned to the Car-class and then are called **class-behavior**.

class-behavior

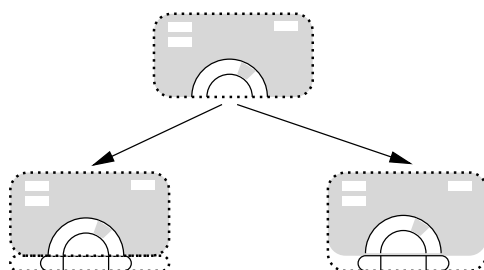
The number of Car-objects, which are described with this class-model, neither characterizes the individual Car-object; but that number describes the *set of Car-objects* which is considered. So, the “number of Car-objects” can be seen as a **class-state**, to be associated with the Car-class. In the frame that fixes the class-structure, the class-states and class-behavior are collected as squares or square-like boxes in the upper corners.

class-state

The graphics above allow another interesting observation: The base-class, that can be extended, cannot be sensibly instantiated; the Car-object would have an undefined gearbox and therefore couldn't describe an average real-world car. – Such classes which cannot be sensibly instantiated are also called **abstract classes**. But, as can be seen above, abstract classes can be used for devising clearer structures and abstract classes can have a defined behavior and a definable state which they transfer onto the extended classes (the ignition on/off state and the other behaviors) and instantiated objects thereof. Abstract classes here are written with a gray-underlaid frame, to make them discernable from non-abstract, instantiable classes. (The Java-implementation of the notion “abstract class” is given on page 183 and an elementary example in Java can be found on page 181.)

abstract class

Car-Class Hierarchy



grey background: abstract class
grey behavior: abstract method

methods and variables

As could be seen above, the very general notion of “behavior” usually is realized as a set of actions or procedures also called **methods**. Whereas the notion of a “state” usually is realized as a set of **variables**. There can be discerned so-called **instance-variables** (ignition with the values on/off, gear position with the values P/R/N/D) used for describing the state of an object and so-called **class-variables** (number of Car-instances) describing the state of the class.

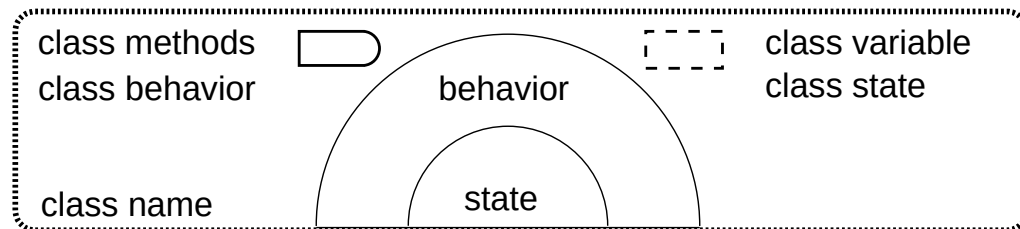
instance-variables and class-variables

⁴⁷The upper-case letter “C” indicates that “Car-object” denotes a mental image — the model — of a car, not the real thing!

instance-
methods and
class-methods

The same classification can be applied to the *behavior* of classes and objects: There are **instance-methods** (start(), accelerate(), ...) that describe the behavior of the individual object. These instance-methods can be invoked for each object individually (car1.accelerate(slow) and car2.accelerate(fast)). – But there are also **class-methods** which describe behavior of the entire class: Like Car.buy(car1) and Car.buy(car2). The act of buying is taken as a class-method, because the buying-process can be described independently from the individual car, whereas for example the price of the car describes an aspect of the car's individual state. Later, in the context of programming-languages, the equivalent to producing and buying a car can be seen as the instantiation of an object by using a class-specific constructor-method.

These remarks make it possible to identify the various abstracted notions in the graphical representation of the class-model:



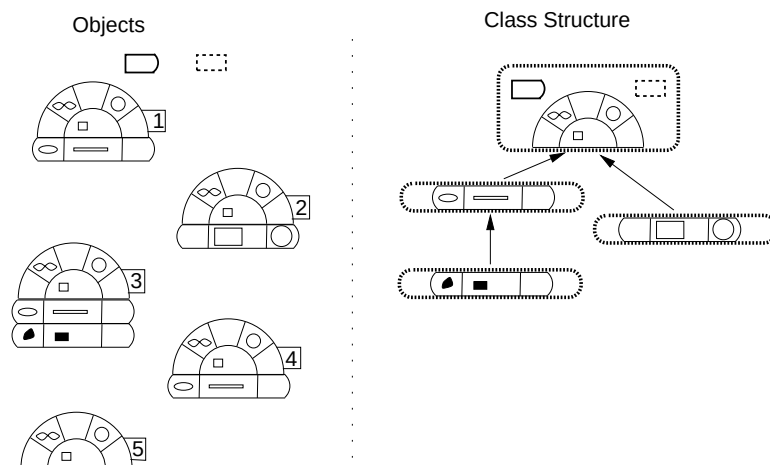
abstract
method

Finally, consider a reason why even unimplemented methods, so-called **abstract methods** may be a sensible idea: The brake()-method is a method of the abstract Car-class. But the braking-process differs in a car with a mechanical gearbox from that in a car with an automatic gearbox (The clutch should be released during a heavy braking-process.). So, specifying the braking process for the entire Car-class becomes difficult because the brake()-method has to be implemented differently for the two derived classes. But still it may be useful to have at least the idea of a brake()-method within the Car-class, because braking is what every car should be able to do. This necessity can be indicated with a non-implemented (abstract) brake()-method in the Car-class. (The Car-class itself couldn't be instantiated, because there exists no conventional car without a transmission of any kind, and therefore the entire Car-class has to be called abstract.) Of course, a class with at least one abstract (unimplemented) method automatically has to be declared abstract, because objects with unimplemented methods violate the idea of a defined behavior.

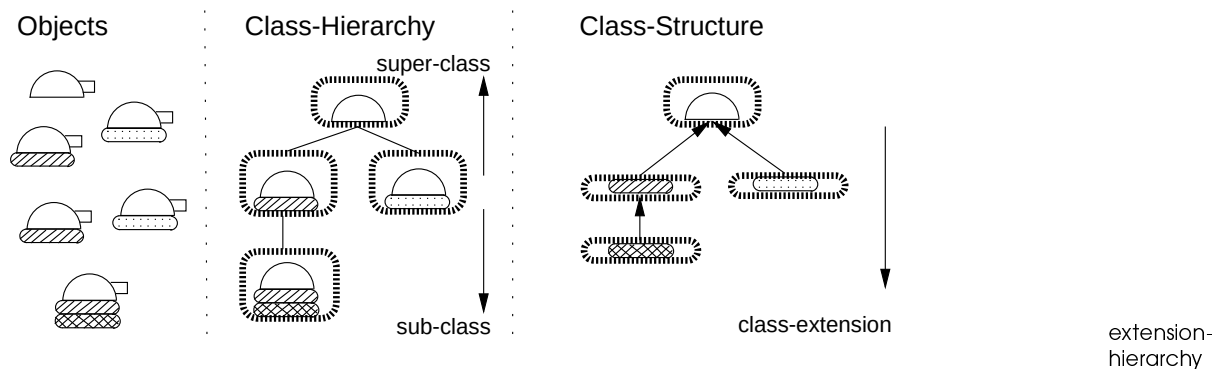
abstract class

What has done with the brake()-method could have been done also with the accelerate()-method, assuming that acceleration is thought of involving a changing of gears. Why the difference? There exists no particular reason for that difference: Class/object-structures are models chosen by an onlooker for describing an area of interest or a problem according to the onlooker's necessities. THERE IS NO INHERENT ULTIMATE TRUE STRUCTURE TO BE FOUND, the choice above was taken just to illustrate the class/object-model. Of course this freedom, or seemingly arbitrary approach, can be used to properly design classes to depict real-world-problems; with the intention of letting this class/object-model be re-used as much as possible. — Making class-design, problem-oriented as well as anticipating future needs, can turn out to be more difficult than expected!

The relation between a collection of objects, class-methods or class-variables and a given class-structure is restated in the following graphic:



Observe that all objects share the class-variables and the class-methods. Objects Number 1 and Number 4 are equal but not the same! (Their state and behavior are the same but they have been given different identities.)



A given class-structure, like that of the Car-class, makes it easy to construct sub-classes by adding class-extensions. In the (resulting) class-hierarchy, also called extension-hierarchy, those classes that have been extended often are called **super-classes**, their extensions consequently are also called **sub-classes**. Observe also that THE MOST COMPACT DESCRIPTION OF THE CLASSES CONSIDERED GIVES THE ORGANIZATION OF THE CLASS-STRUCTURE. Therefore in programming and documentation, classes are economically declared or described by using the more fragmented but least redundant representation given by the class-structure (and avoiding the organization as a class-hierarchy or worse as a collection of objects).

super-classes and sub-classes class-structure: most compact description, thus widely used!

The notions "object" and "class" thus can be seen from two sides:

(Class as Template for Creating Objects) A class can be seen as a collection variables and methods that can be of the instance- or class-kind. A class-hierarchy is given by a structured collection of these variables and methods. The class then serves as a template for objects; and the hierarchy of classes represents a tree-structure of successively refined templates. Class-methods and class-variables can be used independently from the existence of an object.

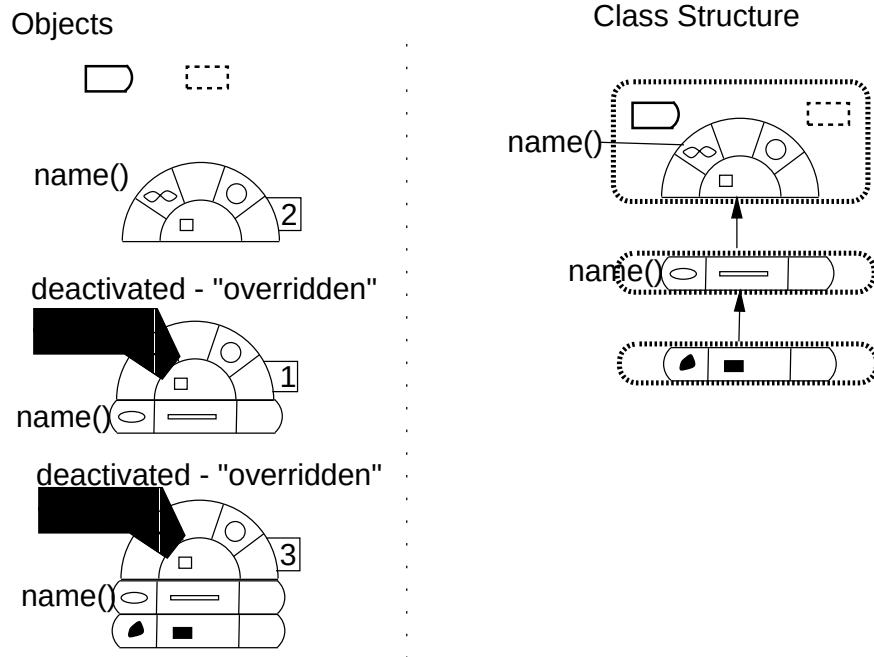
(Object Collection is abstracted to Class) Consider several similar objects *together* with methods that describe how to construct and how to destruct an object: The methods that exist independently from the existence of any object are called class-methods. There may be a variable that has the same value for all objects under consideration, that variable is likely to become a class-variable. Another variable that occurs in some of the objects and contributes to the object's individuality by having different values from object to object needs to be an object's state- or instance-variable. Depending on the variety of the given objects, there are compilable different collections of variables and methods. These collections constitute the classes that can be arranged into a class-hierarchy.

So both approaches, the one starting with the class-notion, or the one starting from the object-notion are part of - or result in the same object/class-model.

OUTLOOK: The class-structure is what will be used to define class-hierarchies in Java on page 163. An example of a larger class-hierarchy is given by the graphic on page 564 in the context of arranging a Graphical User Interface; details of the emerging class-structure of the three base-classes (Component, Container, JComponent) are given successively in the lists on pages 553, 556 and on page 559.

Overriding Methods and Shadowing Variables

Overriding Methods and Shadowing Variables — Consider the brake()-method in the graphics around page 132; there the abstract method has been redefined differently in both sub-classes. What about re-defining non-abstract methods of super-classes? Or formulated conversely, what if a method or a variable in a class-extension has the name of a method or variable in the super-class? First, objects derived from (one of) the super-classes use the original method, BUT all objects derived from the class-extension or its sub-classes use the newly defined (instance-)method or (instance-)variable. The overriding of a method has been visualized in the graphic below:

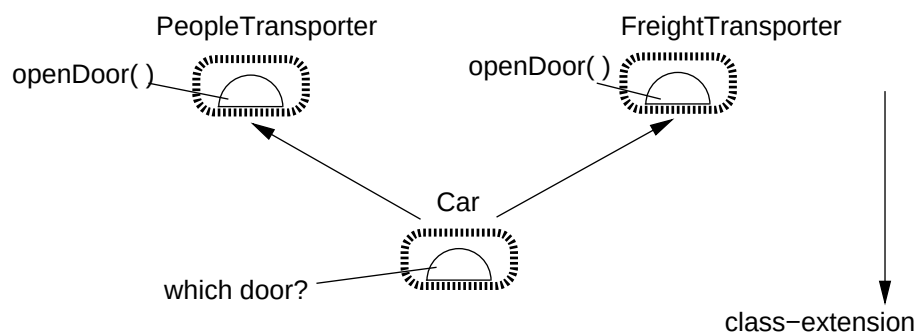


Remember the shadowing of variables by local variables of Java methods as introduced on page 123, shadowing of variables by class-extension can be imagined likewise. Overriding class-methods and shadowing class-variables would produce *class*-specific variants. For an example consider the Java source-code on page 167 and the corresponding footnote following that source-code.

Class-Extension — Inheritance with multiple Super-Classes

Class-Extension — Inheritance with multiple Super-Classes — Another approach to the process of extending classes may be thought of by imagining a class having *several* super-classes. Example: Below the Car-class extends the class PeopleTransporter and the class FreightTransporter. Each instance of a Car may be used to transport people as well as freight:

Class-Structure with Multiple Extension



Example I: The Car-class may extend the class PeopleTransporter, but at the same time the Car-class may extend the class FreightTransporter. This makes it possible to construct an ambiguity: Both classes may use a method openDoor(), but as can be seen with the example of a standard car, such a method usually means different doors (doors to the area with seats or door of the car's trunk). Though, here, the ambiguity may seem constructed, a general method for structuring contexts should be designed to spare the programmer or the user unnecessary complexities. (Sketch the class-structure for this example, the graphic above may serve as an orientation.)

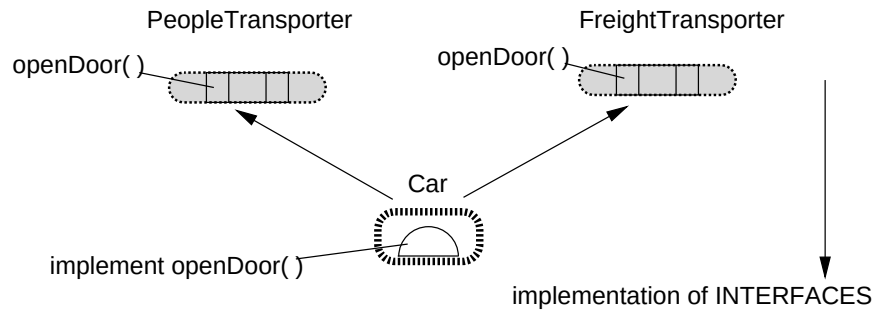
From above emerges a BASIC PROBLEM: MULTIPLE INHERITANCE BECOMES AMBIGUOUS IF TWO BASE-CLASSES HAVE METHODS OR VARIABLES OF THE SAME NAME. The question may arise about which member takes precedence when they both have the same name. Therefore, using classes that extend multiple super-classes should be considered rather careful.

A tricky workaround concerning this predicament can be formulated as follows: Consider a class having only named, but unimplemented methods, also called **abstract methods**. This makes such a class automatically abstract, because it obviously cannot be sensibly instantiated. This kind of class, with no

apparent functionality, furnishes a kind of template and this seems to be the best one can get, to formulate multiple inheritance without an inherent source of contradictions or unnecessary considerations about precedence.

Example I (continued): So, using the example above, PeopleTransporter's openDoor() and FreightTransporter's openDoor() may be unimplemented. When both are extended to the Car-class, the openDoor()-method has to be implemented considering the two types of doors for loading people and for loading freight.

Class-Structure with Interfaces (fully abstract, multiply implementable)



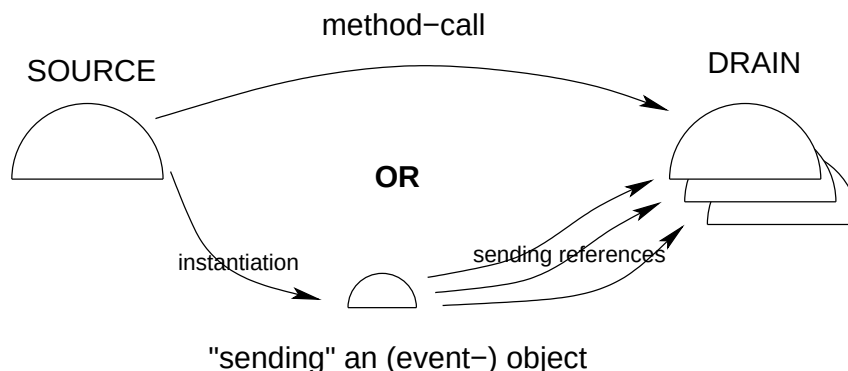
“Classes” which contain only unimplemented, so-called abstract, methods AND allow other inheriting “classes” of the same kind at their side are called “interfaces” in Java. The name “interface” signals the option to realize a kind of multiple inheritance as described above. Java⁴⁸ classes can implement *multiple* “interfaces” and thus furnish a version of multiple inheritance that proves relatively free of inherent complexities. For more see page 185.

Communication-Mechanisms

Communication-Mechanisms — The structuralizing of objects by classes and class-hierarchies may be considered as useful, but coexisting objects should be able to communicate to simulate real-life interaction or just to make the object-model more efficient:

Method-Activation or method-calling represents the natural means of communication between objects. This requires the objects to know each other well, but this knowledge of each other may also involve an unwanted specialization of the objects.

Communication by method-call or by dispatching an Event-Object



Message-Objects exchanged between communicating objects generally requires the objects to know very little about each other: The only knowledge, that remains necessary, is to know how to interpret the message-objects, how to send them and how to listen for them. The communication is thus realized by exchanging messages between objects, this “uncouples” objects. Then, the methods that have to

⁴⁸ The programming-language C++ allows multiple inheritance which has to cope with making inherited behavior non-ambiguous.

be called, are only listening- and dispatching methods, which, by following conventions, generally simplifies matters! For an object in a message-exchange context, a carefully-designed set of message-objects furnishes a relatively lean interface to its outside world. The rest of the implementation of the object could be changed, if necessary, without changing the behavior of the object. This isolation of the inner workings of an object exemplifies the concept called **encapsulation**. Like in proceduralization, encapsulation of object-functionalities means hiding the details of the implementation.

For more Java-related information on general messaging see pages after page 271. If the interaction between objects is realized by exchanging message-objects rather than by issuing object-specific method-calls, then this is one first step towards what is often called **component-model**. There, objects usually send and filter highly normed event-objects. For more information on the area of software-components, see page 311.

Object/Class-Model Keywords

Object/Class-Model Keywords — The following paragraph gives a small glossary on the main notions of the object/class-model:

abstract- or virtual class A CLASS THAT CANNOT BE INSTANTIATED, this means that no instances of that class can be created. Abstract- or virtual classes are collections of common states and behaviors; but these become only part of an object that is an instance of some sub-class of the abstract- or virtual class. Some of the methods may be implemented and others may be declared abstract. Example: A Car-class which lacks the specification of the transmission type is an abstract class, the brake()-method may be abstract but the accelerate()-method may already be implemented.

abstract method A named method with no functionality defined. A class which contains at least one such unimplemented method automatically cannot be instantiated; that means a class with a single abstract method becomes implicitly abstract itself! A standard use of abstract methods can be found in multiple inheritance, where these method-declarations (in Java “interfaces”) are used as templates. But as could be seen with the example of the Car-class, when considering the brake()-method non-abstract, classes can be considered to be abstract without having abstract methods. A car has to have either an automatic transmission or a mechanical gearbox! (Unless it is an electrical car; this option has been ignored as the Car-model was introduced.)

class A collection of possible states and behaviors, either gathered from a set of similar objects or established from a given problem or area of interest (**class as collection of states and behavior**), (**class as an abstraction of a collection of objects**). In programming, a class can ALSO be taken as a programmable type-concept (**class as a customizable type**). A class can be seen as a template that furnishes the fields (state) and methods (behavior) of an object (**class as a template**). A class gives the behavior and the possible states; an object of that class has the same behavior but only one fixed state and an identity. Due to the given identity (a registration-number or some other tagging-instrument) of an object, there can exist several objects with the same state, instantiated from the same class.

class-extension Defining a new (sub-)class by adding other kinds of states and behaviors, respectively kinds of variables and methods to an existent class.

class-hierarchy A collection of classes, arranged according to their dependencies in form of sub- and super-classes. These kind of dependencies usually form a tree-like structure.

class-methods Methods which are associated with the class, not with an individual object. For example, methods (also called constructors) for creating instances of that class. In Java, beside constructors, so-called **static methods** represent the class-methods.

class-variables Variables that are associated with the class, not with an individual object. For example the number of objects that have been instantiated during the use of the class can be counted by using a class-variable. In Java, class-variables are also called **static variables**.

container-object An object which contains a list of references pointing to other objects. Container-objects typically implement buffers, stacks or little non-persistent databases. Java supplies container-objects, like instances of the class `java.util.Vector`. CAUTION: Though Java array-types can index objects, array-types are something different than class-types (See the syntax-diagram on page 143, and the introduction on page 234.). Only from Java class-types can be instantiated objects!

encapsulation Hiding, often complex, self-contained solutions of problems and giving them a much easier programmer- or user-interface. This allows a relative ease of use without having to grasp inner complexities. Proceduralization offers a means of encapsulation by needing only to know the method's signature and return-value to use it. Object-Orientation can be made to work similarly on the level of the entire object. Encapsulated code may document its functionality in a so-called (Application) Programming Interface (API). Especially in the context of networking, the effects of simplifying things, by encapsulation of code, are circumscribed by the notion "transparency".

inheritance All sub-classes (or class-extensions) receive the methods and the variables furnished by their super-classes (sometimes also called parent-classes). INHERITANCE DESCRIBES THE SAME AS CLASS-EXTENSION; but inheritance focuses on the sub-classes whereas extension stresses its starting-point being the super- or parent-classes.

instance of a class An object derived or instantiated from that class. The class represents a template for state and behavior; an object bears a concrete state, a concrete behavior and an identity(-tag). Also refer to the item "object" below.

instance-method A method associated with an individual object (in contrast to a class-method which is associated with the entire class). The nature of that association, for example, makes the method change one of the object's variables. Instance-methods are taken to represent the behavior of their object.

instance-variable Instance-variables are associated with an individual object (in contrast to a class-variable which is associated with the entire class). Instance-variables describe the state of their object.

method A method can be called (for example by other objects) to alter the object's state or to send messages to other objects. In programming practice, a method (also called function, procedure, subroutine) usually represents a kind of named piece of code associated with an object or class. Usually methods are the means by which another object can change the object's variables (alter the object's state).

message A signal from a source-object to a target-object, usually requesting a method-invocation of the target-object. Sometimes, simply calling the method of another object and transferring information within the method's arguments is considered as sending a message! But, often messages are objects themselves, then they bear a standardized structure, which is known to all communicating instances. Then, the method-calls used for communication of objects become highly standardized. Usually a message-object contains the name of the sending object, the name of the receiving object, the name of the method to invoke and some variables that are used as the invoked method's arguments. This concept has been developed further beginning with page 271 and on page 571.

multiple inheritance The case where a class inherits state and behavior of multiple super-classes (In Java, multiple super-interfaces). To avoid ambiguities of the super-classes with same variable- or method-names, only abstract methods and unassigned variables should be used. The Java interface-construct furnishes this restriction.

object An entity with a **state**, a **behavior** and an **identity**. In programming, an object is given as an entity of variables and methods together with an identity. This generality often reduces to an object being seen as a variable that has been instantiated from a class-definition, where the class is taken to be a sort of generalized type-definition.

overloaded methods Instance-methods are identified by their object's name, their own name and their parameter-list⁴⁹. So even in their own class, different methods can have the same name but a differing parameter-list to remain valid. Such methods are called overloaded. Remember the section beginning with page 124.

overridden methods Overridden methods always occur in the context of a class-extension: A class-extension may define a method with the same name and the same list of parameters as a method given by the super-class. The newly defined method becomes the default method of any objects instantiated from this extension or any further extensions. An analogon on the level of variables presents itself: Two variables with the same name but different overlapping scopes produce the following effect: The variable with the limited scope overshadows the other. (Why are there no overridden variables? Because overriding variables would amount to redefining the type of the variable-name, nothing more.)

⁴⁹Class- or static methods are identified by their class-name, their own name and their parameter-list.

Pragmatics: Constructing class-hierarchies, to describe a problem efficiently, usually turns out to be demanding and even may need redesign efforts. But there are some simple rules and conventions that may ease the task: When describing the real-life problem, using the everyday language,

nouns indicate an object or a class to be used in the object-oriented analysis,

verbs indicates a method's functionality and the formulation

is a name indicates something to be an object or a sub-class.

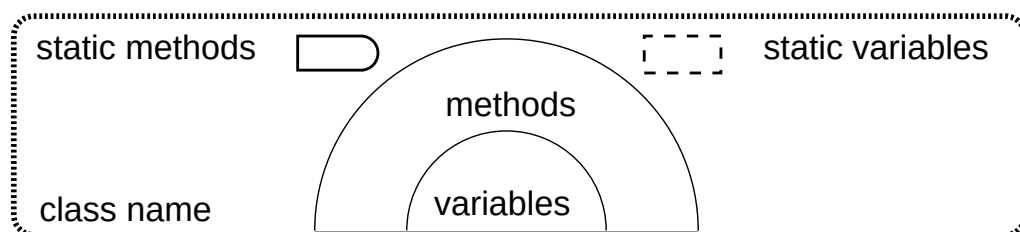
Object-oriented analysis of problems can happen without using a programming-language. Object analysis represents a problem-oriented method of program-design and therefore usually happens *before* constructing source-code. But object-oriented design usually addresses itself to EXTENSIVE real-life problems with less dominant algorithmic complexities. The wealth of approaches and methodologies of object-oriented-analysis and -design is described in books or related Internet-sites.

Concept: Object/Class-Model within Programming-Languages

Concept: Object/Class-Model within Programming-Languages — In many programming-languages, aggregates of variables are called records or structs. Aggregates of command-sequences and variables are described by functions, methods or subroutines. Here aggregates of variables AND methods are called **classes**.

```
class name
{
    variables;
    methods; // usually for manipulation of the variables
}
```

In that context, remember the object-model on page 131 and the graphic of the general class-model on page 134. The general class-model, introducing state and behavior, changes in the context of a programming language to a class-model given by an aggregate of methods and variables:



Classes in programming-languages can be given another variant of definition: Classes just furnish a way for the user to aggregate data (variables) and methods (functions, subroutines). Object-oriented programming-languages also furnish ways to introduce static class-members and ways to derive individual objects from a given class.

For practical programming purposes the notion of a class can be used

- to introduce programmer-defined types,
- to use a class as a self-contained program and
- to use a class-structure as an index for reference when re-using software:

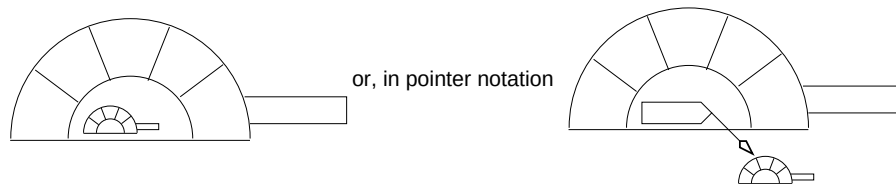
Classes as Custom-Programmed Types Classes can be used to define programmer-specified types within the programming-language. These programmer-defined types give templates for the data and furnish the methods to manipulate the data. The central idea aims at presenting the data and methods for the data-manipulation as an entity. Thus, in other contexts, methods for data-manipulation do not have to be re-written. To the programmer, both, the data and the methods, are related intrinsically; just like any integer type furnishes mathematical functions like addition, subtraction, multiplication and division for its manipulation:


```
class Integer // this is a concept-class and does not work in Java
{
    int i;
    addTo(int j){return i+j;}; multiplyWith(int j){return i*j;}; // etc...
}
```

If the standard types of the given programming-language can be supplemented by types defined by classes, then variables of class-type even can be declared inside classes:

```
class ClassName1
{ ClassName2 identifierOfObject; // declaration of variable of class-type
}
```

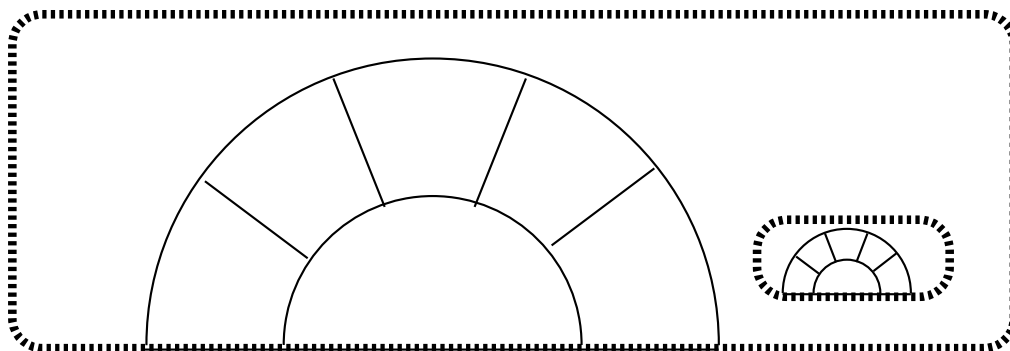
Thus a variable of class-type — an object itself — can be seen as part of the state of an object:



So the state of an object is given by objects (pre-programmed or programmed by the application-programmer) or by variables of primitive-type. Objects of standard classes given by the programming-language (Java: String) or variables of primitive-types (Java: int) finish the recursion indicated by the diagram above (objects in objects in objects ...). For example, in the concept-class above, named Integer, the int-variable of primitive-type, given by the programming-language, defines the state. A practical example of this idea is given on page 142, where a newly-defined type (VarInteger) is used for declaring and instantiating a variable (there the variable can be seen as an equivalent to an object).

Instead of containing only a variable-declaration, a class even may contain a class-definition, a so-called inner class, giving a kind of local type:

```
class ClassName1
{ class ClassName2 // class-definition
    { ...
    }
    ...
    ClassName2 name; // declaration of a variable of that local type (inner class)
}
```



Inner classes are described further beginning with page 157.

Class as a Program A class can be a flexible type, or if a class is furnished with a `main()`-method, then a class can be a program:

```
class Program // this is a concept program and does not work in Java
{
    main(){ statements }
}
```

If the source-code inside the `main()`-method references other classes or creates and references other objects, then the class with the `main()`-method does represent the central piece of a program! Conventionally, classes with a `main()`-method enable an end-user to issue some command on the level of the operating-system, eventually triggering a call of the `main()`-method (the `java`-command entered into a terminal-window of the native operating-system triggers a call of the `main()`-method). This peculiarity gives classes with a `main()`-method all the characteristics of a regular program. Thus, in most object-oriented programming-languages, the class-notion becomes more than simply a flexible way to define new types.

Class-Structure as Index to Software-Library But the perspective of the class/object-model gives another important functionality to object-oriented programming-languages: As can be seen in the graphic on page 135, class-extension lets the code be positioned in a sort of programmer-created hierarchical index-structure for easier identification. Successive class-extensions and the ensuing class-hierarchy can be used to **realize a structure of the involved objects**. Since the super-classes may be used multiply (remember the car example above), that kind of structuring also makes it possible to easier re-use code.

Java: Classes as Programmer-Definable Types and Programs

Java: Classes as Programmer-Definable Types and Programs — Java classes can be more than templates for objects. A Java class with a `main()`-method constitutes the center of a program. The fully-functional example below defines two Java-classes: The first class introduces a programmer-defined type and the second class acts as a program due to its `main()`-method. (The use of the modifiers `public`, `static` and `void` is described elsewhere: `public` on page 175, `static` on page 147 and `void` beginning with page 117)

```
// a Java class: a programmer-defined type or a program
class VarInteger                                // programmer-defined type
{ int i;                                         // state: instance-variable
  VarInteger( int x ){ i = x; };               // class-method: constructor
  public void set( int x ){ i = x; };           // behavior: instance-methods
  public int get(){ return i; }
  public void increment(){ i = ++i; }
  public void print(){ System.out.println( i ); };
}
public class Program                            // program or application
{ public static void main( String unused[] )    // class-method: static main()
  { VarInteger theObject = new VarInteger( 0 );
    String theOtherObject;
    theOtherObject = new String( "Hello" );
    theObject.print();
    System.out.println( theOtherObject );
  } }
/*
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro% javac Program.java
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro% java Program
0
Hello
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro%
*/
```

Another example of a programmer-defined type is shown in the source-code on page 76, where the `Pair`-class has been introduced. These two approaches for using Java classes (program and custom-defined type) are the most typical and elementary ones. – A class may also be used to just furnish a collection of methods; as does the `java.lang.Math`-class which just offers a range of mathematical functions to the Java programmer (for more see pages 94 and 257).

An instantiation of a class with a `main()`-method has no special meaning with respect to the `main()`-method and its program-character: The `main()`-method just remains one of the static methods, except that it can be activated by an external event (the issuing of the “`java`”-command). A first example of such a case is given on page 159.

Java: Reference-Types and their Instantiation with the `new`-Keyword

Java: Reference-Types and their Instantiation with the `new`-Keyword — Remember the introduction of primitive-types on page 83. Variables of primitive-type have their identifier fixed to an address in the computer’s Random Access Memory (RAM). These types, with identifiers being an unchangeable reference,

A class with a
`main()`-method
remains
instantiable!

thus being glued to the RAM-address and exhibiting a fixed behavior (addition, division with number types), are called atomic- or primitive-types. Primitive-types constitute an integral part of the Java language and prescribe the data-format (int: 4bytes) and the possible manipulations (Addition, Subtraction, Multiplication, Division). With variables of primitive-type assignment happens by-value, this means that an assignment copies the value to the variable-name's part of the RAM. Usually a computer processes variables of primitive-types faster than Java objects of class-type. remember: assignment by-value

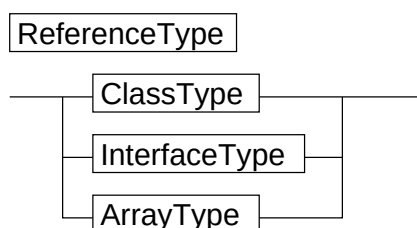
Java offers two kinds of basic non-primitive-types, the so-called reference-types :

class-type Currently discussed. A variable of that type is instantiated by prefixing a constructor-call with a new-keyword; for more see page 145.

array-type Realizes a simple form of aggregation of values or object-references. Variables of class-type as well as variables of primitive-type always can be aggregated as an array. Variables of array-type are recognizable by the brackets [] used in declaration and referencing, for more see page 234.

An example of introducing a class-type has been given in the preceding program on page 142. There has been defined a class(-type) "VarInteger" and from this class has been instantiated an object (referenced by an identifier "theObject").

The interface-type gives a variant of the class-type for cases of multiple-inheritance, for more see page 185.



ReferenceType := *ClassOrInterfaceType* | *ArrayType*

ClassOrInterfaceType := *ClassType* | *InterfaceType*

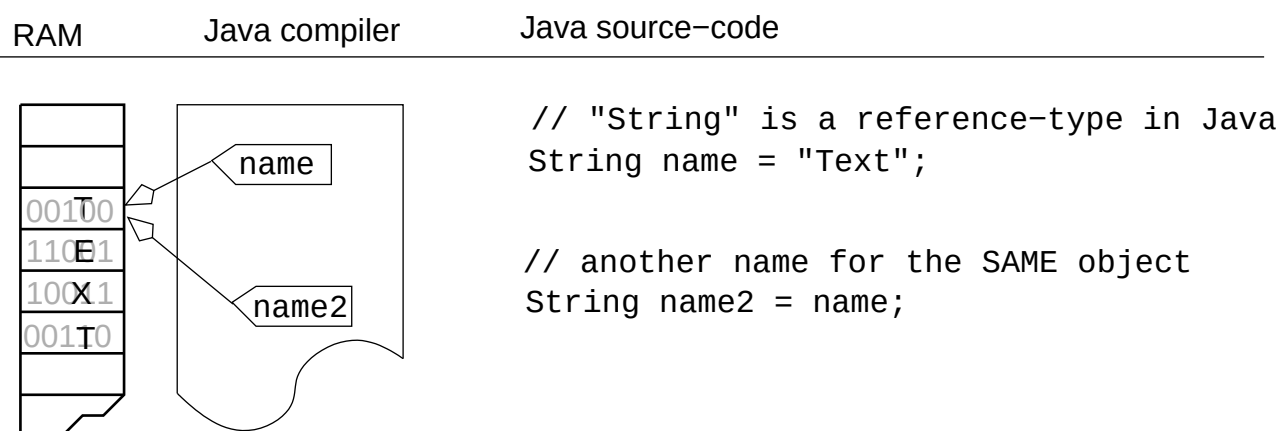
ClassType := *TypeName*

InterfaceType := *TypeName* — *ClassType* 146, *InterfaceType* 187, *ArrayType* 234

(*ClassOrInterfaceType* is used in *ClassInstanceCreationExpression* on page 151)

Like variables of primitive-type, variables of reference-type have identifiers that point to an address in RAM-space. But, the RAM-space-address of variables of reference-type is changed by assignment (=), NOT the value stored there, as it is the case with variables of primitive-type. IMPORTANT TO NOTE: IDENTIFIERS OF VARIABLES OF REFERENCE-TYPE CAN BE SEEN AS ALIASES AND THEIR POINTING-TARGET CAN BE CHANGED BY ASSIGNMENT:

Variable of reference-type:

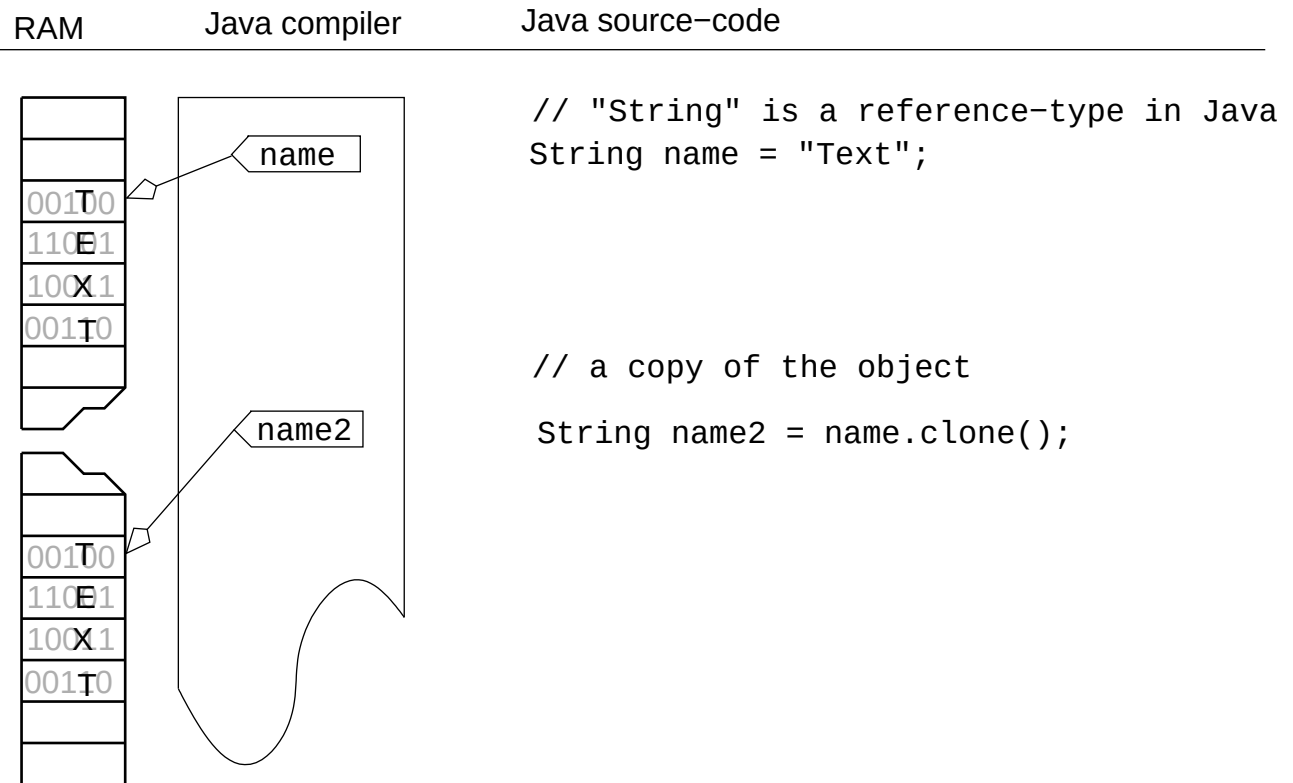


A reference is a container for a RAM-address which can be changed by the programmer by assignment. That's an important difference to Java variables of primitive-type where the references are fixed to an address of the RAM-space. And the act of assigning variables of primitive-type to each other with the equal sign (=) means copying. (see the graphics on page 83) For variables of reference-type assignment assignment by-reference

(=) means: If objects and associated names are given, reassignment of names by using the equal sign (=) only exchanges references, not the entire object, as indicated within the graphic above.

Copying of an object, a variable of reference-type, is done with a `clone()`-method as described below:

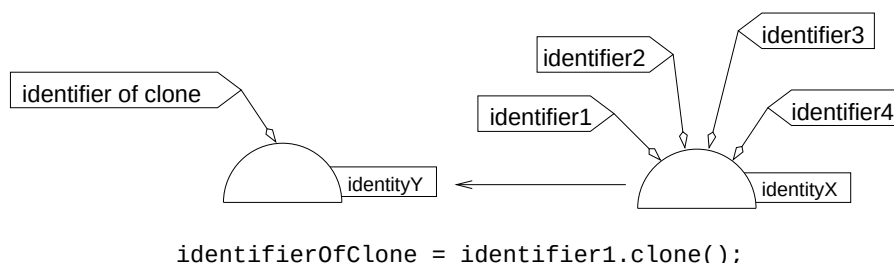
Copying a variable of reference-type with the `clone()`-method:



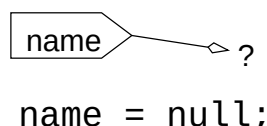
The `clone()`-method PRODUCES A NEW OBJECT that contains the same data AND RETURNS A REFERENCE TO THAT NEW OBJECT! (It does NOT “return the new object” itself!)

This statement is true in all generality: Java-methods with return-types of the primitive kind return the variable-value itself. But if the RETURN-TYPE OF ANY JAVA METHOD IS OF REFERENCE-TYPE, then, like the `clone()`-method, the method returns only a reference to the object!

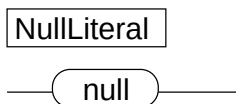
The name of every reference-type could be suffixed with the name “Pointer” (StringPointer instead of String), to reflect the difference to the named value of data that has a primitive-type. Data of primitive-type has only one name, assignment with the “=”-operator copies that value. Whereas data of reference-type can have multiple names, meaning that multiple pointers are referencing the same data; here assignment with the “=”-operator just makes the pointers equal. The assignment-operator, used with identifiers of reference-type, does not copy the data!



Any name of reference-type can be assigned the literal **null**, which lets the name reference nothing.



```
variableName = null; // an identifier of reference-type not yet assigned
```



NullLiteral := null

Instances of reference-types (to be more precise, instances of non-abstract classes and arrays) are created by prefixing so-called **constructor-methods** with the **new**-keyword or by calling so-called **factory-methods**, described later on page 151. (The clone()-method is a factory-method, because calling this method creates a new object and this method returns a reference to that object.)

constructors and
factory-methods

```
ClassName variableName = new ClassName();
// Class-methods with the class-name are called constructors.
```

The new-keyword in conjunction with a constructor allocates the memory, creates and initializes the object. Then the variable-name (identifier) represents a reference to the object; the variable-name does not represent the object itself, contrary to names of variables with primitive-type!

Below the processes of declaration, memory allocation and instantiation are compared for variables of primitive-type and variables of reference-type:

variables of type-	Declaration	Allocation	Instantiation	
-primitive	<i>PrimitiveType name</i>	=	<i>value</i> ;	
-reference	<i>ClassType name</i>	= new	<i>ClassType()</i> ;	← a constructor-call
-reference (array)	<i>PrimitiveType [] name</i>	= new	<i>PrimitiveType[]</i> ;	(arrays, see page 234)
-reference (array)	<i>ClassType [] name</i>	= new	<i>ClassType[]</i> ;	

Java's String
exception

In Java, the String-class has been allowed an exception to the general class-usage; a Java String-object can be instantiated like a variable of primitive-type:

```
String s = "text";
```

Only in this special case of the String-class, Java allows an instantiation like those used with variables of primitive-type. Actually the String-class also allows every String-object to be instantiated using the new-keyword prefixing the String-constructor:

```
String s = new String("text");
```

This standard way of instantiating objects of class-type has been used multiply in the previous programs.

Java: Classes

Java: Classes — The following sections use the class-concept together with the idea of a reference-type to describe the Java-specific realization of programmer-defined classes. This includes, for example, inner classes, anonymous classes and anonymous objects, supplemented by scoping- or accessibility considerations.

In the previous chapter, the description of proceduralization by using methods had more implications than could be foreseen by the pure concept⁵⁰, the same is true with the implementation of the class-concept by the Java programming-language.

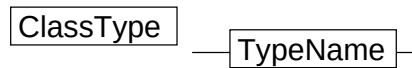
Java: Class-Declaration

Class-declaration may also be called class-definition or class-implementation. The standard Java-form can be seen below:

```
class ClassName { classMembers }
```

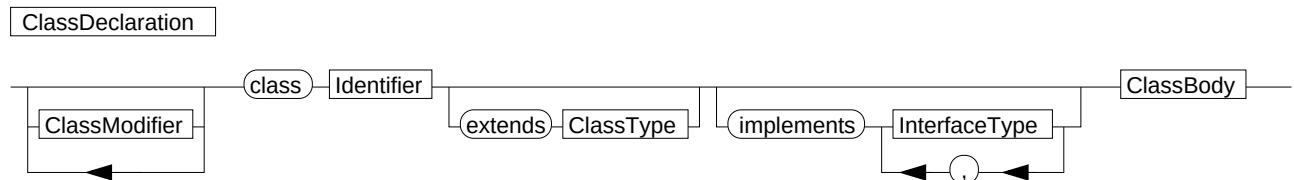
⁵⁰Starting with considering named blocks of code, ending up with local variables making possible overloading, recursion and call-back.

The class-name is also called class-type. The following declaration helps to discern identifiers in the following diagrams:

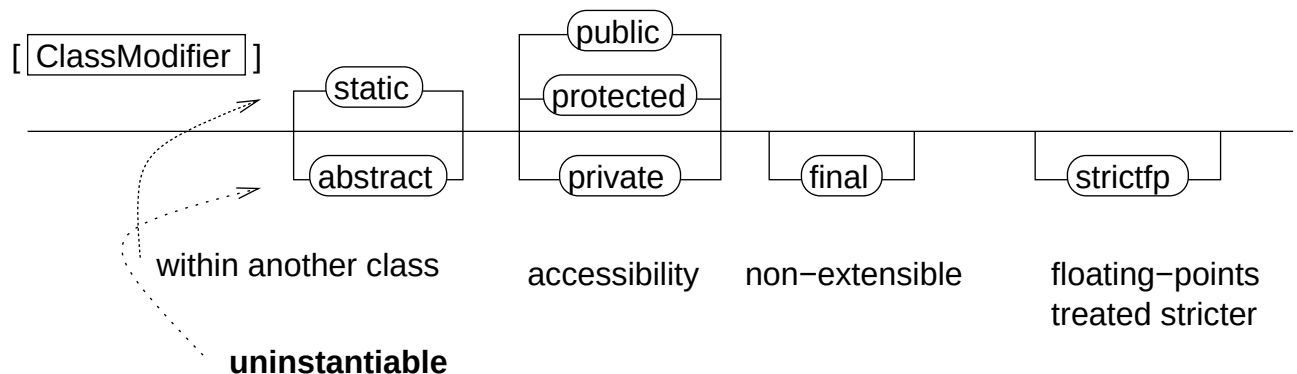


ClassType := *TypeName* — *TypeName* 82

For concrete examples see the programs introduced so-far. The following systematics indicate the remaining possibilities, and are just presented for giving an overall impression and for future referencing.



ClassDeclaration := { *ClassModifier* } *class Identifier* [*extends ClassType*] [*implements InterfaceType* { , *InterfaceType* }] *ClassBody* — *ClassModifier* see below, *Identifier* 81, *ClassType* 146, *InterfaceType* 187, *ClassBody* see below (For the meaning of the keywords *extends* and *implements* see the sections about class-extension and interfacing beginning with page 163)

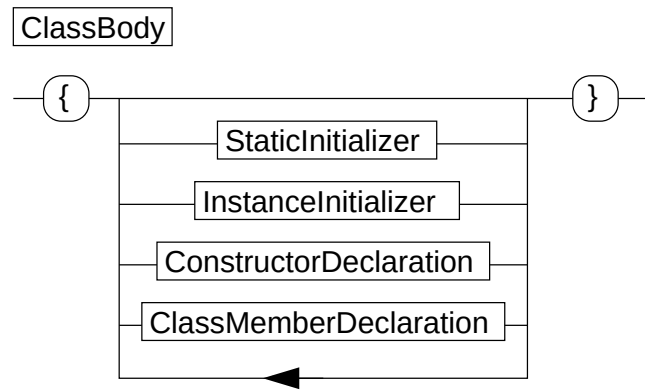


ClassModifier := one of public protected private abstract static final strictfp — (Compare this to the definition of *MethodModifiers* on page 117)

A class can be declared *static* only within another non-inner class (see page 157, past the graphic), an option that may be used rather seldom. An *abstract*-modifier only hinders the Java compiler to support the production of instances of that class, see page 183. For the workings of the modifiers related to accessibility see the sections beginning with page 175. Declaring a class to be *final* makes the Java compiler reject any efforts to extend (page 163) this class. And the *strictfp*-modifier indicates that, within the class, expressions of primitive-types *double* and *float* are treated with more care, as already mentioned on page 117.

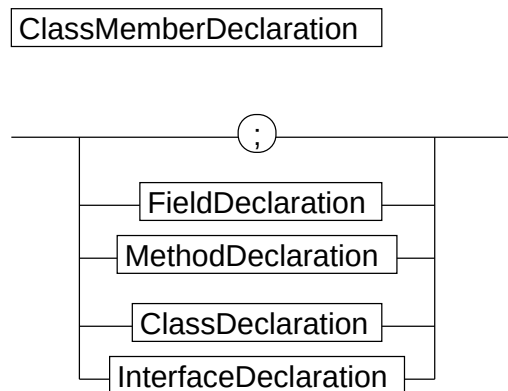
Class-Name: Name of a class as declared in the source-code: `class className{...}`. A class declared *public* has to be stored in a file with the name `className.java`. (One file of source-code can contain only one or none *public* class.) After compilation the byte-code is found in the class-file named `className.class`.

A class-body essentially contains variables and code (methods or method-like entities or class-like entities). In Java, the aforementioned “code” divides into initializers, constructors and members. Variables are also called members:



ClassBody := "{" { *ClassBodyDeclaration* } "

ClassBodyDeclaration := *StaticInitializer* | *InstanceInitializer* | *ConstructorDeclaration* | *ClassMemberDeclaration* — *StaticInitializer* 152, *InstanceInitializer* 152, *ConstructorDeclaration* 150. The Java class-members can be subdivided into the following categories:

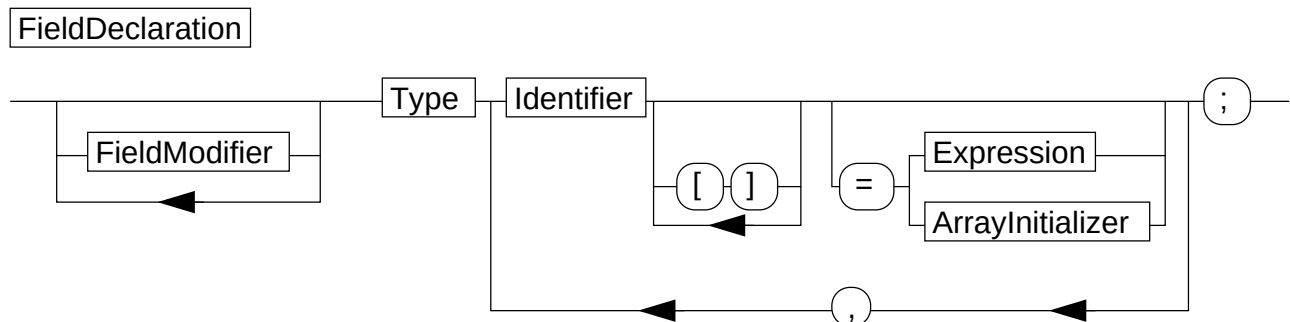


ClassMemberDeclaration := *FieldDeclaration* | *MethodDeclaration* | *ClassDeclaration* | *InterfaceDeclaration* | ; — *FieldDeclaration* see below, *MethodDeclaration* 117, *ClassDeclaration* 146, *InterfaceDeclaration* 187.

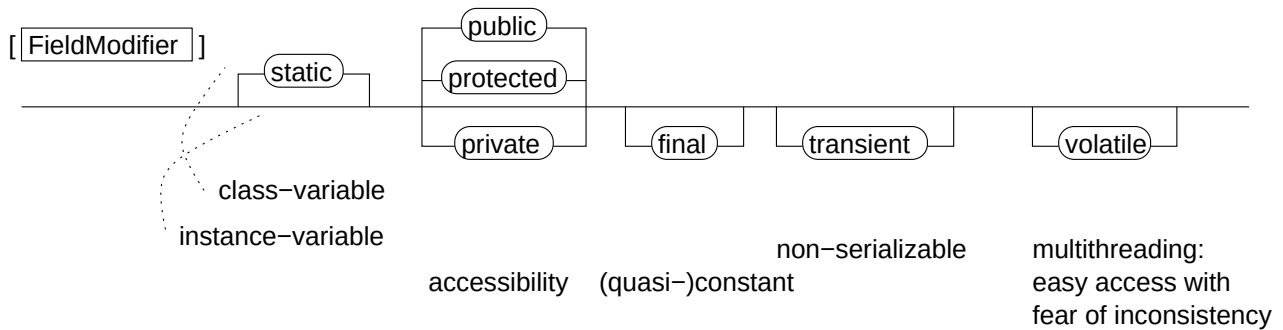
Java fields are class-members representing the standard variables in a class; the standard form of a variable-declaration should be well-known by now:

Type identifier;

This simple idea can be extended in the following way:



FieldDeclaration := { *FieldModifier* } *Type* *VariableDeclarator* { , *VariableDeclarator* } — *FieldModifier* see below, *Type* 76, *Identifier* 81, *Expression* 90, *VariableDeclarator* 121. Within the diagram above, the *VariableDeclarator* has been inserted explicitly. The array-related cases of the syntax-diagram above are treated beginning with page 234.



FieldModifier := one of public protected private static final transient volatile

Field: 1. An instance-variable or
 2. A class- or instance-variable defined in a Java class (also called member-variable).
 Java variables defined inside a method are called local variables.

A static field produces a class-variable; the default produces an instance-variable. For more about controlling scope with the public, protected and private-modifiers consider page 175 (as already stated for the modifiers of method- and class-definitions). Fields declared final are (quasi-)constant (this already has been described on page 122). In the context of serializing objects, for example to make them persistent, transient fields are omitted (see the example on page 309). In the context of multithreading, a volatile field is easily accessible by different threads, but at the cost of potentially inconsistent changes, see page 183.

A typical basic structure of a class-declaration, containing static and non-static class-members, can be given as follows:

```
class ClassName
{
    static Type variableNameS;           // static variable
    static TypeOrVoid methodDefinitionS; // static method
    Type variableNameS;                  // instance-variable
    TypeOrVoid methodDefinitionS;        // instance-method
}
```

The specification of method-declarations has been anticipated beginning with page 117.

An example of a static method represents the, by now well-known, main()-method. Remember that this method characterizes Java-code which can be run by an end-user issuing the java-command in a terminal-window of the native operating-system.

The members with the so-called modifier static are those that are associated with the class itself. Whereas the other members in this example are associated with a derived instance of that class. This follows the idea of “class-” and “instance behavior” of the class/object-model presented on page 133. (Due to its importance, the scheme above has been reformulated on page 154.)

Now can be seen why, up to now, most of the methods in the source-code had to be declared static: Since the class was not instantiated and since the source-code used the class-definition as a program by declaring a main()-method, the auxiliary methods, to be accessible, had to be declared static too⁵¹.

Anticipating the idea of class-extension, as introduced already beginning with page 132, a method or a variable having been defined or declared inside the block of a class-definition and remaining readily accessible after a class-extension process is called member of the given class:

Member: 1. Class-method, instance-method, class-variable (field) or instance-variable (field) in an Java object. In Java, (inner-)class and (inner-)interface declarations are also called members. BUT constructors, finalizers or code-blocks run on loading the class (static-initializers) or instantiating an object (instance-initializers) are not members of a class! The defining condition for being a members of a class, or a **class-member** is given when extending the class and the declaration **remains a valid part in the extended class** (page 166)! As a special case consider that, members declared private are still valid and can be accessed by the original code, but they are not accessible by the code in the extended class.
 2. A class inside a package is called package-member or top-level-class (for package see page 239).
 3. Elements inside a Java-array (for arrays see page 234).

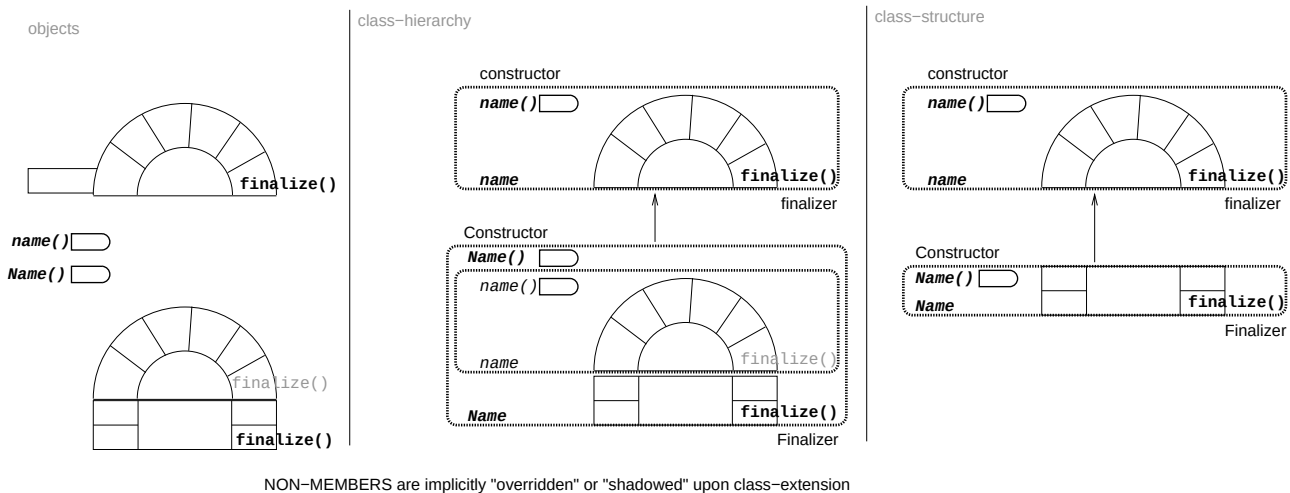
Member-variables of primitive-type, which have been left uninitialized, are given their default values (page 85). (Local variables of primitive-type always have to be initialized explicitly, as already stated past the gray box on page 121.) Constructors are treated extensively in the next section, static-initializers,

⁵¹More on class-methods and class-variables can be found in the TechTip of September 12th, 2000.

members of a
class remain valid
after
class-extension

static-initializers,
instance-
initializers,
constructors and
finalizers are **NOT**
members of the
class

instance-initializers and finalizers are introduced in the section past the next.



Java: Class-Instantiation/Object-Creation

Java: Class-Instantiation/Object-Creation — With entities of class-type, the denotations “instance”, “object” and “variable” often can be used interchangeably. In Java, objects can be instantiated using `new`-prefixed constructors or factory-methods. Thus, objects often can be created or derived from classes by using two different means!

As has been described on page 145, a constructor-call — when prefixed with the `new`-keyword — creates an object and fills the allocated RAM-space with data or does other initialization work and finally returns a reference to that object. For example, a constructor may contain the bulk of the code that defines the class, as shown within the source-code on page 546.

Java: Defining Constructors

If the programmer did not explicitly define a constructor, then each class definition implicitly furnishes a default constructor⁵²:

```
class name1{} // This class has a default constructor name1().
class name2{ name2(){ super();}} /* This class has a custom-constructor,
                                substituting the default constructor, but
                                equivalent to the default one!
                                */
class name3{ name3( int i ){}} // This class has no default constructor!
```

BUT, as can be seen above, if the programmer has explicitly defined a single constructor, then the default constructor becomes unavailable! Thus, a constructor may also be defined explicitly, either by overriding⁵³ the default constructor or by defining a constructor with a non-empty list of parameters. The code-block of the newly defined constructor can be used to enter code like that of a regular method. As already mentioned at the beginning, the constructor’s code-block even may represent an entire program!

Three things make the definition of constructor-methods unique:

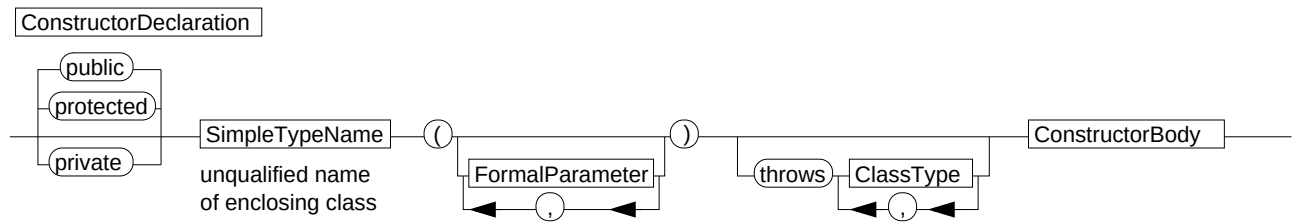
name Constructors always are given the name of the class whose objects they help to instantiate.

modifier and return-type Constructors do indicate their return-type by their name and they are defined without any modifier `static`, this means constructors can be seen as being declared *implicitly* static! In a way constructors resemble static methods given with an implicit return-type. Constructor definitions lack an explicit `return`-clause!

⁵²Here, the method-call `super()`; explicitly instantiates an object of class-type `java.lang.Object`, which is given as the immediate super-class of classes like the kind defined above. Ignore this information that had to be anticipated, or refer to the text around page 166, for an impression of Java’s class-extension mechanism, which uses method-calls like `super()`.

⁵³The overriding of methods has been introduced conceptionally around page 135, in the context of class-extension. Java’s mechanisms for overriding methods are introduced around page 168.

new-keyword By convention, a constructor only can be called when prefixed by a new-keyword (page 145). ONLY TOGETHER WITH THE new-KEYWORD A CONSTRUCTOR ALLOWS THE JAVA PROGRAMMER TO INSTANTIATE A CLASS, that means to produce an object and to return a reference to that object.



ConstructorDeclaration := { *ConstructorModifier* } *ConstructorDeclarator* [*Throws*] *ConstructorBody*

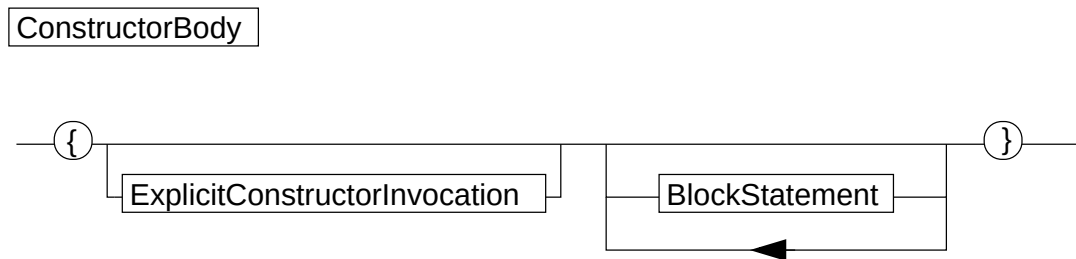
ConstructorModifier := ONE OF public protected private — more on those modifiers, see the sections beginning on page 175.

ConstructorDeclarator := *SimpleTypeName* "(" [*FormalParameter* { , *FormalParameter* }] ")"

SimpleTypeName := *ClassType*

Throws := throws *ClassType* { , *ClassType* } — *FormalParameters* 118, *ClassType* 146, *ConstructorBody* see below.

(See also the *MethodDeclaration* on page 117, there exists a *ResultType*, here NOT!)



ConstructorBody := "{" [*ExplicitConstructorInvocation*] { *BlockStatement* } "}" — *ExplicitConstructorInvocation* 166, *BlockStatement* 105

As already has been indicated, constructors can be defined with a parameter-list. And like methods in general, often constructors are overloaded (page 124) by furnishing different versions with differing parameter-lists. As a first example consider the "Pair"-class from page 76, modified as follows:

```
class Pair2 // overloaded constructor and factory-method
{ int x1, x2;

    Pair2(){ x1 = 0; x2 = 0; };    // default values
    Pair2( int x, int y ){ x1 = x; x2 = y; };
    // above, the constructor with two arguments for non-default cases

    static Pair2 getAPair( int x, int y ){ return new Pair2(x,y); }
    // above, a factory-method
}
```

private
constructors

A CLASS HAVING ONLY **private**⁵⁴ CONSTRUCTORS PREVENTS ITS INSTANTIATION BY METHODS OF OTHER CLASSES. Then, only static methods of the same class ("factory-methods" of the same class) can call the private constructors. Classes like the `java.lang.Math`-class which are used as collections of static methods use a private constructor (declared using a line of source-code like `private Math(){}`) to remain uninstan-
 tiable.

OUTLOOK: For more on constructors see the following section and, especially for the case of constructors that call each other, consider the remarks on page 166, in the context of class-extension. An example of a constructor-callback has been given within the source-code on page 631.

Java: Class-Instantiation with Constructors and Factory-Methods

Java: Class-Instantiation with Constructors and Factory-Methods — Remember the introduction of the notions of declaration, allocation and instantiation beginning on page 75. The instantiation of Java classes

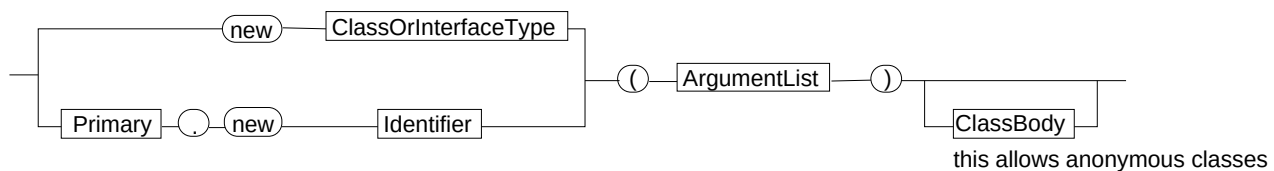
⁵⁴Beginning with page 175, modifiers like `private` are presented.

already has been introduced in various programs (Java source-code on pages 120 and 142) and in the introduction of reference-types (page 145). Recall the simple reoccurring basic formulation:

```
// ----- CONSTRUCTOR-CALL
ClassName objectName;
objectName = new ClassName( optionalArguments );
//or
ClassName objectName = new ClassName( optionalArguments );
```

Now consider a more general version:

ClassInstanceCreationExpression



ClassInstanceCreationExpression := new *ClassOrInterfaceType* "(" *ArgumentList* ")" [*ClassBody*] | *Primary* . new *Identifier* "(" *ArgumentList* ")" [*ClassBody*] — *ClassOrInterfaceType* 143, *ArgumentList* 119, *ClassBody* 146, *Primary* 91, *Identifier* 81

Again: The creation of an object and the returning of a reference to that object is done by using the new-keyword AND the constructor TOGETHER.

Constructors often are overloaded, that means, as required they bear the name of their class, but they have different parameter-lists. See the example given by the extended “Pair2”-class a few paragraphs ago.

Some objects are created and references to them are returned by calling so-called factory-methods, as the one defined in the “Pair2”-class above. Observe that factory-methods need no new-keyword when called.

```
// ----- CALL OF A FACTORY-METHOD
ClassName objectName;
objectName = someFactoryMethod( optionalArguments );
//or
ClassName objectName = someFactoryMethod( optionalArguments );
```

Besides the “getAPair()”-method in the “Pair2”-class above, an arch-example of a factory-method is given by the class-method `Object.clone()` which creates a new object — a copy of the calling object — and returns its reference giving it the type `Object`, for more see page 164.

OUTLOOK: If instantiation remains possible only by using factory-methods, then constructor(s) usually have been declared private. Protected constructors can be only accessed inside sub-classes. (Private- and protected-modifiers are described also on pages 175 and 192.) For more on constructors, especially concerning class-extension, see page 166.

Factory-Method: A factory-method in Java is a method that allocates and instantiates an object and then returns a reference to that object, without the need of being prefixed by a new-keyword. Usually an object is created using a constructor-call, prefixed by the new-keyword. — A factory-method does the same within the context of a simple method-call. Examples for factory-methods are: `Object.clone()`, (page 192) `Class.newInstance()` (page 252) or `InetAddress.getByName(String host)`(page 382) Factory-methods may create objects and return associated references also from other classes and objects. Factory-methods that return instances of those classes in which these factory-methods are declared themselves have to be `static`; otherwise how to get an instance anyway?

What advantage comes with having the two possibilities of creating objects? FACTORY-METHODS CAN CREATE OBJECTS WITHIN THE CONTEXT OF ANOTHER CLASS AND ONE CLASS CAN BUNDLE SEVERAL FACTORY-METHODS FOR OBJECTS OF DIFFERING CLASS-TYPES. And factory-methods can be given more descriptive names than the constructor bearing only the class-name of the given class.

Java: Class- and Object Lifespan

Java: Class- and Object Lifespan — The virtual Java Platform, which is given by the native platform with a running Java Runtime Environment (JRE) — this virtual platform cannot remember each piece of code

that it has ever worked on. Therefore after each new start of the Java platform, classes are transferred from the hard-disk to the RAM and made visible to the Java Runtime Environment. That process is called **loading a class**. Many classes are loaded automatically by the system, others are loaded by request from the user or some Java program, just by trying to instantiate a variable of class-type. (For more on introducing classes to the Java Runtime Environment, see the notes about package installation on page 242 and class-loaders on page 433.)

This section tries to answer the questions: What are the places in a class-definition that can hold programmer-definable source-code? When is the respective source-code used? The class-definition can contain a piece of code, that is worked through directly after the class has been loaded into the Java Runtime Environment; this piece of code is given by the so-called **static-initializer**. Inside the block (statements enclosed by curly brackets) of a class-definition, such a static-initializer is given by another block which is prefixed by the `static`-keyword:

StaticInitializer

— (static) — Block —

directly enclosed by a class declaration

StaticInitializer := `static Block` — Block 105

Consider such a class with a static-initializer:

```
class SelfLogStatic
{
    static{ System.out.println("Class loaded"); }
}
```

Contrary to static-initializers, which are run when the class is loaded into the Java Runtime Environment, **instance-initializers** get worked through each time a class is instantiated. Like the static-initializer, an instance-initializer is given by a block of code that is part of the class-definition. In some way instance-initializers resemble unnamed constructors⁵⁵. — Inside the block of a class-definition, an instance-initializer appears as a simple block of code, which is run each time the class is instantiated (a class is instantiated by a constructor-call or by a call of a factory-method):

InstanceInitializer

— Block —

directly enclosed by a class declaration

InstanceInitializer := `Block` — Block 105

```
class SelfLogStaticInstance
{
    static{ System.out.println("Class loaded"); }
    // above, static-initializer of the class
    //
    // below, instance-initializer of the object:
    { System.out.println("Class instantiated = object created"); }
}
```

How does an instance-initializer differ from a constructor? Instance-initializers are (run AFTER the call of the constructor of the immediate super-class returns, but they are) **run BEFORE the constructor's code is run**. — Again, what comes first instance-initializer or constructor? The sequence is given as follows:

constructor of the super-class (next section) → instance-initializer → constructor of the current class

As already mentioned, the constructor method can be used to run almost any type of code.

A class-definition may also include a `finalize()`-method (with a `public` modifier, a `void` return-type and no parameters):

⁵⁵Next to instance-initializers and constructors, these declarations in a class are not called members (Java's definition of a member can be found on page 148).

```
public void finalize(){... /* if necessary: */ super.finalize(); };
```

(The use of the last `super`-qualified statement is referenced in the text following page 166.) The `finalize()`-method, an instance-method, automatically is called exactly once by the Java Runtime Environment (JRE), before the garbage-collector of the JRE erases the object from the computer's Random Access Memory (page 444). So, where can be positioned code to be run shortly before the object is erased from the computer's Random Access Memory? Into the `java.lang.Object.finalize()`-method.

Besides the `main()`-method, the `finalize()`-method happens to be the second triggerable method introduced so far. (Triggerable-method (page 127) means: Implemented by a Java programmer, but usually called from outside the program by the Java Runtime Environment.) If there was no explicit call of the `finalize()`-method, then the `finalize()`-method is called by the Java Runtime Environment just before deallocating the associated object. The default version of that method does nothing, the overriding method may, for example, close files or terminate network-connections.

OUTLOOK: Exceptions coming from a `finalize()`-method are ignored. The `finalize()`-method runs in a thread of its own. Threads calling the `finalize()`-method do not hold locks so the garbage-collector never has to wait.

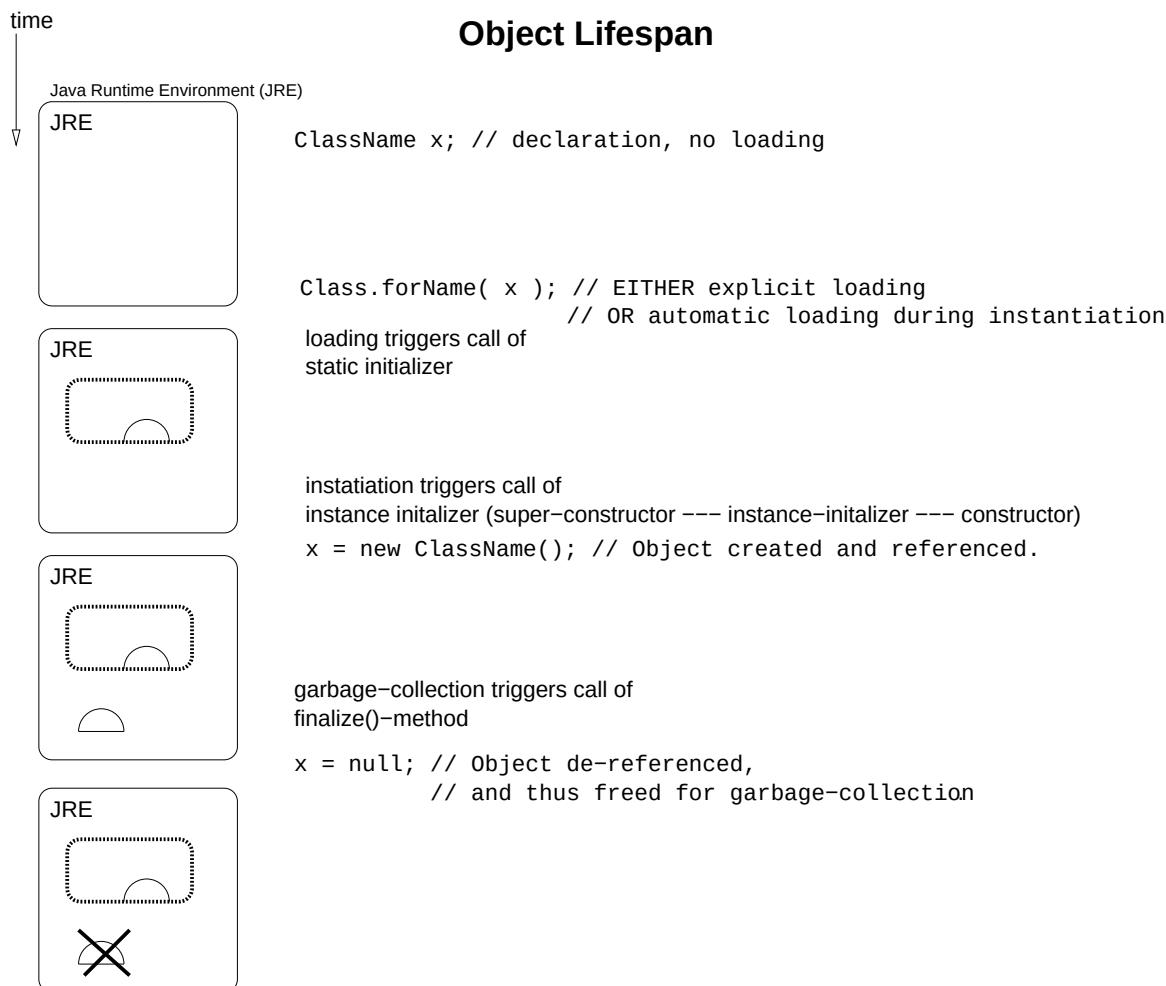
And finally consider the program "SelfLog" that initiates the creation of a "SelfLog..."-object:

```
class SelfLogStaticInstanceFinalize          // class-definition
{
    static{ System.out.println("Class loaded"); }           // static-initializer
    { System.out.println("Class instantiated = object created"); } // instance-initializer
    public void finalize(){ System.out.println("object erased"); } // finalize()-method
}

public class SelfLog
{ public static void main( String unused[] )
  {
    SelfLogStaticInstanceFinalize x;                      // class declaration
    System.out.println( "object declared" );
    try{ Class.forName( "SelfLogStaticInstanceFinalize" ); } // explicit class-loading
    catch( java.lang.ClassNotFoundException e){ e.printStackTrace(); }
    x = new SelfLogStaticInstanceFinalize();                // class allocation/instantiation
    x = null; System.gc();
    // above, preparing to erase the object from the computer's Random Access Memory
    // for(int i = 0; i < 1000; i++){ int q=i^2; }
    // waiting for the object to vanish, unnecessary with some OS's
  } }
/*
--- The instantiation loads the class automatically into the current Java Runtime
    Environment. Nevertheless has been shown how to explicitly load a class without
    instantiating it. Comment out the try, catch and new lines to see that declaring
    an identifier does not load the class.
--- Here has been used the default constructor which is furnished for every class.
--- Waiting for destruction may become unnecessary,
    because the program and the Java Runtime Environment might terminate faster
    than the destruction can be reported.
*/
/*
[localhost:~/JavaCodeFiles/BasicProgrammingI/SCObjectOrientation1] marycosway% javac SelfLog.java
[localhost:~/JavaCodeFiles/BasicProgrammingI/SCObjectOrientation1] marycosway% java SelfLog
object declared
Class loaded
Class instantiated = object created
object erased
[localhost:~/JavaCodeFiles/BasicProgrammingI/SCObjectOrientation1] marycosway%
*/
```

The waiting routine in the program may appear rather simplistic (but effective), a better waiting routine can be found beginning with page 603 (using a class `javax.swing.Timer`). The program above and its development-process are later used to illustrate the class-extension process (page 163).

The entire process has been visualized below:



Java: Introducing and Accessing Class- and Instance-Members

Java: Introducing and Accessing Class- and Instance-Members — The class/object-model introduced instance-related kinds of state and behavior associated with each individual object and class-related kinds of state and behavior associated with the class from which the object was instantiated. The counterparts of these kinds of state and behavior, as specified by the Java language, are instance-variables, instance-methods, class-variables and class-methods (as already presented in the pseudo-code on page 148). The declaration of variables and definition of methods can be done straightforward: **Adding the static-keyword in front of a variable- or method-declaration inside a class makes that member a class-member** (a static variable or a static method). A static member exists within the loaded class until the class is unloaded, that means the static member exists as long as the Java Runtime Environment is running. Instance-members exist only until the object is erased from the RAM by the garbage-collector of the Java Runtime Environment.

```
class ClassType // a class with class-members and instance-members
{
    static Type variable;           // class-variable
    static TypeOrVoid methodName( parameterList ){...} // class-method
    Type variable;                 // instance-variable
    TypeOrVoid methodName( parameterList ){...}       // instance-method
}
```

Without the static-modifier⁵⁶, variable- and method-declarations define instance-variables and -methods. So, in Java, variables and methods are instance members by default.

Again the static-modifier of the main()-method in programs reveals the main()-method to be a class-method; the class itself may be instantiated but the main()-method remains associated with the class.

⁵⁶Modifiers, especially those that concern themselves with variants of accessibility, are treated later extensively, beginning on page 171.

(And, as remarked on page 127, a call of the `main()`-method can be triggered externally by the an end-user entering `java ClassName` into the appropriate terminal-window of the native operating-system.)

But how are the above-mentioned variables and methods accessed from outside the object? Accessing the members of a class can be compounded to the following indication:

```
ClassType object = new ClassType();
object.instanceMember
ClassType.staticMember
```

In the case of the instance members, qualifying⁵⁷ the member-variables or member-methods with the object's identifier gives the desired access. Static members become accessible by qualifying their identifier with the class-name. An example of member declaration and access of the members is given by the set of the following two related classes:

```
/*- The following class stores the number of its instantiations (staticCounter)
    and labels each object with a number (objectNumber).
--- This is done using static and non-static member-variables.
--- The associated code can be found in a static- and a non-static initializer.
--- The output can be done choosing between static or non-static member-methods.
*/
class TheClass
{
    static int staticCounter;                                // class-variable
    static void staticOut( String s, int n )
    { System.out.println( s + " staticOut: " + n ); }
    int objectNumber;                                       // instance-variable
    void instanceOut( String s, int i ){ System.out.println( s + " instanceOut: " + i ); }

    static{ staticCounter = 0;}                             // static-initializer
    { staticCounter++; objectNumber = staticCounter; }      // instance-initializer
}
public class Access
{ public static void main( String unused[] )
  { TheClass obj1,obj2;
    obj1 = new TheClass();
    obj2 = new TheClass();
    TheClass.staticOut( "Number of objects ", TheClass.staticCounter );
    obj1.instanceOut( "no. of obj1: ", obj1.objectNumber );
    TheClass.staticOut( "no. of obj2: ", obj2.objectNumber ); // read access
    obj2.objectNumber = 5;                                   // write access
    obj2.instanceOut( "Test, write access: ", obj2.objectNumber );
  } }
/*
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] marycosway% javac Access.java
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] marycosway% java Access
Number of objects staticOut: 2
no. of obj1: instanceOut: 1
no. of obj2: staticOut: 2
Test, write access: instanceOut: 5
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] marycosway%
*/
```

declaration-
assignment-
sequence

In the source-code, the relative position of the initializers, static and non-static, to the declaration of the counter- and number-variables has to be properly chosen (This is one exception to the statement about arbitrary positioning of declarations inside one block, on page 122).

The procedures of accessing variables and methods can be seen refined and generalized by considering

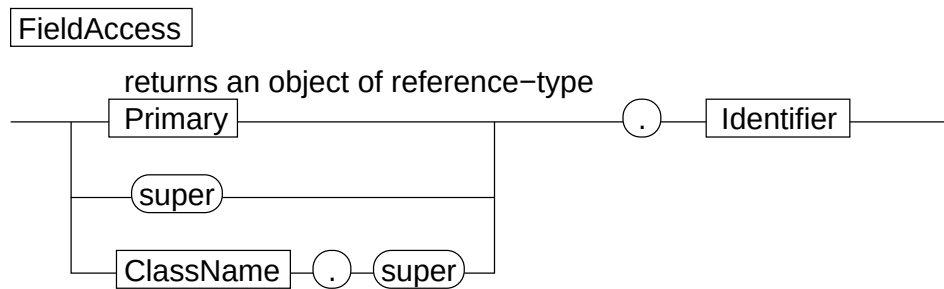
- field-access expressions (elementary examples have been given in the source-code above, in all generality in the syntax-diagram below)
- array-access expressions (page 235) and
- method-call or method-invocation expressions (page 118).

One of the simplest version of an access-expression is given by just using a single identifier; this simple expression is used when accessing a variable or a method from inside an object or a class. A field-access expression fetches the values of variables FROM INSIDE AND FROM OUTSIDE A CLASS OR FROM ITS DERIVED OBJECT. Field-access expressions also fetch static and non-static variables of class-declarations, the so-called class- and instance-variables, which also are called field-variables! Field-access expressions

access from
inside and from
outside

⁵⁷Qualifying means prefixing separated by a period (.).

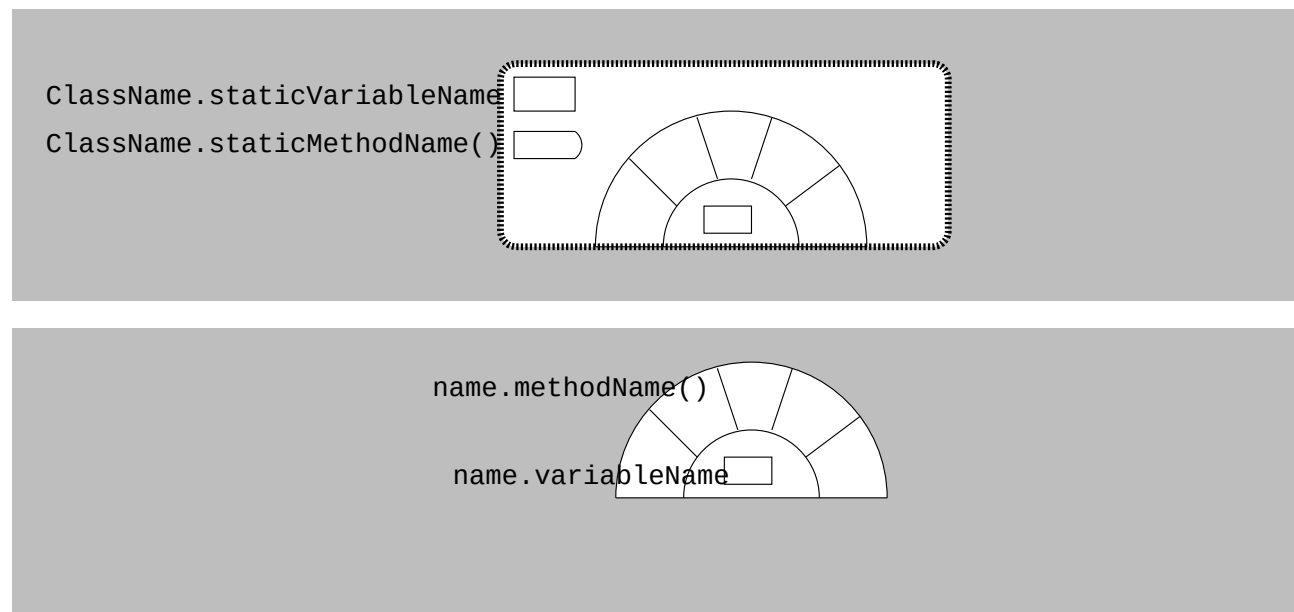
give access to the variable-declarations in the class and its derived objects, as presented above in the class “Access”.



FieldAccess := *Primary* . *Identifier* | *super* . *Identifier* | *ClassName* . *super* . *Identifier* — Primary 91, Identifier 81, ClassName = ClassType 146.

The Primary-expression inside the FieldAccess expression has to produce a reference-type to yield a valid Java-expression. And, as can be seen when referencing the Primary-expression, a field-access expression itself is a Primary expression.

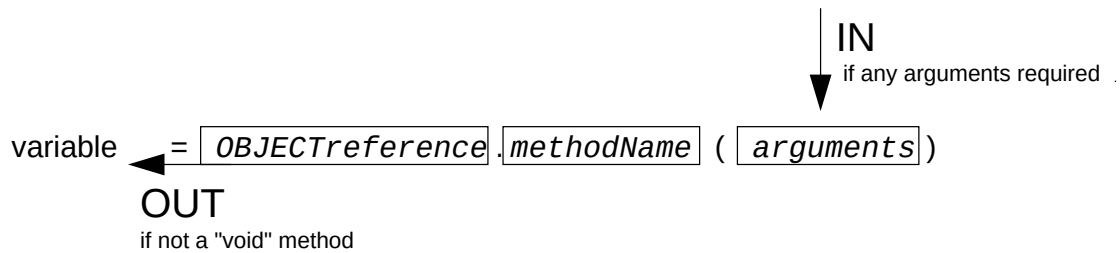
Access from Outside the Class/the Object



Above has been discussed what to do when class-members are to be called from outside the class and from outside the object. How about calling class-members from the inside? As already mentioned, just the identifier suffices for accessing a field from source-code within the class or object. Furthermore there can be used the *this*- and *super*-keywords, but their usage is discussed beginning with page 166 and page 172.

Finally, as a supplement to the chapter about proceduralization, here in the class-context, consider the visualization of the difference between the call of an instance-method and the call of a class-method:

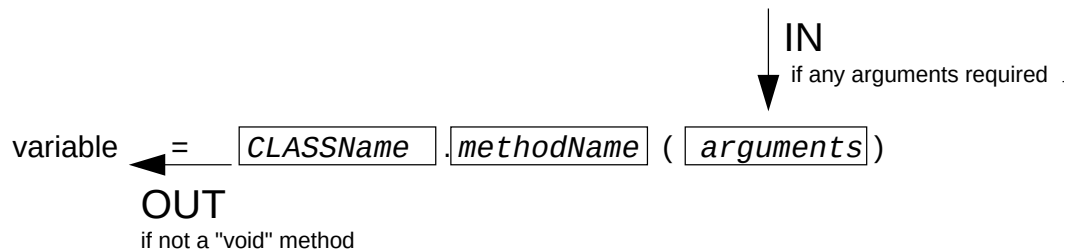
Calling an INSTANCE-method from outside the OBJECT



example: `source` . `addListener` (`drain`)

The sole difference lies in the qualification of the method-name, above with a reference to an object, below with the name of a class!

Calling a CLASS-method from Outside the CLASS



The practical example has been given in the program above, where static- and non-static methods have been declared and called. For a much more extensive presentation with a lot more remarks for fine-tuning source-code, see the ➡ Java Language Specification 2nd Edition around Section 15.11.

Java: Kinds of Classes

Java: Kinds of Classes — The graphic on page 141 shows an object with a member-variable; the member-variable there is an object itself or references an object. This concept already has been applied regularly when using classes as custom-defined types and declaring a member of a class to be of such custom-defined type (for example, consider the references of “TheClass”-type within the “Access”-class of the source-code on page 155).

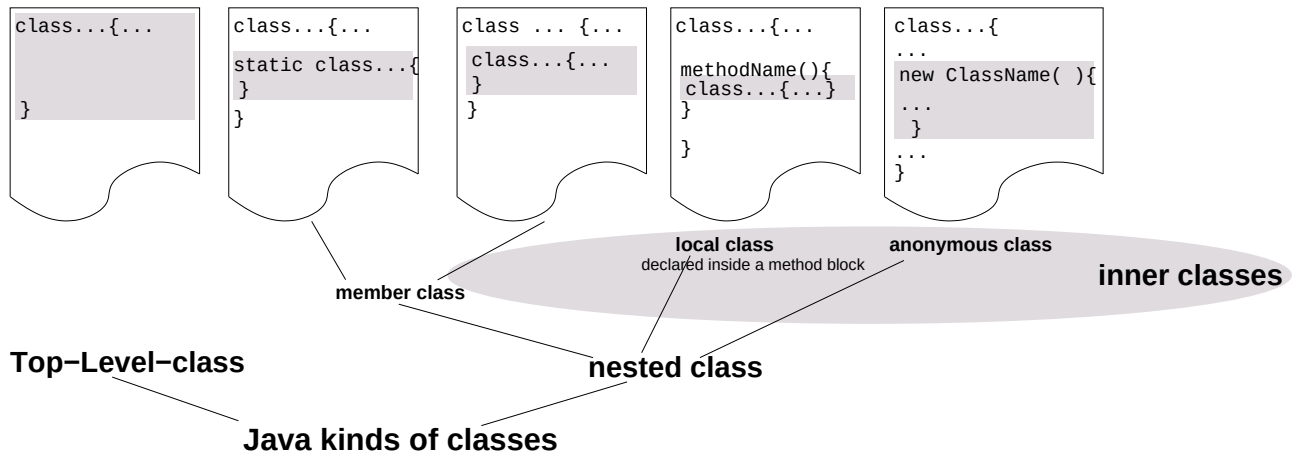
Furthermore, the above-referenced graphic on page 141 is followed by another graphic, which introduced the general idea of defining classes within classes. In Java this concept can be realized in various ways and is given various names as demonstrated in the graphic below.

The Java-classes that have been declared up to now were so-called top-level-classes. A **top-level-class** is a class which has not been declared within any class- or method-declaration. (The set of classes found inside a Java package (page 239) are top-level-classes.)

Any non-top-level-class is called **nested class**, this includes so-called member-classes, local classes and anonymous classes:

A **member-class** (or **member-interface**) is defined directly inside another class or interface, declared as a regular class-member (recall the member-definition on page 148).

Non-static nested classes are called **inner classes**. Anonymous classes are treated in the next section. **Local classes**, are named classes declared inside code-blocks of methods, either directly or within other classes locally defined there. Local classes are not members of classes. On the level of typing, local classes are the equivalent to local variables.



Top-level-classes are also used as programs (with a static `main()`-method). To avoid unnecessary complexities, inner classes defy static fields (variables, methods, classes, initializers) — inner classes are free of static declarations of the kind: static variables, static methods, static classes or static-initializers.

Consider some access characteristics of instance-variables in member-classes: First consider an instance-variable inside a static member-class:

```
class TopLevelClass
{ static class StaticMemberClass{ int instanceVariable } }
//
// somewhere in another class:
TopLevelClass.StaticMemberClass instance =
    new TopLevelClass.StaticMemberClass();
instance.instanceVariable = 1;
```

As the second case consider a non-static declaration of a class as a member of another class, making it a so-called inner class⁵⁸:

```
class A{ class B{ int c } };
// somewhere in another class:
A a = new A(); A.B b = a.new B(); b.c = 1;
```

The scope of member-classes is given by the class-body in which they have been defined. Local classes are declared and exist only in a block of code, usually that of a method. *Local classes never have static members*. The scope of local classes is the block in which they are defined.

Here a demonstration of several ways of defining a class inside a class:

```
class TopLevel    //// ----- a top-level-class together with nested classes, like:
{
    static void staticMethod( String s )
    { System.out.println( "static method called from " + s ); }
    void instanceMethod( String s )
    { System.out.println( "instance-method called from " + s ); }

    static class StaticNested    //// ----- a statically nested class, a static member class
    { StaticNested()
        { System.out.println( "StaticNested class instantiated" );
          staticMethod( "Nested Top Level class" );
        } }

    class NamedInner            //// ----- a non-static member-class, an inner class
    { NamedInner()
        { System.out.println( "NamedInner class instantiated" );
          TopLevel.this.instanceMethod( "NamedInner class" ); // access to enclosing instance
          TopLevel.staticMethod( "NamedInner class" );
        } }

    public static void main( String unused[] )
    {
```

⁵⁸Test these code fragments by inserting them into the program below: The top-level-classes may just be appended to the text-file and the parts preceded by “// somewhere in another class” may be positioned inside the `main()`-method, maybe adding a print-out (`System.out.println( instance.instanceVariable + b.c );`).

```

class Local                                //// a local class within a method
{ Local(){ System.out.println( "Local class instantiated" );}
}
new StaticNested();                        /// an anonymous object:
                                           /// Useful if only instantiation necessary,
                                           /// and no further reference needed.

TopLevel theInstance = new TopLevel();
theInstance.new NamedInner();              /// anonymous member object in theInstance
new Local();
staticMethod( "main()-method" );
theInstance.instanceMethod( "main()-method" );
} }
/*
[localhost:~/JavaCodeFiles/BasicProgrammingI/SCObjectOrientation1] marycosway% javac TopLevel.java
[localhost:~/JavaCodeFiles/BasicProgrammingI/SCObjectOrientation1] marycosway% java TopLevel
StaticNested class instantiated
static method called from Nested Top Level class
NamedInner class instantiated
instance-method called from NamedInner class
static method called from NamedInner class
Local class instantiated
static method called from main()-method
instance-method called from main()-method
[localhost:~/JavaCodeFiles/BasicProgrammingI/SCObjectOrientation1] marycosway%
*/

```

HINT: The Java compiler transforms the nested-, member- and local classes into regular classes as can be seen when finding the list of additional class-files in the folder of the program's class-files. As a rule, a Java programmer should use the complexities the Java language furnishes only if they make the programs easy to read or if they are particularly suited to the problem! Otherwise the program becomes difficult to read (as seen above) and the Java compiler produces a lot of, possibly unnecessary, extra code. Therefore it may prove useful to know something about nested classes and it may turn out equally useful to avoid using them, if possible.

Another kind of classes, the so-called adapter-classes, which relate to nested classes, are introduced in the grayed box on page 190.

Java: Anonymous Objects and -Classes

Java: Anonymous Objects and -Classes — When a class has been defined without assigning it a name, then such a class is called an anonymous class. Since an anonymous class has no name, such class can only be of any use if instantiated and processed within the same expression of its definition. Usually the object instantiated from that class also remains anonymous.

Anonymous classes and their instances may occur as arguments of a method or in the return-clause of a method, (or as threads or as thread implementations, page 208) generally they may occur within any allocation expression.

```
OriginalClass(){ extensionCode } // an anonymous class
```

Anonymous objects occur in return-expressions, in arguments, even as a single standing statement in the case of a thread-instantiation (source-code on page 204). – An anonymous object may even be instantiated only to issue the method-call of the constructor-method (see source-code below). – Some screen-windows on the computer's monitor have to be produced only once without further references, in those cases the window-objects usually are instantiated anonymously. This reminds the application-programmer of the ephemeral character of those objects. For an example see the source-code on page 527, there the component onto which the image is drawn remains anonymous, whereas the named window-object is referenced at least two times.

```
new ClassName() // anonymous object
new OriginalClass(){ extensionCode } // anonymous object from an anonymous class
```

Anonymous classes have no constructors, no static members. Therefore objects instantiated from anonymous classes have to be furnished with data by instance initializing, just because there are no constructors to handle that initialization. How can objects be instantiated without using a constructor or having a name? An allocation expression alone gives an anonymous object, an allocation expression with a non-trivial class-body⁵⁹ gives an anonymous class which usually also has to be instantiated into an anonymous object:

⁵⁹This can be seen as a very special kind of class-extension, which is treated explicitly later (page 163).

```

/* demonstration of anonymous objects and anonymous classes */
class NamedClass
{ NamedClass(){ System.out.println( "Constructor-call" ); }
  public static void main( String unused[] )
  {
    // anonymous OBJECT, derived from a named class:
    new NamedClass();

    // anonymous CLASS, because it extends the named class with additional code,
    // instantiates an anonymous object of its type
    // (the following is an allocation expression with non-trivial class-body)
    new NamedClass()
    {
      { // instance-initializer can supplement the role of default constructor
        System.out.println( "a truly anonymous class instantiated" );
      }
    };
    System.out.println( new String( "finally an anonymous String-object" ) );
  } }
/*
--- Two constructor-calls are seen below,
    because two anonymous objects are instantiated with the same constructor.
--- The second time the named class is extended by the code making it truly anonymous.
*/
/*
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] marycosway% javac NamedClass.java
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] marycosway% java NamedClass
Constructor-call
Constructor-call
a truly anonymous class instantiated!
finally an anonymous String-object
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] marycosway%
*/

```

As another example, of considerable practical importance, note again the introduction to anonymous threads on page 208.

Singleton

Singleton — There are cases in which a class should produce only one instance per Java Runtime Environment⁶⁰. Those objects are called singletons. For an example (the Java security-manager) see the footnote on page 463. How can a Java programmer design a class so that it yields only one instance? In a three step procedure: By producing one private instance on loading the class, then inactivating the default constructor by overriding it with an empty and private constructor-version and finally by furnishing a public `getInstance()`-method, that retrieves a public reference to the only existent instance:

```

public class Singleton
{ private static final Singleton theInstance = new Singleton();
  private Singleton(){}
  public static Singleton getInstance(){ return theInstance; }
}

```

The declaration as `final` fixes the reference to the actual object and disallows the redefinition of the identifier “theInstance” onto another object. The example has been taken from the site <http://java.sun.com> (search for the word “singleton” and consult the list of questions).

The Java Class-Model Recalled

A Java object is a collection of associated variables and methods. Such an object is an instance of a Java class. The Java class defines the associated variables and methods. Methods collect statements and control structures that manipulate the variables.

Classes can be seen as generalized and customizable type-constructs. A class with a `main()`-method qualifies as the core of a piece of standalone code, namely a program.

Where, other than in a `main()`-method, can be found the Java code that contains the program-statements? Statements can be found in (non-) static-initializers of the classes and in methods, especially constructors. Later, in multithreading, Java code with program-statements also can be found in `run()`-methods that run (quasi-)parallel. How do Java objects communicate with each other? By calling each others methods!

⁶⁰Why not use the class itself with class-variables and class-methods? A class, for example, cannot fit into a serializable object-graph (page 192) or into a component-model (page 311).

Classes, with a `main()`-method or with a constructor containing a non-trivial flow of commands can be seen as programs, or — thinking of possible method-calls into other classes — the core of a program. But only the classes with the `main()`-method allow the external start-mechanism of letting an end-user enter the “`java ClassName`”-command in a terminal-window of the native operating-system.

Comparing Objects

Comparing Objects — An object in Java always is of reference-type, this means that if one of its identifiers is reassigned, this identifier then points to some other object, as described beginning with page 143. So the Java-identifiers make it possible to refer to an object, but they do not fix the identity of the Java object. Therefore, before trying to compare objects, it may be useful to be able to assert their identity.

Java: Identity of Objects

Java: Identity of Objects — The *state* of a Java object is given by its instance-variables, its *behavior* is given by its instance-methods. The object can be addressed by identifiers of proper reference-type. Members of that object can be addressed by using an identifier of the object qualifying the name of the respective member. But since each identifier can be reassigned to reference another object, identifiers cannot serve as identity-tags for the object. — Does Java supply some equivalent to an *identity*-tag, as introduced on page 131? Yes, the Java-system-specific identity-tag of each object, existent in a Java Runtime Environment, is given by the `int`-value returned by a `hashCode()`-method. That method can be called for each object within a Java Runtime Environment:

```
public class ObjectIdentity
{ public static void main( String unused[] )
  { Object ref = new Object();
    System.out.println
      ( "ref.hashCode() = " + ref.hashCode() + // identity-tag, an int-value
        "\n ref.toString() = " + ref.toString() // string with information about the object
      );
  } }
/*
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro% javac ObjectIdentity.java
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro% java ObjectIdentity
ref.hashCode() = 135120904
ref.toString() = java.lang.Object@80dc808
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro%
*/
```

The two methods `hashCode()` and `toString()` are instance-members of the class `java.lang.Object`. For more on the `Object`-class, that underlies all Java-objects, see page 164. The un-overridden `toString()`-method can be used too for asserting the object's identity, but this method often is overridden.

Means for CHECKING WHETHER TWO IDENTIFIERS POINT TO THE SAME OBJECT can be listed as follows:

== First the equality operator (`==`) used with two references returns a boolean value answering the question “Do the references point to the same object?”.

hashCode() Second the `hashCode()`-values of primitive-type `int`, compared with each other answer the question of identity.

toString() In the third case, the strings of an un-overridden `toString()`-method can be used to identify an object⁶¹.

The name of a variable of primitive-type represents the information about its identity. The identity of an object (which is always of reference-type) has to be assessed by comparing the `int`-values of the object's `hashCode()`-method.

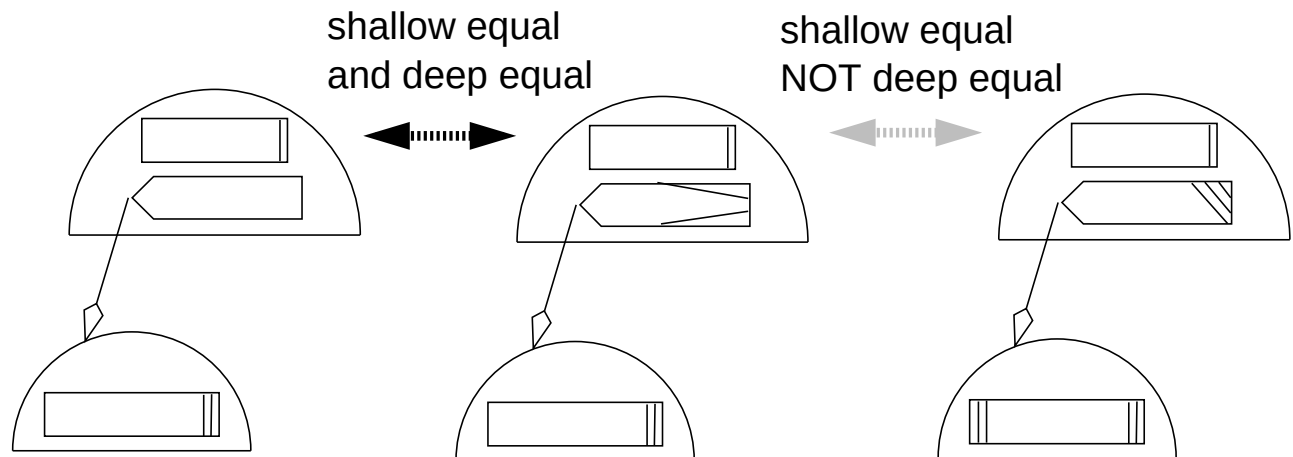
Java objects themselves can contain variables of reference-types, (graphic on page 141) and thus the object may exist within an entire graph of references (object-graph), as indicated in the graphic following page 192. This makes it possible to widen the definition of equality for objects:

⁶¹The use of the equality operator (`==`) proves to be the safest alternative, because the two methods (`hashCode()` and `toString()`) may have been overridden by a programmer, yielding unexpected return-values.

Concept: Equality of Objects

Concept: Equality of Objects — Remember the comparison operators for variables of primitive-type, which have been presented around page 96. Variables of reference-type, especially those of class-type can be considered much more difficult to compare, due to their potentially complex inner structure.

Remember the graphics on/or following page 143: Two references having a differing name may refer to the same object. Or two references of differing name may refer to different objects, but those different objects may contain the same data or different data. — Since variables in objects can be of reference-type, and these three cases may arise within an object, the concept of comparing objects with each other can diversify significantly. One of the simpler conceptual cases is presented below (The iglu-like structures again represent objects. They may contain rectangles representing variables of primitive-type. And they may contain pointed rectangles that represent the identifiers of variables of reference-type.):



In the graphic, all three objects, to be compared, contain the same variable of primitive-type, but three different references. Sameness on the level of primitive-type is called **shallow equal**. Only two of these three objects contain references (differently named, but) to same secondary objects; this kind of comparison, is a case of **deep equality**. Imagining entire reference graphs can introduce a wide variety of more or less rigid forms of deeper- or shallower equalities, between the pure concepts of shallow equal and fully deep equal.

OUTLOOK: Copying objects with references gives similar freedom like comparing them. Copying an object may not involve all levels of references, this will be called shallow copying or cloning. A completely duplicated object, which includes the duplication of all referenced objects is called **deep copy**. Clones are always shallow equal. (For Java copying and cloning see page 191)

Java: Comparing Objects (1)

Java: Comparing Objects (1) — Consider two identifiers (name, name2) of reference-type as introduced by the graphic on page 143. In the first graphic the identifiers name and name2 reference the SAME object, exactly then the statement name == name2 is true:

```
ReferenceType name, name2;
name == name2
// is TRUE when "name" references the SAME data-chunk as does the "name2".
```

Of course, then the data referenced by both identifiers is the same.

But reconsider the graphic on page 143 and now consider the graphic that follows the aforementioned. Here the two identifiers point to different chunks of data (different objects), which contain the same information, so-called copies or clones. Now, the previous test produces a different result:

```
name == name2
// is FALSE when "name" and "name2" reference DIFFERENT PIECES OF DATA,
// even if these contain the same information!
```

Only with variables of primitive-types the VALUES of the variables are tested for equality by using the equality operator (==). **WITH VARIABLES OF REFERENCE-TYPE THE POINTER-INFORMATION OF THE IDENTIFIERS IS TESTED FOR EQUALITY WHEN USING THE EQUALITY OPERATOR (==).** Therefore it can be useful to write own versions of methods for comparing objects in a specific context (a Java programmer might call these methods for example: deepEqual() and shallowEqual()), much like the equals()-method furnished with the String-class, as presented below:

```

class Equality
{ public static void main( String unused[] )
{
    String ref1, ref2, clone;
    ref1 = new String( "Hello World!" );    // allocation and instantiation
    ref2 = ref1;                            // assignment of reference
    clone = new String( ref1 );             // allocation, instantiation
                                           // of an object with the same content.

    if( ref1 == ref2 )
        System.out.println( "References point to same object" );
    if( ref1 != clone )
        System.out.println( "References to different objects (here containing the same content)" );
    if( ref1.equals(clone) )
        System.out.println( "asserts the equality of the content (even if objects are different)" );
} }
/*
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro% javac Access.java
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro% java Access
References point to same object
References to different objects (here containing the same content)
asserts the equality of the content (even if objects are different)
[localhost:~/JavaCodeFiles/Basics1/SCObjectOrientation1] andreadipietro%
*/

```

A more extensive description of the `equals()`-method give the sections about deep- and shallow comparison and copying. For more on the associated copying and cloning processes see page 192.

The following code fragment tests for EQUALITY OF CONTENT: In the first case for a variable of primitive-type, in the second case for the variable of reference-type:

```

int i;    // variable of primitive-type
Object o; // variable of reference-type
if( i==1 && o.equals( new String("x") ) );

```

Comparing object-references with the equality operator (`==`) means testing whether the references point to the **SAME** object, this also can be seen as a test for identity⁶². (The type `java.lang.Object`, which also can be used shortened to `Object`, represents the most general class-type imaginable in Java. This `Object`-type occurs naturally in the next section when introducing the notion of “class-extension” in Java, see page 164.)

The equality operator (`==`) tests variables of primitive-type for equality of values, whereas variables of reference-type, references, are tested for identity of the referenced object.

`==` tests equality
for primitive-type,
tests identity for
reference-type

Java: Class-Extension

Java: Class-Extension — Remember the successively extended classes `SelfLogStatic`, `SelfLogStaticInstance` and `SelfLogStaticInstanceFinalize` from the section about the lifespan of objects (beginning with page 152). The source-code of the first class has been successively extended. — Remember also the definition of an anonymous class which extended the non-anonymous “base”-class (page 159). — These elementary programming examples already prepare Java’s implementation of the concept of class-extension. Remember also that the concept of class-extension already has been introduced in the wake of presenting the object-model on page 131. The basic idea can be reformulated: There is given an initial class with a functionality to be extended, not by rewriting the entire class but by supplementing code pieces which enhance the basic functionality. This is made possible by using the `extends`-keyword of Java:

```

class SelfLogStatic
{ static{ System.out.println( "Class loaded" ); } }

class SelfLogStaticInstance extends SelfLogStatic
{ { System.out.println( "Class instantiated = object created" ); } }

class SelfLogStaticInstanceFinalize extends SelfLogStaticInstance
{ public void finalize(){ System.out.println( "object erased" ); } }

public class Trigger
{ public static void main( String unused[] )

```

⁶²The expression `"x"=="x"` asserts the identity of two DIFFERENT `String`-objects and therefore should return the boolean value `false`. Nevertheless some Java compilers may return `true`. To assert the equality of the content rather use the, if necessary overridden, `equals()`-method, which definitely lets the expression `"x".equals("x")` return the boolean value `true`.

```

    { SelfLogStaticInstanceFinalize x;
      x = new SelfLogStaticInstanceFinalize();
    } }

/*
jacobneuhau@linux:~/JavaCodeFiles/Basic1/SCObjectOrientation2 > javac Trigger.java
jacobneuhau@linux:~/JavaCodeFiles/Basic1/SCObjectOrientation2 > java Trigger
Class loaded
Class instantiated = object created
jacobneuhau@linux:~/JavaCodeFiles/Basic1/SCObjectOrientation2 >
*/

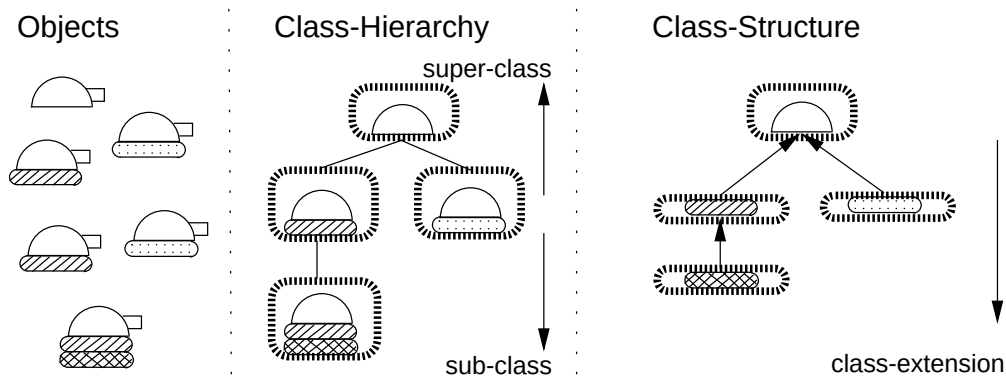
```

This program does essentially (without the waiting-routine) the same as the program on page 153. Earlier steps of development need not be completely re-written to integrate them into the extension, as it was the case earlier. In this elementary context the previous argument (“no re-writing”) may sound somewhat lame, but in the context of a proper analysis of the object-structure of a larger task, this possibility of incremental development *can* be of great advantage. This is the case especially then, when the design of the class-structure has been devised generally enough to anticipate future needs of extension. Properly arranged, the undeniable overhead of coding the extensions *can* result in practical advantages:

CODE REUSE the arrangement in an index-structure may save time and memory to write analogous source-code.

STRUCTURE the whole code can be broken into an hierarchically indexed class-structure

Reconsidering the class/object-model; in Java, **the Java programmer develops a class-structure** in the Java source-code. Due to its importance the graphic which illustrates the three notions “objects”, “class-hierarchy” and “class-structure” of page 135 is reprinted here:

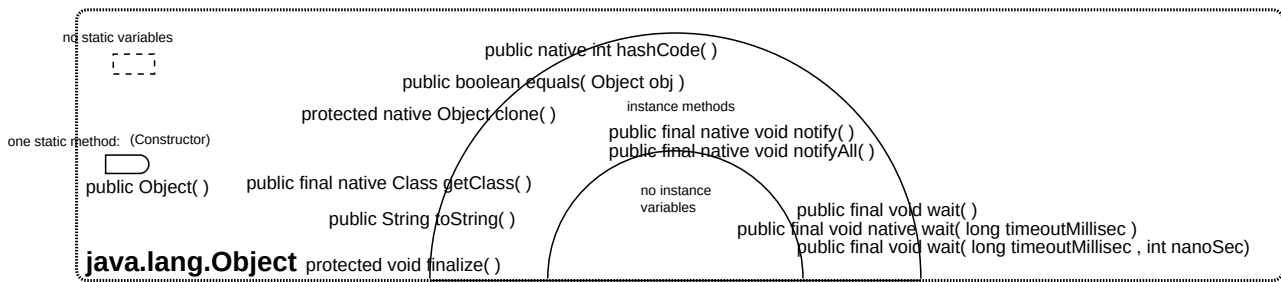


When writing source-code in class-structures, classes of the class-hierarchy have to be reassembled in the mind of the programmer by following the `extends`-keywords in the source-code and collecting the class-members. Thus, the keyword `class` in Java followed by the keyword `extends` displays a piece of a class-structure that the Java compiler transforms to a class. Most documentations like the  Java Platform API Specification [JPAPIS] or pre-written class-libraries or the Java Developers Almanac are organized economically according to given class-structures. Of course, the derived objects are instances of the classes of the hierarchy.

The Class `java.lang.Object`

The Class `java.lang.Object` — In Java only exactly one single⁶³ class-name can follow the `extends`-keyword, otherwise the hierarchy would have multiple starting-points. Thus in Java there can exist only one ultimate super-class. — The super-class of all classes in Java is given by the `java.lang.Object`-class. The `Object`-class has no super-class. All objects derived from pre-defined- or programmer-defined-classes implicitly are sub-classes of the `Object`-class. When a Java programmer defines a class without an `extends`-clause, then the definition can be thought to bear an invisible “`extends Object`”-clause. A Java programmer notices this, mainly by encountering methods like `equals()` and `finalize()` with each instantiated class! For more on the `Object`-class in the context of type-casting objects or their references, see page 169, and on cloning objects see page 192. A short revision on the class `Object` is given on page 252.

⁶³Tautological to stress the importance.



The graphic above shows the signatures (plus return-types and modifiers) of the various methods found in the class `java.lang.Object`: A constructor, implicitly static, as a class-method, but not a class-member. The already introduced `hashCode()`-, `equals()`- and `clone()`-methods. As well as the `finalize()`- and `toString()`-methods⁶⁴. The methods `getClass()`-, `notify()` and the overloaded `wait()`-methods will be introduced later.

Class-Extension: Member-Access

Class-Extension: Member-Access — Above has been shown how to extend a class (`extends`-keyword in class-declarations facing the class “Trigger” on page 163). How can be accessed the members of extended class? As a part of a class-hierarchy, an extended class is a class of its own, its members are accessible in the same way as described previously with non-explicitly extended classes:

```
class Original
{
    int aVariable; // instance-variable
    int aMethod(){ return aVariable; } // instance-method
    void out( String s ){ System.out.print(s); }
}
class Extension extends Original
{
    int anotherVariable; // instance-variable
    void anotherMethod( String s ){ out( s ); } // instance-method
}

class ElementaryAccess
{ public static void main( String unused[] )
{
    Extension e;
    e = new Extension();
    e.anotherVariable = 11;
    e.aVariable = 10;
    e.anotherMethod("anotherMethod-call\n");
    e.out
    ( "e.aVariable = " + e.aVariable
    + "\ne.anotherVariable = " + e.anotherVariable
    + "\ne.aMethod() = " + e.aMethod()
    + "\n"
    );
} }

/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > javac ElementaryAccess.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > java ElementaryAccess
anotherMethod-call
e.aVariable = 10
e.anotherVariable = 11
e.aMethod() = 10
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 >
*/
```

⁶⁴Standard debugging- or informational uses for overriding methods of the class `Object` may suggest themselves:

```
public String toString()
{ return "nameOfInstanceMember = " + nameOfInstanceMember;
}
```

and for example

```
protected void finalize()
{ System.out.println("object no." + hashCode() + " is erased from RAM.");
}
```

For more on the `toString()`-method see also the TechTip of February 5th of 2002.

So, that what is to be known about accessing members in extended classes has been told in the section on accessing class- and instance-members, beginning on page 154. (This free accessibility of class- and instance-members may be restricted, due to shadowing, overriding methods, type-casting, un-shadowing, un-overriding, using access-modifiers, as described in the following sections.)

Those declarations within a class that become part of the sub-class are called **class-members**⁶⁵, a natural effect when considering the composition of a class-hierarchy from a class-structure, as visualized by the graphic on page 135. But static-initializers, instance-initializers⁶⁶, constructors and finalizers are NOT members of the class and therefore NOT inherited by sub-classes! So constructors and finalizers have to be reconsidered in the context of class-extension:

Java: Class-Extension with Constructors and Finalizers

Java: Class-Extension with Constructors and Finalizers — Using constructors for instantiating a class (=creating an object and returning a reference to this object) has been shown, beginning with page 145. How a Java programmer can define constructors has been described on page 149. Finalizers have been introduced on page 152. — Member-variables and member-methods of super-classes⁶⁷ are inherited, but *constructors and finalizers are not inherited* (see the graphic on page 149). Nevertheless, constructors of a sub-class implicitly call the constructor of the super-class, if no explicit call of the kind `super()`, `super(...)`, `this()` or `this(...)` is found at the beginning of the constructor's block.

If a class-definition lacks an explicit definition of a constructor, then the invisible default constructor

```
ClassName(){super();}
```

is called automatically (remember the footnote on page 149). The constructor-call statement of the kind `super()` calls the constructor of the super-class. These automatic constructor-calls iterate, until the constructor of the ultimate super-class in Java — `java.lang.Object` — is reached. The default constructors have the modifier of their enclosing class (A public class has a public constructor); this can be changed by customizing the constructor when declaring it explicitly.

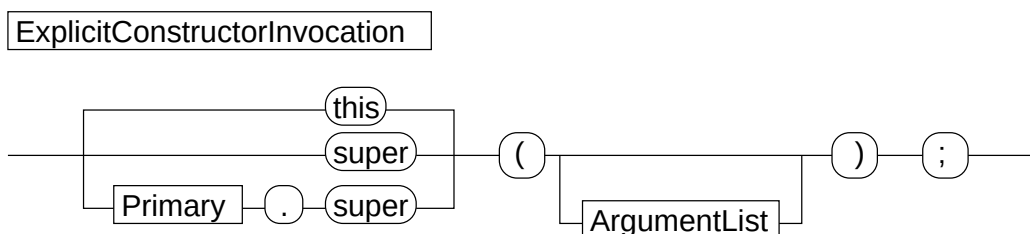
When *declaring constructors explicitly with non-trivial parameter-lists*, two cases may be discerned: The first statement is no explicit constructor-call, then the default constructor of the super-class is called implicitly and automatically at the beginning.

```
ClassName( optionalParameters ){ no explicit call ... } // same as
ClassName( optionalParameters ){ super(); ... }
```

An explicit constructor-call either with a `super()`- or a `this()`-call needs no implicit helping:

```
ClassName( optionalParameters ){ super( optionalArguments68 ); ... }
```

The `super()`-call always has to be the first statement in the block of source-code which implements the constructor. This has been made explicit by the ConstructorDeclaration on page 150. As an example of constructors calling constructors of a super-class consider the program below; there can be seen how a constructor-call results in an explicit constructor-call chain.



ExplicitConstructorInvocation := (`this` | `super` | `Primary . super`) "(" [*ArgumentList*] ")" — Primary 91, *ArgumentList* 119. (Used in *ConstructorBody* on page 150)

⁶⁵This is the defining criterion for being a (class-)member (as defined on page 148).

⁶⁶The timing-relation between constructors and instance-initializers (super-constructor → instance-initializer → current constructor) has been described around page 152.

⁶⁷Remember that those classes, that have been extended, often are called super-classes; their extensions consequently are called sub-classes.

⁶⁸Remember, parameters are found in method-declarations or -definitions, arguments are found in method-calls. In contexts outside of this text, these notions may be used interchangeably.

How about constructors calling each other within the same class-context? Constructors can call each other by using the explicit constructor-call statement, `this(...)`, as the first statement in the constructor-block:

```
ClassName( optionalParameters ){ this( optionalArguments ); ... }
```

Both cases from above are exemplified in the following program:

```
/* constructor chaining */
class Original
{
    void out( String s ){ System.out.print(s); }          // called by BOTH 'Original()' -methods
    Original(){ out( "Original constructor called " ); } // called by 'Original( String s )'
    Original( String s ){ this(); out( s ); }             // called by 'Extension()'
}
class Extension extends Original
{ Extension(){ super( "from constructor of Extension \n" ); } }
class ConstructorChain
{ public static void main( String unused[] )
  {
      new Extension(); // anonymous object
  } }

/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > javac ConstructorChain.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > java ConstructorChain
Original constructor called from constructor of Extension
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 >
*/
```

An example of constructor callback is hidden on page 631.

Finalizers have to be redefined in each class-extension. If found useful, a Java programmer can establish a finalizer calling-chain as indicated on page 152, which refers to the `finalize()`-method of the super-class by qualifying the method with the `super`-keyword. (The default `finalize()`-methods do nothing).

Java: Overriding Methods and Shadowing Variables (2)

Java: Overriding Methods and Shadowing Variables (2) — (A precursor of the following has been introduced with the source-code, on page 124, considering shadowing effects with local variables of methods.) Consider a class-extension like the one introduced in the previous program. What happens if the variable-name and the method-signature were chosen to be identical to their counterparts in the super-class? So, what happens when members of the extensions use the same identifiers or signatures as the members of the extended class? Consider the suitably modified source-code and its output:

```
/* override and shadow */
class Original
{
    int aVariable = 1;                // instance-variable
    int aMethod(){ return aVariable; } // instance-method
    void out( String s ){ System.out.print(s); }
}
class Extension extends Original
{
    int aVariable = 10;               // overshadows the Original variable
    int aMethod(){ return 3*aVariable; } // overrides the Original aMethod()
}

class OverrideAndShadow
{ public static void main( String unused[] )
  {
      Extension e = new Extension();
      e.out
      ( "e.aVariable = " + e.aVariable
        + "\ne.aMethod() = " + e.aMethod()
        + "\n"
      );
  } }

/* -- Effect: The variable-name and method-signature reference the definitions
              of the extended class,
              the definitions of the original class are not used!
```

```

*/

/*
marycosway@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > javac OverrideAndShadow.java
marycosway@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > java OverrideAndShadow
e.aVariable = 10
e.aMethod() = 30
marycosway@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 >
*/

```

From the output above can be inferred: If methods (method-signature) or variables (variable-name), which are defined in a class-extension, have the same method-signature or variable-name as members of the super-class, then STANDARD REFERENCING ACCESSES ONLY THE MEMBERS OF THE EXTENSION. This is often communicated in the following way: **A method in a sub-class overrides⁶⁹ a method of a super-class**, in the case they both have the *same* signature (*same* name AND *same* parameter-list).

overriding

overshadowing **A variable defined in the sub-class overshadows another variable in the super-class**, if both have the same identifier⁷⁰.

The same, what has been said above, is true for static variables and for static methods⁷¹. An illustration and conceptual description of the process of overriding has been already given on page 135.

Remember also the similarity to the shadowing effects evoked with local variables and parameters of methods (page 123).

Overriding a Method: Overriding a method means to declare a method with the **SAME NAME AND THE SAME PARAMETER-LIST** when declaring a sub-class of a given class. The Java compiler does not complain, but uses this newly defined method henceforth when working with this sub-class or instances of it. Within this level of class-extension and within all further extensions, the overriding method substitutes the overridden method. Example: The `clone()`-method of the `java.lang.Object`-class usually is (publicly) overridden to serve the specific needs of the sub-classes. (Overloading a method means declaring a new method with the **same** name and a **different** parameter-list.)

The Java compiler can be advised to reject the overriding of a method by declaring the method `final`. Declaring methods `final` also can be used to make the method-calling faster: The use of a `final`-modifier binds the method-signature directly to its code; so there exists no overhead by a run-time look-up process of the interpreter. This can be reformulated as “suppressing the late-binding of methods” or “switching from late-bound methods to early-bound ones”.

early-bound/late-bound methods

Early-bound method, late-bound method: Early- and late-binding describes the way compilers treat method-calls. In early-binding the compiled code does not have to search the appropriate method, because the connection has been already fixed during compilation. (Somewhat like in the case of the naming of primitive-types in Java, see the graphic on page 83.) This makes method-calls fast. — In a late-binding language, a method-call is connected to the method at run-time, not at compile-time (somewhat like the programmer can change the names of variables of reference-type, page 143). With a late-binding language, like Java, the Java compiler allows the overriding of methods. Late-binding of a method’s identifier to the method’s code takes more time but adds flexibility, like the option of overriding methods.

The `final` modifier is the switch that makes a method early-bound, instead of leaving it late-bound. Only late-bound methods can be overridden on class-extension, `final` or early-bound methods reject any attempts to override them.

overriding has to extend accessibility!

Methods which override others always have to be equally or more accessible than overridden methods: A public method can override a method with default scope, such a method can override a protected method. The modifier keywords `public`, `protected` and `private` are described in one of the next sections about scoping, beginning with page 175.

The overriding effect can be circumvented only for static methods (☞☞☞ Java Language Specification 2nd Edition Subsubsection 8.4.8.5) by a widening-cast. A process called “widening-cast” is introduced in the section below.

⁶⁹An overloaded method has the same name(=identifier) but a **different parameter sequence**. (page 124). Of course, a class-extension can define a method that overloads a previously existing one.

⁷⁰The Java TechTip of October 10th of 2000 also considers shadowing effects.

⁷¹This statement can be proved to be true by changing the source-code above as follows: Each member of the classes “Original” and “Extension” is redefined by a leading modifier `static`. The line in which the object “e” is instantiated has to be omitted. The remaining symbols “e” are to be substituted with the reference “Extension”. And finally, possibly the identifier “Override-AndShadow” has to be adapted to a new file-name. A run of the program yields the same output that has been printed out above.

Java: Type-Casting III (Casting of Reference-Types)

Java: Type-Casting III (Casting of Reference-Types) — Type-casting already has been introduced with primitive-types (page 100). In principle, casting variables means to use the binary data and to reinterpret that binary data using a different kind of data-type; in practice, the process of reinterpretation has to be done carefully if sensible results are expected. Here, type-casting is presented also using reference-types:

Object to Primitive-Type and Reverse

Every primitive-type in Java has a predefined class-counterpart that is furnished by the Java language. This makes it possible to freely transfer a value between primitive-types and class-types in the following cases `byte-Byte`, `short-Short`, `int-Integer`, `long-Long`, `float-Float`, `double-Double`, `char-Character`, `boolean-Boolean`. (Remember, operations on variables of primitive-types are relatively fast, whereas variables of reference-type allow a unified and flexible approach to typing, more concrete: The associated class-types furnish a wider range of functionalities (conversion-methods) than the primitive-types⁷².)

To get a class-type, use the appropriate constructor:

```
double primitiveDouble = 1.23;
Double doubleObject = new Double( primitiveDouble );
```

Objects or constructors of objects, like the ones above, are also called **wrappers** when they leave the wrapper arguments of the constructor unchanged.

To get a primitive-type from a class-type, use the predefined instance-method `toClassType()`:

```
Integer integerObject = new Integer(123);
int primitiveInteger = integerObject.toInteger();
```

Where can be found further information about these wrapper-classes which are offered by the Java language for the primitive-types? See page 256 or consider the   Java Platform API Specification [JPAPIS] or the some book about the   Java Language Specification for the `java.lang`-package and the wrapper-class in question.

MiniFAQ⁷³: How to make an `int`-value appear in a `String`-object? Use the expression with an anonymous `Integer`-object:

```
new Integer( primitiveInteger ).toString()
// OR shortest within source-code:
"" + primitiveInteger
// Implicit casting triggered by the use of the string concatenation-operator.
```

Wrapper: Wrappers are constructors of objects or objects themselves for which the following two points are being stressed:

1. A wrapper-object references or encapsulates another object or variable and thus provides **new methods of accessing or working with** the object. Examples: `Integer` objects wrap integer-values, extending the range of possible operations. In the case of wrappers of variables of primitive-type, the original value of the wrapped variable cannot be changed. `java.security.SignedObject`- or `java.security.GuardedObject`-classes wrap an object and protect it against unwanted access, see the program on page 474.
2. Wrapper-objects **transform or filter the encapsulated data**, like `FilterStreams`. See page 292.

Type-Casting among Identifiers of Reference-Type: Widening and Narrowing

Remember that type-casting of primitive-types might change the value, for example casting a `float` or `double` into being an `int` (integer) means truncating the digits past the decimal-point (page 100). Contrary to that, **CASTING IDENTIFIERS OF VARIABLES OF REFERENCE-TYPE NEVER CHANGES THE OBJECT!** Remember also that every class, which is defined in Java without an explicit `extends`-clause, can be seen as having an invisible “`extends java.lang.Object`”-clause in its definition. Remember also that the super-class of all classes in Java is the `Object`-class. The `Object`-class has no super-class. All objects derived from predefined- or programmer-defined classes thus are sub-classes of the `Object`-class.

Consider a previously introduced, predefined class like the `String`-class:

⁷²And remember that, arguments of method-calls of primitive-type are transferred by-value, whereas class- or array-types within the parameter-list of a method-definition produce calls-by-reference (page 119).

⁷³FAQ=Frequently Asked Question(s)

```
String s = new String("A string of characters.");
```

A (String-)object also represents an instance of the super-class (Object). The actual object only carries the additional information (the string) that comes from instantiating the sub-class String⁷⁴. Thus, an object which has been instantiated using a definite class-type can be referenced by identifiers which may have the type of any super-class of the definite class-type. Short and imprecise: References to an object can be type-cast to any of its super-classes. Such a cast is called **widening of the identifier's type**. That widening can be done without an explicit cast-command:

```
Object o = s;
```

A reference only can be cast along its path in the class-hierarchy that leads to the ultimate super-class, here the class `java.lang.Object`. Short and imprecise: Casting references to an object outside its class-hierarchy is impossible⁷⁵.

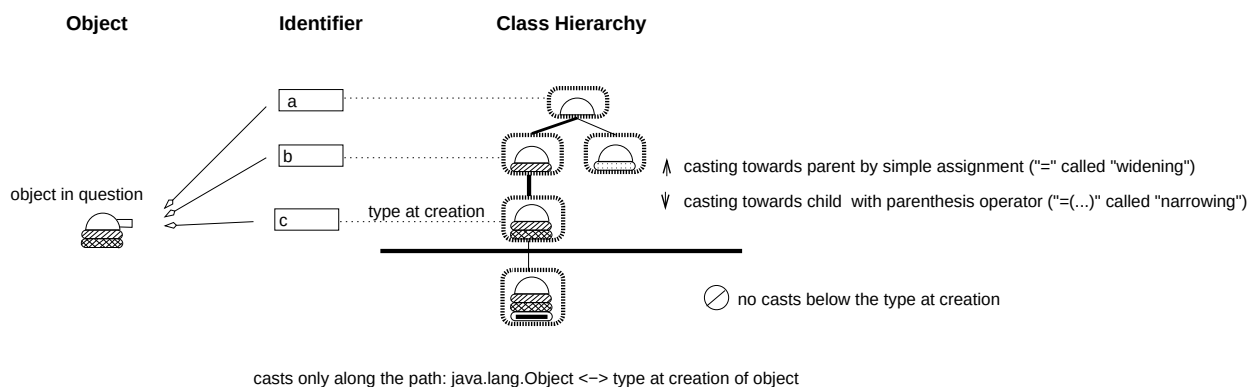
Casting the identifier of an object to a sub-class in its class-hierarchy becomes possible only if that object actually is an instance of that sub-class or if that object is an instance of a sub-class of that sub-class. Why this restriction? Because an object cannot be made automatically more refined than it was created:

```
String t; t = (String)o; // valid, in context of the two code-lines above
```

What has been indicated with the String-class is visualized below and exemplified with custom-written classes in the last lines of the program presented in the next section. Casting of class-variables thus always involves three entities: A given object, a class-hierarchy and various identifiers of that object.

Then, casting can be re-described from an object's point of view as follows: An object can have multiple references of various class-types; all these class-types have to be inside the path from `java.lang.Object` to the class-type according to which this object initially has been created. (This includes casts to interface-types (page 185) that are implemented by classes on that path!) In other words, the type of an identifier pointing to an object can be widened arbitrarily, but that type can only be narrowed along the line of class-extension leading to the class-type to which the object has been instantiated:

Casting of Class-Types: Object View

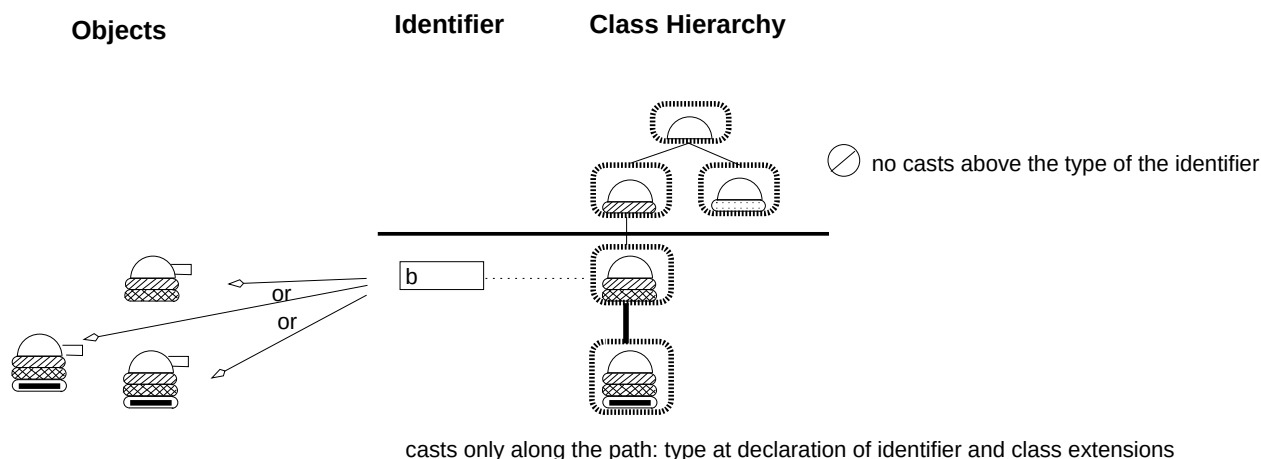


Casting also can be seen from a reference's point of view: A fixed single reference, an identifier, declared to be of some class-type can point to any object of an extended type. Such a reference never points to objects of any super-class. Example: A car with an automatic transmission remains to be a car, but the label "car" does not describe every four-wheeled object (bus, truck).

⁷⁴To speak in the language of the car-example: A car with an automatic gearbox remains a car, a car with a mechanical gearbox also remains a car.

⁷⁵A conventional Car-object shouldn't be sensibly be castable into being of FlyingObject-type; because, for example, steering and braking behaviors are expected to differ fundamentally.

Casting of Class-Types: Identifier View



The casting-information gathered so far, can be compiled into the following table:

to\from	Primitive-Type	Class-Type
Primitive-Type	As presented in the big table on type-casting in the section beginning on page 100.	the <i>toPrimitiveType()</i> -method in the case of wrapper-classes.
Class-Type	instantiating the class-counterpart (wrapper) of the primitive-type.	Same path of class-hierarchy required. The object should be of lower class-type than the two reference-types involved in the cast. Then widening or narrowing is possible.

Invalid casts throw a `ClassCastException`. For an extensive description of casting rules see books about the Java Language Specification, especially the Java Language Specification 2nd Edition Section 5.5 [JLS]⁷⁶.

For what tasks can be used the casting-concept in general, and the casting of types in particular? One aspect has been mentioned in the MiniFAQ above; there has been shown how to put an integer-value into a `String`-object, which proves useful for generating screen output. Another aspect of some importance may be the following: If some procedure implemented by a method only needs the object's data defined in some super-class, then it might be better for the method to take the object bearing the type of the super-class and to rely on auto-casting, than to require unnecessarily specific data in its parameter-list. One overloaded `print()`-methods on page 124 gives an elementary example of this strategy: The parameter of type `java.lang.Object` allows any identifier of reference-type to become an argument of a method-call.

Java: Validity of Identifiers: Scoping and Access-Control

Java: Validity of Identifiers: Scoping and Access-Control — Remember that the words “scope” and “scoping” describe the places in source-code where an unqualified identifier (variable, method, class) is valid (gray box on page 122). Using single identifiers outside their scope produces errors and exceptions of the kinds `cannot resolve symbol` (at compile-time) or `NoClassDefFoundError` (at run-time) errors. The **scope of an identifier** is the environment of its validity within the source-code or within the scope of an programming platform. Within the source-code the scope of an identifier denotes the part of the source-code in which the SINGLE, UNQUALIFIED identifier can be used to access its data, even if the part of source-code extends over packages, or files or a network in the case of class-names. (More on packages can be found later in this section.)

But next to the scope of an identifier, there exists a way to ACCESS DATA OR CODE FROM OUTSIDE THE SCOPE of its identifier(s): Remember the introduction and use of qualifiers for accessing class-members (page 154). ACCESSING DATA OR CODE FROM OUTSIDE THE SCOPE of its identifiers includes the un-shadowing of variables within objects/classes and methods, and also the un-overriding of methods, if

⁷⁶Another short introduction is given in the TechTip of January 10th 2000.

possible. This even includes the ability to control such an access with permissions and policy files as introduced in the chapter about security. All of these methodologies produce accessibility without changing the idea of scope:

Access-Area or area of accessibility of an identifier: The area of access includes those places from which the identifier can be referenced also by using additional means (qualification of identifiers, un-shadowing of variables, un-overriding of methods, having been granted the permission of access) to gain access to the data or code referenced by the identifier. Unlike scope, the area of access need not be fixed at compile-time and may even change during run-time.

Scope is a property of the identifier, whereas the area of access usually is given by and depends on the various environments, for example: If access has been forbidden, then the area of access may be *smaller* than the scope of the identifier!

Example with Hierarchical File System (HFS): The scope of a file-name is given by the folder in which the file exists. On an isolated computer-platform, the access-area for this file is made up by the entire operating-system, because the Hierarchical File System furnishes a system-wide path-name which accesses that file, for example `/usr/home/Desktop/fileName`. If the computer is connected to the Internet and the file has been made publicly available, then the extended file-name, prefixed by protocol- and domain-name, "`http://www.someName.org/usr/home/Desktop/fileName`" accesses the file, thus the access-area includes virtually all computers of the Internet; the scope remains the folder of the file!

As introduced on page 117, in the context of method-declaration, **modifiers** are used in declarations of classes, methods, variables and interfaces; there, modifiers may also determine scope, access-area and other qualities. The `Static`-keyword (page 147) which precedes class-members, or the `Void`-keyword which marks those methods that do not return values (page 117), are such modifiers. Modifiers which allow to control scope and access-area are given by the keywords `public`, `protected` and `private`; later these modifier are going to be considered more extensively.

Un-Shadowing and Un-Overriding

Un-Shadowing and Un-Overriding — Together with considering scopes of identifiers goes shadowing of variables; remember that variables declared in method-blocks are local variables. Those local identifiers shadowed non-local identifiers of the same name (page 123). As could be seen in a preceding section, shadowing effects can be observed even in extending classes. In extending classes the equivalent for "shadowing of methods" has been called "overriding". The following paragraphs show cases in which shadowed variables and overridden methods still can be accessed.

Un-Shadowing and Un-Overriding FROM INSIDE AN OBJECT

Instance-variables can be unshadowed by using the keyword `this`: One of the common cases for using the keyword `this` arises in method-declarations where variables of the parameter-list (as implicit local variables, page 121) have the same name as variables declared in the environment of the method-block:

METHOD block
Non-static method

```
int x = 3; ...

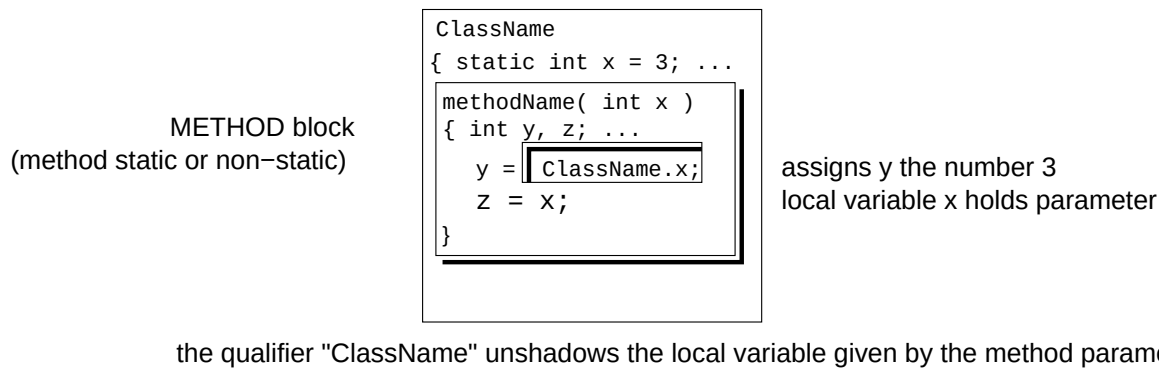
methodName( int x )
{ int y, z; ...
  y = this.x;
  z = x;
}
```

assigns y the number 3
local variable x holds parameter

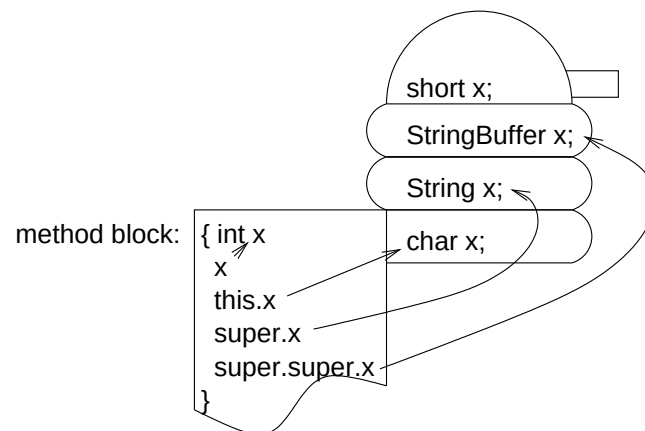
the qualifier "this" unshadows the local variable given by the method parameter

```
class Test
{ int x = 10; // member-variable
  void methodName( int x )
  { System.out.println( "local variable = " + x +
    " member-variable = " + this.x );
  } }
// a call
new Test().methodName(1); // yields: local variable = 1 member-variable = 10
```


This works also with static variables:



As indicated by the previous example of the case of the keyword `this`, the keywords `super` and `this` work only in constructors and non-static environments like instance-methods or instance-initializers. Applying the use of `this` and `super` explicitly to an instance of a class is demonstrated in the graphic below:

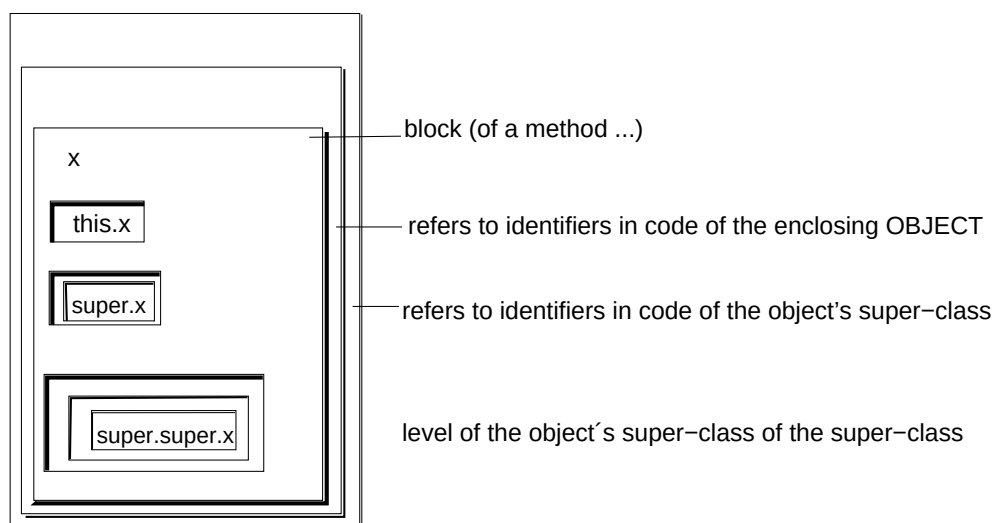


Again, `super`- or `this`-keywords can only occur in the body of an instance-method (non-static method) or a constructor or an initializer for an instance-variable.

```

this.variableName; // references the identifier in the enclosing object.
super.variableName; /* references an identifier of the same name
                     which has been declared in the super-class
                     of the enclosing object! */
    
```

From these code-blocks, the keyword `this` returns the identifier of the code-enclosing *object* (not that of the code-enclosing class!). THE KEYWORD **super** QUALIFIES INSTANCE-MEMBERS OF THE SUPER-CLASS.



Why is it useful to work with these keywords `this` and `super`, why not use the original identifiers? Generally it appears to be impossible to know the object's identity at the time of the class-definition! And,

identifiers of reference-type can be assigned easily into pointing to another object. Furthermore, changes of class- and object-names thus do not affect the code inside. (This represents an aspect of the concept of encapsulation.) The keywords `this` and `super` are often found in constructors: Overloaded constructors can communicate with each other by using the `this`-keyword, instead of using an identifier referencing the enclosing object (see the constructor-chain on page 167). The constructor of a super-class is invoked by using the `super()`-call in Java. In pieces of source-code, declared without the `static`-modifier or constructors, the `super`-keyword can be used to call overridden methods:

```
super.methodName()
/* References the (non-static) method in
   the enclosing object which has been declared in the super-class. */
```

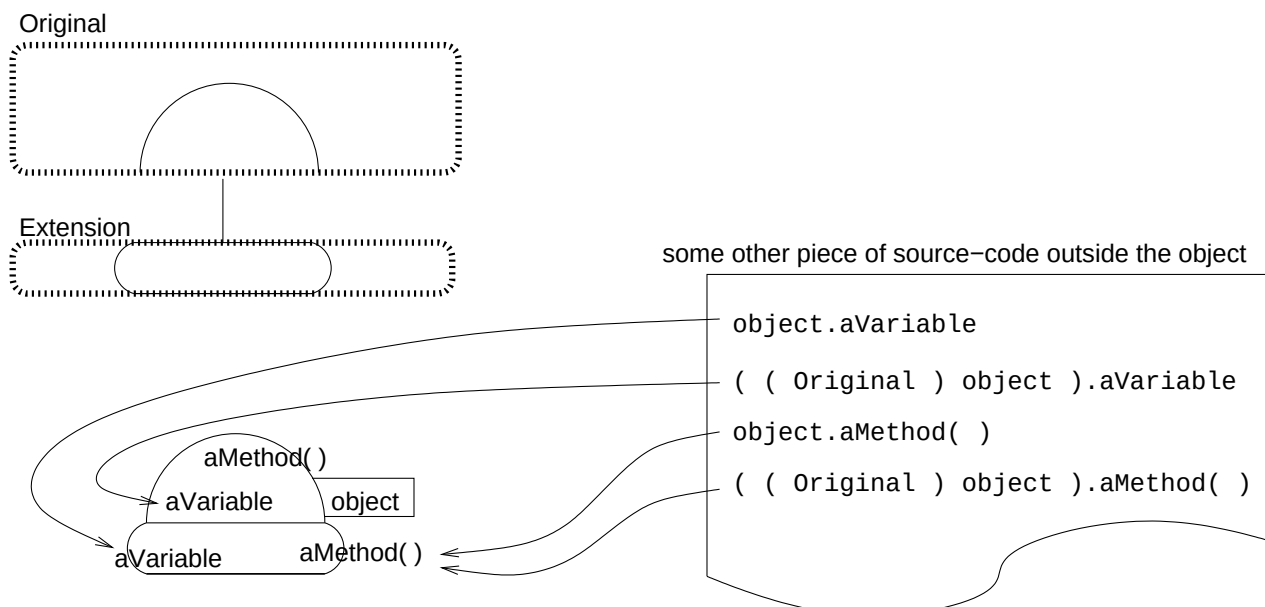
The access strategies presented so far are demonstrated in the following program. To access static variables and static methods use the class-name as qualifier of the static member-name.

```
class Original
{
    int aVariable;                // instance-variable
    int aMethod(){ return aVariable; } // instance-method
    void out( String s ){ System.out.print(s); }
}
class Extension extends Original
{
    int aVariable=10;              // overshadows the Original variable
    int aMethod(){ return 3*aVariable; } // overrides the Original aMethod()

    void accessFromInside(int aVariable) // argument-name overshadows the instance-variable
    { out
      (
        "ACCESS FROM INSIDE: Parameter of method          aVariable = "+aVariable
        +"\nVariable of enclosing class                  this.aVariable = "+this.aVariable
        +"\nVariable of super-class of the enclosing class super.aVariable = "+super.aVariable
        +"\n((Original)this).aVariable = " + ((Original)this).aVariable + " = super.aVariable"
        // the method:
        +"\nMethod of enclosing class                    aMethod() = " + aMethod()
        +"\nOverridden aMethod unshadowed    super.aMethod() = " + super.aMethod()
        +"\n"
      );
    }
}
// The code above is part of the following class "Unshadow",
// therefore these classes should be found by the Java compiler in one text-file.
```

Un-Shadowing and Un-Overriding FROM OUTSIDE AN OBJECT

The first thing to note may be that **OVERRIDDEN METHODS CANNOT BE ACCESSED FROM OUTSIDE THE CLASS OR INSTANCE.**



When accessing instance-members from outside the instance, only variables can be unshadowed by type-casting as presented in the program below:

```
// this class uses the preceding classes Original and Extension
class Unshadow
{ public static void main( String unused[] )
{
    Original o, oo;
    Extension e, ee;
    o = new Original();
    e = new Extension();
    // ----- NOW ACCESS FROM OUTSIDE
    o.out( "\nACCESS FROM OUTSIDE\n" );
    // same results with e.out( "\nACCESS FROM OUTSIDE\n" );
    ((Original)e).aVariable = 1;
    /* Access to members INside the class with 'super' and 'this',
       from OUTside the class with casting! */
    e.aVariable = 10;
    o.out
    ( "e.aVariable = " + e.aVariable + " is instance member"
      + "\n((Original)e).aVariable = " + ((Original)e).aVariable
      + " is instance member of parent, unshadowed"
      + "\ne.aMethod() = " + e.aMethod()
      + "\nbut CASTING a reference gives no access to OVERRIDDEN METHODS:"
      + "\n((Original)e).aMethod() = " + ((Original)e).aMethod()
      + "\n"
    );
    e.accessFromInside(100); // ----- NOW ACCESS FROM INSIDE

    // ----- finally some possibilities for casting
    oo = (Original)e;
    ee = (Extension)oo;
    /* ee = (Extension)o;
    // throws a Runtime exception:
    // Exception in thread "main" java.lang.ClassCastException: Original at...
    */
} }
/*
marycosway@linux:~/JavaCodeFiles/Basic1/SCObjectOrientation2 > javac Unshadow.java
marycosway@linux:~/JavaCodeFiles/Basic1/SCObjectOrientation2 > java Unshadow

ACCESS FROM OUTSIDE
e.aVariable = 10 is instance member
((Original)e).aVariable = 1 is instance-member of parent, unshadowed
e.aMethod() = 30
but CASTING a reference gives no access to OVERRIDDEN METHODS:
((Original)e).aMethod() = 30
ACCESS FROM INSIDE: Parameter of method          aVariable = 100
Variable of enclosing class                      this.aVariable = 10
Variable of super-class of the enclosing class   super.aVariable = 1
((Original)this).aVariable = 1 = super.aVariable
Method of enclosing class                       aMethod() = 30
Overridden aMethod unshadowed                   super.aMethod() = 1
marycosway@linux:~/JavaCodeFiles/Basic1/SCObjectOrientation2 >
*/
```

Again observe that casted references cannot access overridden methods from outside; again overridden methods cannot be referenced from outside!

Thus overridden accessor-methods like `getVariable()` and `setVariable()` of a variable declared `private` (see later) cannot be reactivated from outside the object.

Overriding combined with scoping as presented in the next section produces new possibilities such as the following: Scope-restricted methods can be made accessible by overriding them properly and calling them from the overridden methods, see the pieces of source-code on page 192 and on page 285.

Access-Control

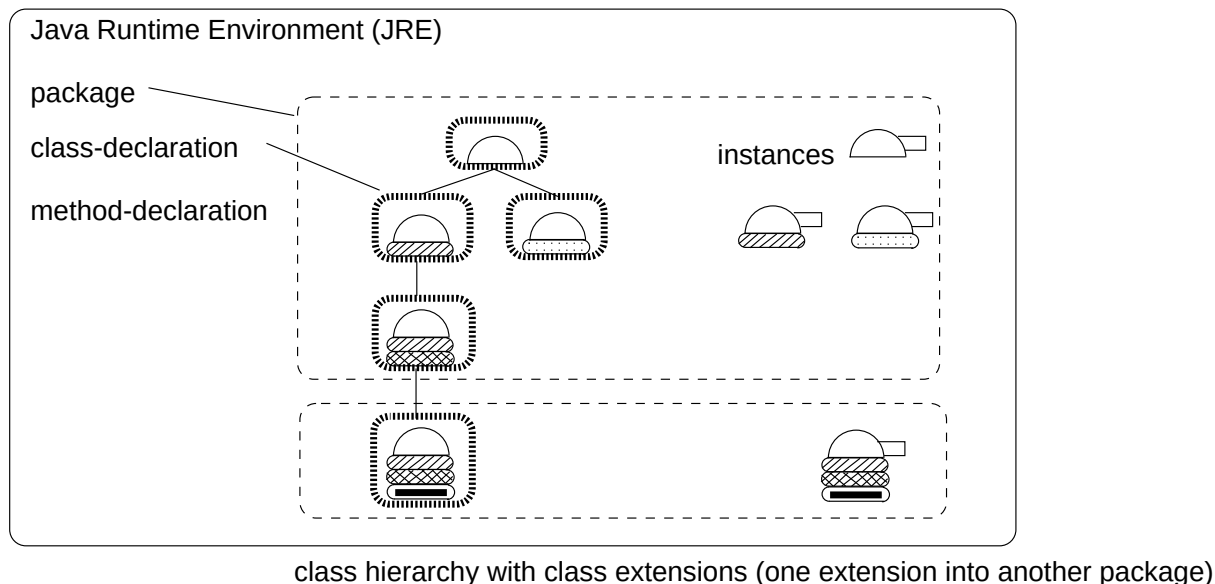


Access-Control — What kind of environments are meant when considering scoping and access-control of Java identifiers? There may be method-blocks possibly related by method-call chains (like in recursion) containing local variables — the class-definition containing static members — the instantiated object containing instance-members — the package⁷⁷-environment which acts like a surname for a given

⁷⁷What is a package? Classes — rather top-level-classes — can be aggregated to so-called packages. A package usually is given by a file that contains such an aggregation of top-level-classes. As has been said, an application-programmer can see package-names like surnames for classes, for example `java.lang.Object` instead of simply `Object`, or an identifier `java.io.InputStream` instead of a simple name `InputStream`. Java packages used as a means for structuring code are described later.

set of classes, but there exists also the entire Java Runtime Environment (JRE). If some source-code has not been declared as being part of a package, then that source-code is part of the so-called default-package in the Java Runtime Environment.

Some Java Access and Scoping Environments



Classes (page 146), members of classes (page 154), constructors (page 150), variables (page 147) and methods (page 117), all have a scope and access-areas. During their definition, these programming constructs and their respective scopes and access-areas, can be partly influenced by using modifiers: **private**, **default** (=no modifier), **protected** and **public**. Consider a simple example of using the scoping modifiers with instance-members of a class:

```
class AccessModifiersOnInstanceMembers
{
    private byte privat = 4;
        // Scope: The object itself.
        // Access-area: Those objects instantiated from this class.
        // NO access from objects instantiated from extended classes.
        //
    int default = 4;
        // Scope: Objects instantiated from this class AND
        // objects of extended classes in the SAME package.
        // Access-area: The package of its definition.
        //
    protected short getValue(){ return 4; }
        // Scope: Objects instantiated from this class AND
        // instances of extended classes even in some OTHER package!
        // Access-area: The package of its definition and its scope.
        //
    public String note = new String( "four alternatives" );
        // Scope: Objects instantiated from this class or any sub-class.
        // Access-area: Objects instantiated from any loaded class within the
        // Java Runtime Environment, if qualified by some reference of the object.
}
```

A private access of class-members makes the associated identifier **ONLY VALID WITHIN THE CLASS BUT NONE OF ITS EXTENSIONS**. If the class or one of its extensions is instantiated, then a non-static identifier is valid only inside source-code of the same class-type⁷⁸ (not within code of the extensions

⁷⁸That means also: Different instances of exactly the same class-type can access each other's private members! The following piece of code is supposed to prove that assertion:

```
class PrivateAccess
{
    public static void main( String [] unused )
    {
```

or any outside code). Access from outside the same class-type is considered NOT possible, either by qualifying the identifier with a reference to the instance or by any other means within the Java source-code.

The default case of class-members makes the identifier valid and known only within the sub-classes INSIDE the class where the identifier has been defined, so much for the scope. The access-area is given by the entire package, access from other classes of that package is possible by qualifying the identifier with a reference to the instance.

A protected access of class-members makes the identifier valid and accessible inside the package AND extends its scope to all the sub-classes (regardless in which package they exist). The scope thus is larger than in the default case. Access from the originating package is possible by simply qualifying the identifier with a reference to the instance. Access from everywhere else is NOT possible or NOT EASILY possible (page 285).

A public access of class-members makes the identifier valid and accessible in all loaded packages, in the entire Java Runtime Environment. Thus the access-area is given by the entire Java Runtime Environment. The scope (identifier suffices to reference the member) remains the object itself in the case of instance-members, the scope remains the class itself in the case of class-members. But the access-area (access becomes possible by qualifying the identifier with a reference to the instance) is given by the entire Java Runtime Environment.

The following fully-functional piece of source-code shows differing aspects of the access-modifiers in a working environment. The package declaration and installation has been anticipated here. Classes of installed packages (page 242) become accessible within a given Java Runtime Environment; a package, which is not properly installed, produces compile-time-errors like `Class c className not found in import`. The first piece of source-code defines a class “Original0” in a package with the name “Experimental”.

```
/* Put the associated class-files ("Original0.class" and "Extension1.class")
 * into folder with name "Experimental";
 * and put that folder ("Experimental") into a folder inside the current-directory
 * (or inside a folder of the CLASSPATH).
 * (for more see the chapter about packages)
 */
package Experimental;
public class Original0
{ private    int privat    = 1;
      int default = 2;
      protected int protecte = 3;
      public    int publi    = 4;
      public Original0()
      { System.out.println( "\nfrom Original0 instance: privat = " + privat );
        // All other members are fully accessible from here.
      } }
}
```

The class “Original0” declares four instance-members, each a variable of primitive-type with a different type of access-modifier. The private member is accessible only from within (here the block of the con-

```
Privat p1 = new Privat(1);
Privat p2 = new Privat(2);
p2.set( p1, 0 );
// System.out.println( "p1 = " + p1.x );
// the above produces a compile-time error: x has private access in Privat
System.out.println( "p1 = " + p1.accessPrivate() );
System.out.println( "p2 = " + p2.accessPrivate() );
}
}
class Privat
{ Privat( int in ){ x = in; }
  private int x;
  public void set( Privat p, int s ){ p.x = s; } // <---- the case of mutual modification
  public int accessPrivate(){ return x; }
}
/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > javac PrivateAccess.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > java PrivateAccess
p1 = 0
p2 = 2
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 >
*/
```

structor) an instance of class-type “Original0”; all other attempts of access, like qualification or casting should fail.

The second piece of source-code adds a class “Extension1” to the “Experimental” package.

```
/* To compile,
 * put the associated files of source-code ("Original0.java" and "Extension1.java")
 * into a folder of name "Experimental".
 */
package Experimental;
public class Extension1 extends Original0
{ public Extension1()
  { System.out.println
    ( //"\nfrom Extension1 instance: privat = " + privat +
    // the above produces a compile-time error: not accessible
    "\nfrom Extension1 instance: default = " + default +
    "\nfrom Extension1 instance: protecte = " + protecte +
    "\nfrom Extension1 instance: publi = " + publi
    );
  }
}
/*
[localhost:~/JavaCodeFiles/Basic1/SCObjectOrientation2/Experimental] andreadipietro% javac Extension1.java
(PROBLEM) Extension1.java:10: cannot resolve symbol
(PROBLEM) symbol : class Original0
(PROBLEM) ...
--- possible problems in compilation:
    cannot resolve symbol: class Original0 OR
    Superclass Experimental.Original of class Experimental.Extension1 not found
--- remedy: If not already done,
    put source-code with a package declaration into a folder of the same name
[localhost:~/JavaCodeFiles/Basic1/SCObjectOrientation2/Experimental] andreadipietro% javac Extension1.java
(PROBLEM) Extension1.java:10: cannot resolve symbol
(PROBLEM) symbol : class Original0
(PROBLEM) ...
--- AND compile it from parent folder as shown below:
[localhost:~/JavaCodeFiles/Basic1/SCObjectOrientation2] andreadipietro% javac Experimental/Extension1.java
[localhost:~/JavaCodeFiles/Basic1/SCObjectOrientation2] andreadipietro%

(ALTERNATIVE) Compiling the code that uses the package,
                AUTOMATICALLY compiles the uncompiled code of the package!
*/
```

The private member of the “Original0”-instance is unaccessible from an instance of any extension. The extension within the same package is part of the scope of all the rest of the member-variables; since as can be seen, they remain accessible by using solely their identifiers.

The third and final piece of source-code does sub-class the existent hierarchy outside its “Experimental”-package. This makes it possible to demonstrate the different functionalities of the protected and public access-modifiers as applied in the “Original0”-class:

```
/* Any piece of source-code without a package-declaration, causes
 * the compiled byte-code to be run by the Java Runtime Environment as a
 * part of the (so-called) default-package!
 * This is true especially for the classes below: "Extension2" and "ImportingExtension":
 */

import Experimental.Original0; // needs class Original0 to be declared public
import Experimental.Extension1; // needs class Extension1 to be declared public

class Extension2 extends Extension1
{ Extension2()
  { System.out.println
    (
    //"\nfrom OUTSIDE PACKAGE but in sub-class of Extension1 instance:default = "+this.default+
    /* ^ the above produces a compile-time error:
    * default is not public in Experimental.Original0; cannot be accessed from outside package
    */
    //"\nfrom OUTSIDE PACKAGE but in sub-class of Extension1 instance: protecte = "+this.protecte
    );
  }
}
/*
 * The class below finally contains the main()-method:
 */
class ImportingExtension
{ public static void main( String unused[] )
  { Original0 original = new Original0();
    // this needs constructor Original0() to be declared public
    Extension1 extension1 = new Extension1();
    // this needs constructor Extension1() to be declared public
    System.out.println
    (
    //"\nfrom OUTSIDE package Extension1 instance: default = " + extension1.default +
    /* ^ the above produces a compile-time error:
    * default is not public in Experimental.Original0; cannot be accessed from outside package
    */
  )
}
```

```
// "\nfrom OUTSIDE package Extension1 instance: protecte = " + extension1.protecte +
/* ^ the above produces a compile-time error:
 * protecte has protected access in Experimental.Original0
 */
"\nfrom OUTSIDE PACKAGE package Extension1 instance: publi = " + extension1.publi
);
//////////////////////////////////// first class OUTSIDE the package
Extension2 ext2 = new Extension2();
System.out.println
(
// "\nfrom instance of sub-class outside package: default = " + ext2.default +
/* ^ the above produces a compile-time error:
 * default is not public in Experimental.Original0;
 * cannot be accessed from outside package
 */
// "\nfrom instance of sub-class outside package: protecte = " + ext2.protecte +
/* ^ the above produces a compile-time error:
 * protecte has protected access in Experimental.Original0
 */
"\nfrom instance of sub-class outside package: publi = " + ext2.publi
);
} }
```

(above is part of below) First, as can be seen (“Extension2”-constructor), the protected member remains accessible from a class extended outside the package. The classes of the package “Experimental” have to be declared public to be known outside the package, and to be instantiable in a piece of source-code outside the package their constructors have to be declared public. Additionally only the public members of the “Original0”-instance remain accessible outside their package. (below is part of above)

```
/* PREREQUISITE: package "Experimental" is loadable, that means for example:
PREREQUISITE: JavaCodeFiles/Basics1/SCObjectOrientation2 contains a folder
named "Experimental",
which in turn contains the files of
source-code "Original0.java" and "Extension1.java"
(then these files are found and compiled automatically
by the single compilation command below!)
(As usual, be aware of typing errors!)
PREREQUISITE: OR the package has been placed into the ext-folder
as a compiled and jarred file.
*/
/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > javac ImportingExtension.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > java ImportingExtension

from Original0 instance: privat = 1

from Original0 instance: privat = 1

from Extension1 instance: default = 2
from Extension1 instance: protecte = 3
from Extension1 instance: publi = 4

from OUTSIDE PACKAGE package Extension1 instance: publi = 4

from Original0 instance: privat = 1

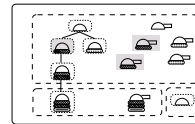
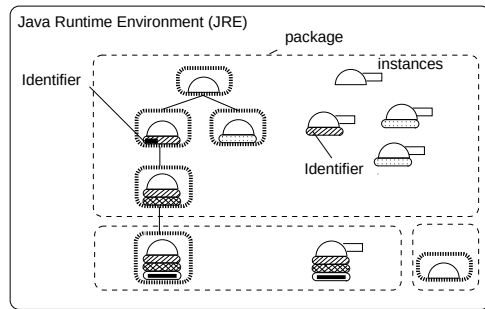
from Extension1 instance: default = 2
from Extension1 instance: protecte = 3
from Extension1 instance: publi = 4

from OUTSIDE PACKAGE but in sub-class of Extension1 instance: protecte = 3

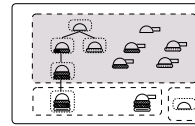
from instance of sub-class outside package: publi = 4
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 >
*/
```

The output above is given to trace back individual instantiations and print-commands coming from the different pieces of the three chunks of source-code presented earlier. Either by using the `println()`-commands or by following the course of the source-code, the output-listing can be associated with the locations within the source-code. If possible, do freely experiment with the pieces of code. A coarse visualization of the access-areas of members declared with the private, (default), protected and public access-modifiers is given below:

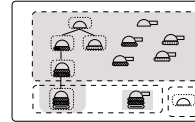
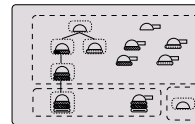
class hierarchy with class extensions (one extension into another package)



private identifier valid and recognized within object alone



non-modified identifier's area of access is its own package

protected identifiers additionally are accessible in ALL sub-classes
(also sub-classes declared within other packages)

public identifiers are accessible from all loaded packages

There are means for circumventing the effects of access-modifiers in Java, but these means have a rather indirect character and work strictly within the rules of the language-specification of Java⁷⁹. Such an example of Java is given by the Observable/Observer source-code on page 285, there is shown how to unprotect a method by overriding it.

Modifiers and Accessibility

Modifiers and Accessibility — Below are listed the effects of the access-modifiers `public`, `protected` or `private` on different programming entities, like class, method, constructor and the like. In the case marked as “default” no access-modifier prefixes the declaration. The notion of an access-area below is always meant as excluding the cases of shadowing variables or overriding methods. (Be reminded of the definition of the notion “access-area” on page 172)

public (means accessible from everywhere within the current Java Runtime Environment, by qualification or casting)

field/variable Accessible from any loaded class or from any instantiated object. If necessary, instance-variables have to be qualified by an identifier referencing the object, whereas static variables (class-variables) have to be qualified by the class-name or even by the entire package-name.

method is accessible from any loaded class or instantiated object. Instance-methods have to be qualified by an identifier referencing the object, static methods (class-methods) have to be qualified by the class-name or even the package-name. When declaring interface-types (page 185), each method-declaration becomes implicitly public and abstract.

constructor Public constructors are accessible from any loaded class or instantiated object, if necessary, qualified by the package-name. A public CONSTRUCTOR MAKES MEMBERS AND INNER INTERFACES IMPLICITLY public!

class Used only for top-level-classes or member-classes which become accessible from all loaded classes or instantiated objects. Outside the default-package also public classes have to be qualified by the package-name. A file of source-code or a so-called compilation unit(=usually the same as a file of source-code, more around page 246) can contain only one public class.

interface (page 185) qualified by the package-name, is accessible from any loaded class or interface. This accessibility includes the interface's field-variables (quasi-constants) and the interface's methods, INDEPENDENTLY OF THEIR MODIFIER. Non-public interfaces give their field-variables and methods default accessibility, which means access from the same package.

protected (means accessible from any sub-class, even if that sub-class has been declared within another package, AND, with proper qualification, accessible within the package of declaration)

field/variable A protected field-variable is accessible from any class in the package of declaration and any sub-class. If necessary, the variable-name has to be qualified with the class-name (static variable) or the name of a reference to the object (instance-variable).

⁷⁹Unlike the tricks in the language C++, which there, at worst, can make the entire concept of access-modifiers immaterial.

method is accessible from any class in the package of its declaration and any sub-class. If necessary, the method-name has to be qualified with the class-name (static method) or the name of a reference to the object (instance-method).

constructor is accessible from any class in the package of its declaration and any sub-class.

class A top-level-class produces compile-time error.

inner class accessible from any class in the same package and any sub-class.

default (means access-area is same package, access from outside the package is denied, even with a qualified identifier)

field/variable Default accessible by classes of the same package. The unqualified name of the variable is recognized within the class or object. Other classes or objects within the same package get access by properly qualifying the variable-name.

method Default accessible by classes of the same package. Same as with variables.

constructor Default accessible by classes of the same package. A class, with all constructors declared without a modifier, can be instantiated only by code belonging to the package in which the class has been declared.

inner class Default accessible only over classes of the same package that have access to the enclosing class.

class/interface known to all classes or interfaces within the same package. Access from outside the package is denied, even if the identifier is qualified.

private (means accessible only from within the top-level-class in which declared OR objects of exactly the same class-type. Any private members of a class are NOT accessible from any sub-class.)

field/variable A private variable is only accessible in the class in which it is declared, especially there is no access from any of the sub-classes. Thus private members are not inherited, a private variable cannot be shadowed. Any private instance-variable is only accessible from all objects of exactly the same class-type, as shown in the footnote on page 176.

method A private method is only accessible in the class in which that method has been declared, especially there is no access from any of the sub-classes. A method cannot be declared to be private and abstract together, because abstract methods must be overridden, but private methods are not inherited by a sub-class. So, private members are not inherited, a private method cannot be overridden.

constructor only accessible within the top-level-class in which it is declared, especially there is no access from any of the sub-classes, thus if only private constructors exist — without being called by static methods of the same class — the class cannot be instantiated. This becomes useful in cases where the class should not be instantiable; for example when the class provides only a collection of static methods, like the `java.lang.Math`-class.

class A top-level-class produces compile-time error.

Overriding methods or shadowing variables with more restrictive access-modifiers causes a compile-time error.

Java: Other Modifiers

abstract (means to be implemented within some class-extension) For more see the next section.

methods are declared without a block that implements the method, they are just specified by the return-type of the method, the name and the parameter-list. Abstract methods have to be overridden in a sub-class by an implemented method. Each abstract method can only be declared in an abstract class, which may also contain non-abstract methods. Any abstract method cannot be declared private or final, because no method of that kind can be overridden.

class/inner class Each abstract class cannot be instantiated; but members, variables, inner classes and methods may be implemented and may be ready for inheritance. For example, the class `java.lang.Number` has been declared to be an abstract super-class of `Byte`-, `Short`-, `Integer`-, `Long`-, `Float`- and `Double`-classes. Each instance of class-type `Number` should be one of the mentioned sub-classes, so to enforce this restriction, the `Number`-class has been defined

abstract. Methods of the `Number`-class include the self-explaining conversion methods `byteValue()`, `...`, `doubleValue()`, which all are implemented suitably for each sub-class. Another kind of uninstantiable classes are those given with `private` constructors, as has been mentioned a few paragraphs before.

static (part of the class, rather than part of an individual object of that class-type)

field/variable also called class-variable, because a variable declared with the modifier `static`, is allocated and instantiated solely within the class which has been loaded. Such a variable is found within none of the derived instances/objects. If a class-variable is changed, then the change is seen by all objects that are instances of that class or of one of its sub-classes. As an example see the program on page 155, there a class-variable is used to count how many objects of that class have been instantiated in the current Java Runtime Environment.

Non-static variables are the default case of defining variables in Java, these variables are also called instance-variables, because non-static variables are allocated and instantiated as part of a specific object to be derived from that class. Each object of a class has its own set of instance-variables associated with it.

method also called class-method, because it is associated with the class and not with an instance of the class. Example: `java.lang.Math.tan(x)`. So, no instance is necessary to invoke the static method of a loaded class. If a method carries a kind of state, then this state is associated with the class. Each static method, accessing fields of an instance of that class, has to do this by explicitly referencing the individual object. In static methods the keywords `super` and `this` are undefined.

Methods declared without a `static` modifier, are called instance-methods or non-static methods. These methods are associated with an instance of a class. If an instance-method carries a kind of state, then it may have a different state for each object derived from the class. Instance-methods may use unqualified references to instance-members of their class. Instance-methods may also use references with keywords like `this` and `super` (see page 172 and page 166).

class/inner class A static top-level-class produces a compile-time error. Member-classes may be declared `static`, then they become part of the enclosing class, not the instance. A static inner class cannot access non-static variables and methods of its enclosing class.

Non-static inner classes can access all variables (`public`, `protected`, `default` and `private`) and methods of the enclosing class. Those kind of classes are often used as programmer-implemented forms of local types.

Static data or methods can be used in constructors of the class, instance-data or methods cannot be used that way.

Static class-members can be accessed directly by qualifying their name with the class-name and the qualification-operator (`.`).

final (makes unoverridable methods, quasi-constant fields, non-extensible classes)

method A final method cannot be overridden, that means there cannot be a method in any sub-class having the same name and the same parameter-list (signature). A method with the same signature in a sub-class causes an error to be thrown. A `final` method still can be overloaded, that means, methods with the same name but a different parameter-list still can be declared in a sub-class! Declaring methods to be `final` makes their invocation faster as has been mentioned on page 168.

field/variable A final variable is a constant (`final float NEAR_PI=3.1415;`) if instantiated on its declaration, otherwise (`final float BLANK;`) the variable is called a **blank-final**. Blank-finals CAN BE INITIALIZED EXACTLY ONCE AFTER THEIR DECLARATION. Static blank-finals can be assigned a value in a static-initializer ONCE, POSSIBLY OUT OF A CHOICE. Non-static blank-finals can assigned a value in an instance-initializer or a constructor ONCE POSSIBLY OUT OF A CHOICE. Therefore, the initializer does not have to be a constant expression, for example:

```
final int DAYS_IN_YEAR;
if(isLeapYear(new Date())) DAYS_IN_YEAR=366; else DAYS_IN_YEAR=365;
```

Contrary to `final` methods, where no method of the same signature can exist in any sub-class, there may exist a variable in a sub-class and that variable may have the same name as the final variable in the super-class! Thus CONSTANTS MAY BE SHADOWED.

class/inner class A final class cannot be sub-classed, thus a final class cannot be extended. Thus the identifier of a final class cannot appear after the extends-keyword in the declaration of another class (The, by default, late-binding Java compiler often can optimize operations for final classes.). All methods in such a class are implicitly final. For example, the String-class and other core Java-classes have been declared final to assure their standardized functioning, no programmer can extend these classes.

All final inner classes or final local classes act the same as final classes.

If **ALL FIELDS** of a final class have been declared final, then the class may be called “constant class” and the objects instantiated from that class may be called “constant objects”. Thus final fields produce (quasi-)constants, final classes do not, unless all their fields are also declared final!

transient field-variables are not saved and restored by the standard object-serialization mechanism.

That mechanism allows to transform serializable objects into sequences of bytes without carrying the class-information, so serialized objects can be even stored on a disk and thus made persistent. That’s useful when creating a database. For a practical example see the program on page 309.

volatile field Such variables are not guaranteed to give a globally consistent value when modified by different threads, because they are not cached internally to assure their up-to-date status. Seldom used.

native method The declaration of a native method ends with a semicolon rather than with a block.

The keyword native means that such a method is not implemented in Java. A native method has to be implemented in some other programming-language, often platform-specific C++ or assembler. Native code may become useful for managing a time-critical process, but there are considerable disadvantages like a specific installation-process and usually lost portability of the remaining Java code. This text does not discuss native methods.

synchronized method A method declared synchronized makes the computer’s processor exclusively work on that method’s code without interruption by any other Java thread. Therefore the keyword synchronized may be better illustrated by the word machine-synchronized. For more see the chapter on multithreading.

strictfp class or method makes operations on floating-point variables (of type float and double) less affected to over- and underflow errors.

Java: Class-Extension with Abstract Classes and Interfaces

Java: Class-Extension with Abstract Classes and Interfaces — Abstract classes and interfaces can be used in several ways when structuring source-code in an object-oriented way. Those types of class constructs can be used as templates, by letting methods unimplemented for later consideration. Both types of class-constructs can thus be used to formulate the structure of a programming task, in the case of interfaces also realizing a form of multiple inheritance. Interfaces also can be used to give otherwise unassociated classes a common base for communication, therefore the name. Furthermore interfaces can be used for tagging, that means marking objects as being copyable, or cloneable, even for marking them as being serializable, which allows the Java compiler to transform them into an exportable sequence of bytes.

When introducing them, these types of class-constructs seem to convey very little content, but as indicated above, they can be found in a surprisingly broad range of applicational uses. A conceptual introduction using the “Car”-class has been already given on pages 133 to 136. A first example in Java has been given by the abstract class `java.lang.Number` when considering the modifier `abstract` on page 181.

Java: Abstract Classes and Abstract Methods

Java: Abstract Classes and Abstract Methods — Remembering the class/object-concept, with its “Car”-class, one can suspect that there remains the Java implementation of abstract classes, abstract methods and probably some more unpredicted effects. Java allows abstract classes to be defined as follows:

```
abstract class aName{ declarationsOfTheAbstractClass }
// an abstract class

// to become instantiable, an abstract class has to be extended:
class name extends aName{ additionalDeclarationsOfTheExtension }
new name();
```

In Java, abstract classes have to be extended, they have to be super-classes to make an instantiation or object-creation possible. ABSTRACT CLASSES ARE NEVER INSTANTIATED, there exist no objects of the type of an abstract class⁸⁰.

(methods) Abstract classes may contain fully-defined methods as well as they may contain methods declared abstract. Those methods which are fully-defined methods may also contain an empty-block, to be overridden by the methods of sub-classes.

(variables) Because ABSTRACT CLASSES are never instantiated, they CAN ONLY ASSIGN VALUES TO STATIC VARIABLES!

The associated syntax-diagram is that of a ClassDeclaration with an abstract modifier which can be found on page 146.

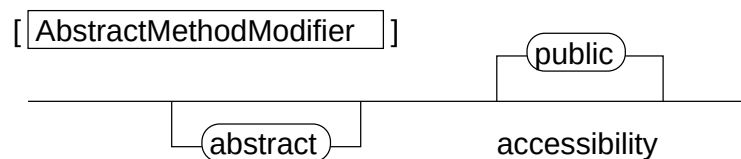
Abstract methods are Java methods which have been defined with an abstract modifier. Those definitions merely introduce the return-type of the method and its signature (name and parameter-list).

```
abstract void print( String s );
```

So, the abstract modifier introduces unimplemented methods. Where could that be useful? This can be useful to clarify a structure without knowing the implementation. Remember the conceptional examples beginning on page 132, there the “brake()”-method had been declared abstract. And consult the programming example on page 186.

A class which declares one abstract method has to be declared automatically abstract, otherwise the Java compiler sends a message like “class *className* must be declared abstract. It does not define *methodName* from class *className*”.

AbstractMethodDeclaration := { *AbstractMethodModifier* } *ResultType* *MethodDeclarator* [*Throws*] ; — (See MethodDeclaration on page 117, the choice of modifiers is drastically restricted and there exists no MethodBody)



AbstractModifier := public | abstract

Again: DECLARING A CLASS ABSTRACT MAKES IT UNINSTANTIABLE. A class with no abstract methods still may be declared abstract, if the application-programmer wants to enforce a class-derivation process. But usually classes are declared abstract because some of their methods cannot be specified at the time the class is written.

Abstract Method (non-implemented method) A method declared without a method block and thus left unimplemented. When the enclosing class is extended to a non-abstract class, an abstract method is *required* to be defined. (Defining a method with an empty block in the first place, would not guarantee that method to be overridden and redefined in a process of class-extension!) Any abstract method defined in a class automatically makes the enclosing class abstract.

Abstract Class 1. (non-instantiable class) A non-instantiable class, either by having been explicitly declared abstract or by containing an abstract method. There will be found NO objects of that class-type in any Java Runtime Environment.

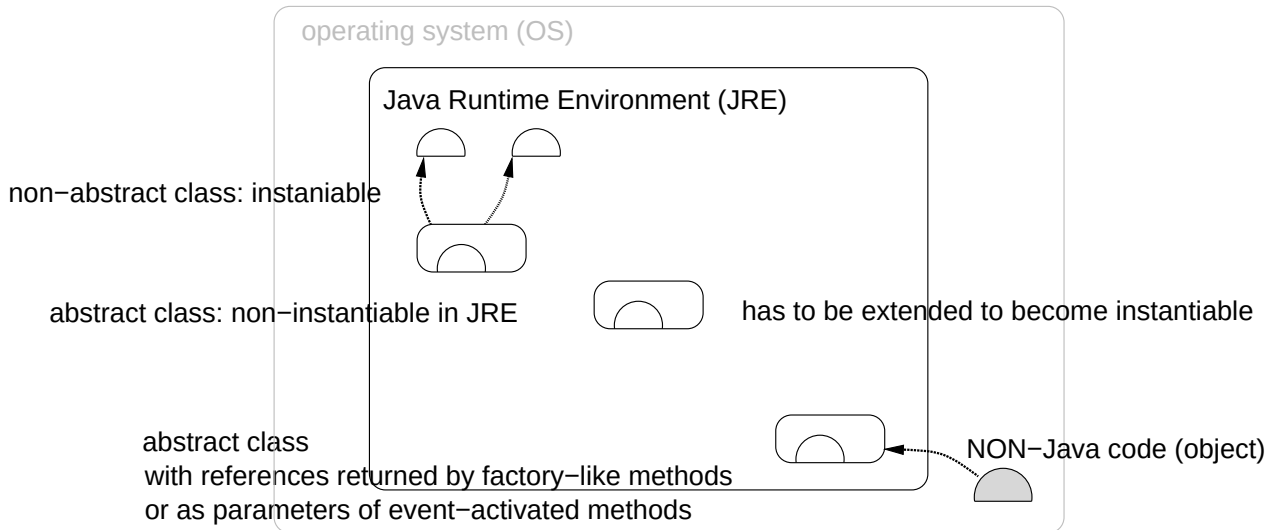
2. (valid class-type) An abstract class still can be used as class-type for declaring identifiers and in casting identifiers.

3. (class-type of references pointing outside the Java Runtime Environment, JRE) Some methods return references of that type for reusing native code. (See below:)

CAUTION: Instead of introducing a special type of external or native identifiers, in some cases, Java pre-written code (given as class-files bundled to packages) reuses the abstract-modifier to describe references to native code: One of those very special cases in which an identifier can reference an entity of an abstract class-type is given by the Image-class of the java.awt-package: The Image-class (page 527) is declared abstract; nevertheless there are factory-like-methods which return references of class-type Image. How can this be? Those references point to image-information available as native data (=that means NOT a Java object), this saves data-transformation-time, but unfortunately also reduces the possibilities of manipulation by Java source-code. There are other (page 551) classes of

⁸⁰Nevertheless, identifiers of objects can be cast to or declared to be of an abstract class-type.

that kind, for example the `java.awt.Toolkit`-class, referencing non-Java code outside the Java Runtime Environment. Such a reference of abstract class-type is given by calling a factory-like-method `java.awt.Toolkit.getDefaultToolkit()` (page 551 for `Toolkit`, page 529 for `Image`). The abstract `java.awt.Graphics`-class gets its reference from being a parameter in a triggerable method, see page 527. In other words: In the case of the triggerable `main()`-method, the `String`-parameter imports entire Java **String-OBJECTS**, whereas the triggerable `paint()` or `paintComponent()`-methods import solely a **REFERENCE** of abstract class-type `java.awt.Graphics` to a native piece of code!



The original idea of abstract classes was to have class-“templates” that do not allow (Java-)objects to be instantiated. Does the mechanism — to use REFERENCES OF ABSTRACT CLASS-TYPE TO NATIVE CODE — deface the idea of abstract classes? To a certain extent, yes; but this choice of approach allows the Java programmer to extend those classes, adding own Java-code, and furthermore, by using the “reference”-mechanism above, supplementing this own Java-code by the accessible native code. — This constitutes an example of practical considerations overriding conceptual simplicity or, putting it differently, conceptual rigidity.

Factory-like-methods: Factory-like-methods just return references of pre-existent objects or code; contrary to factory-methods that instantiate Java objects and return their references. Most of the factory-like methods are recognizable by their return-type that has been declared abstract. So, most factory-like-methods return references of some ABSTRACT class-type (`java.awt.Image`, `java.awt.Toolkit`) to entities of native code. Few factory-like methods return references of non-abstract class-type like the instance-method `java.lang.Object.getClass()` which returns the reference of a pre-existent `Class`-object. Another method acts either as a factory-method or as a factory-like method depending on the context: The `java.lang.Class.forName(String)`-method either loads a class, and after creating the associated `Class`-object returns a reference of it; or, if the class has been loaded already, just returns the reference of the then pre-existent `Class`-object.

Compare this to the consideration of factory-methods on page 151. Observe also those cases of abstract classes instantiable by factory-methods: The class `java.util.Calendar`, `java.text.DateFormat` and `java.util.TimeZone` on page 262!

Thus the modifier `abstract` describes a

class to be extended for instantiation, a

class-type of identifiers returned by factory-like methods, these identifiers reference non-Java code or data, and a

method to be implemented in an extended class.

Java: Interface-Type

Java: Interface-Type — Remember the considerations on multiple super-classes and the use of unimplemented methods to avoid problems of shadowing and precedence (page 136). The tradeoff for this advantage were non-implemented, abstract methods. Holding that in mind, consider a special kind of abstract class-construct in Java: Let **ALL** its methods be declared abstract and let **ALL** variables be

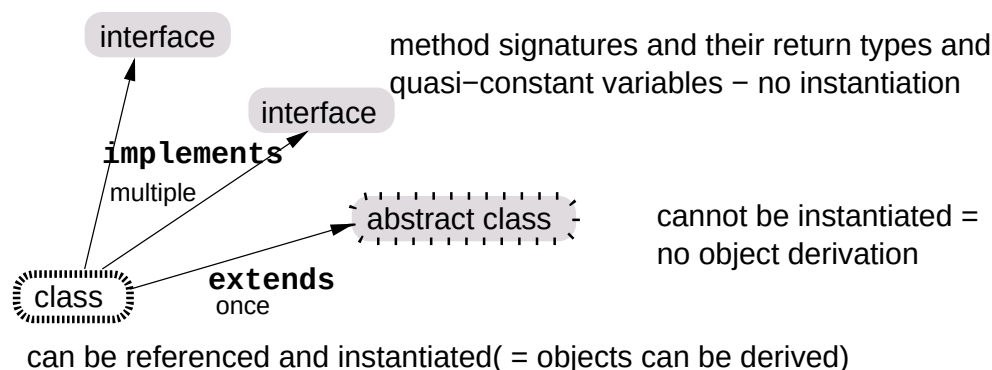
interface:
data-type

constant (or rather declared `final`, page 182). Any class still can extend only exactly one class of that kind. — Java's `interface`-keyword is used to declare types with ALL methods being `abstract` AND a single class may implement *multiply* many of those interfaces. The Java compiler recognizes all methods in interfaces to be implicitly declared `abstract` and all variables to be implicitly declared `final`.

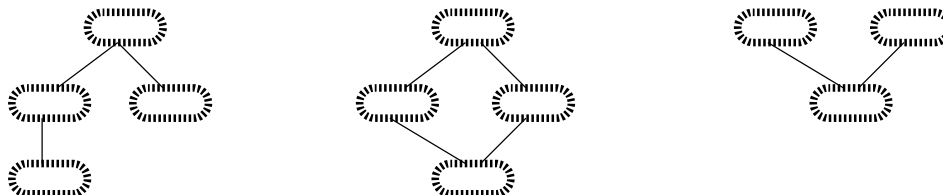
```
interface Ephemeral
{ aMethodName();
  primitiveType A_BLANK_FINAL;
  primitiveType A_CONSTANT = value;
}
// below a declaration, somewhere else in the source-code:
Ephemeral name; // identifier for objects of (interface-)type "Ephemeral"
```

Of course, interfaces become almost pure type-templates, but that suffices to become useful, as can be seen in the following paragraphs.

The graphic below visualizes the essential difference between interfaces and abstract classes in Java:



With Java-interfaces different kinds of class-hierarchies imaginable, only the first hierarchy could have a regular Java class at a place other than the lowest position:



With the `interface`-type in Java, method-declarations, or rather the introduction of their return-types and signatures, are implicitly `abstract`. These specifications can be transformed into runnable source-code:

```
interface Interface1{ void out1(); int CONSTANT = 1; }
interface Interface2{ void out2(); }
/* Java interfaces: Only furnish method-signatures and return-types,
   they never implement their methods!
   Java interfaces: Their method-declarations are implicitly abstract,
   variable-declarations are implicitly final.
*/

abstract class Abstract{ void outID(){ System.out.println( this.toString() ); } }
// Java object.toString(), by default, returns
// the system-unique tag given to each Java object.

/* Abstract classes allow fully-implemented methods,
   only methods explicitly declared abstract are not implemented.
   Abstract classes are never instantiated, therefore they can only have static variables!
*/

class JavaClass extends Abstract implements Interface1, Interface2
{
  public void out1(){ System.out.print( "implementation1 " + CONSTANT ); }
  public void out2(){ System.out.print( "\timplementation2 \t" ); }
}
/* A Java class can implement multiple interfaces
   (this realizes an aspect of multiple inheritance).
   A Java class can extend only one class. This includes an abstract class.
```

```

*/

class InterfaceAbstract
{ public static void main( String unused[] )
  {
    JavaClass o = new JavaClass();
    o.out1(); o.out2(); o.outID();
  } }

/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > javac InterfaceAbstract.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > java InterfaceAbstract
implementation1 1      implementation2      JavaClass@2457b6
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 >
*/

```

In short: **An interface is a collection of method-signatures, modifiers and associated return-types** (method-signature = method-name + parameter-list) **and constants**. An interface acts as a template for an implementing class to fill with source-code. One class can implement multiple interfaces. Interfaces sometimes even are empty and act solely as tags. interface: collection of signatures and constants

For example, abstract classes and interfaces are used non-trivially in structuring streams according to their input-/output-characteristics: See the four abstract classes on page 296. Abstract classes and interfaces are also used in structuring streams according to their data-transfer-characteristics. Consider also the two interfaces implemented on page 308.

Since interfaces never implement their methods, they cannot have **private** members. All methods in an interface are implicitly **abstract**! All variables in an interface are chosen to be implicitly **static** and **final**. That makes the VARIABLES OF AN INTERFACE CONSTANTS.

One class can implement multiple interfaces; but can extend only one - possibly abstract - class. An abstract class still may act like a template, either if all methods are declared abstract, or if methods are implemented with an empty-block `{}` to be overridden in a derived class. (Abstract classes, with all their methods implemented empty, are also called **adapter-classes**, see page 190.)

The **extends**-keyword gives the single inheritance functionality to Java. The **implements**-keyword gives a restricted functionality of multiple inheritance: Only interfaces can be “implemented” multiply by one class. interface: multiple inheritance

Interfaces are not special abstract classes, because a sub-class extends one abstract class, but a sub-class may implement more than one interface. And methods in interfaces are implicitly abstract, whereas they have to be fully defined in abstract classes, if they have not explicitly chosen there to be abstract.

Interfaces allow a Java-specific variant of multiple inheritance. Generally multiple super-classes cause conflicts in inheritance. In Java, interfaces are those class-like structures where multiple interfaces can contribute to one sub-class. Since interfaces do not implement methods there exists no multiple, potentially conflicting inheritance of methods: Two methods with the same signature coming from different interfaces cannot conflict, they will have only one implementation⁸¹.

Another view on interfaces stresses their ability to declare a common set of methods for otherwise unrelated sets of classes. Example: A storage-interface for classes (Serialization interface) declares, but does not implement, methods for storage and retrieval of objects.

Abstract methods, abstract classes and interfaces have a purely structuralizing function in the Java language. They should be used to give the code a better consistency and a clearer design. In almost all cases one could accomplish the programming goal without using these functionalities. interface: type, type structuring

If working on some practical problem and the desire arises to remember a particularly useful collection of methods and other members, then do group the signatures to an interface for future use and problem-specific implementation.

InterfaceType

—

TypeName

—

InterfaceType := TypeName — TypeName 82

The syntax of interface-declaration is given more precisely:

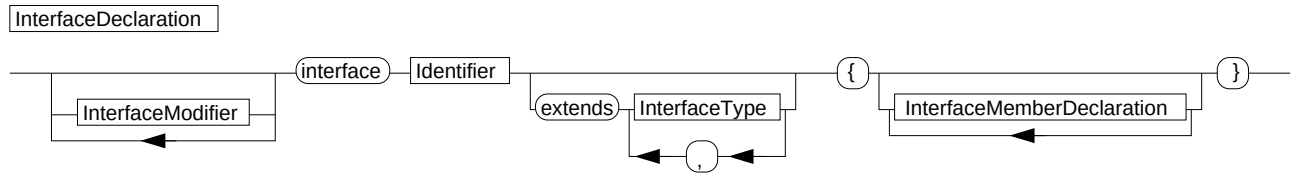
⁸¹In the case of variables, ambiguities have to be resolved by qualifying the ambiguous variable-name:

```

interface iName{ int x=1; }; interface jName{ int x=10; }
class name implements iName, jName{iName.x; /* gives 1 */ jName.x /* gives ten */}

```

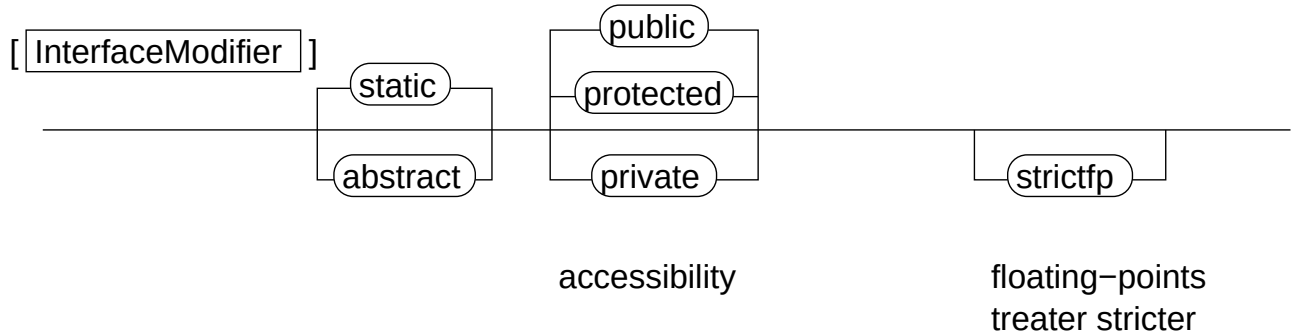
The standard way to solve ambiguities of identifiers uses qualification. As can be seen above, and as can be seen when motivating the use of packages on page 239.



InterfaceDeclaration := { *InterfaceModifier* } interface *Identifier* [extends *InterfaceType* { , *InterfaceType* }] *InterfaceBody*

InterfaceBody := "{ " { *InterfaceMemberDeclaration* } " }" — *InterfaceModifier* see below, *Identifier* 81, *InterfaceType* see above, *InterfaceMemberDeclaration* see below too.

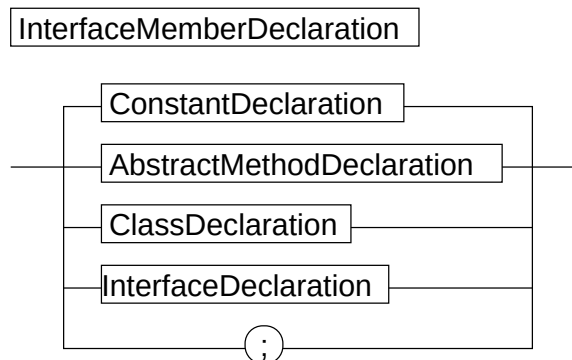
With Java-interfaces the modifier *abstract* has no meaning.



InterfaceModifier := one of public protected private abstract static strictfp

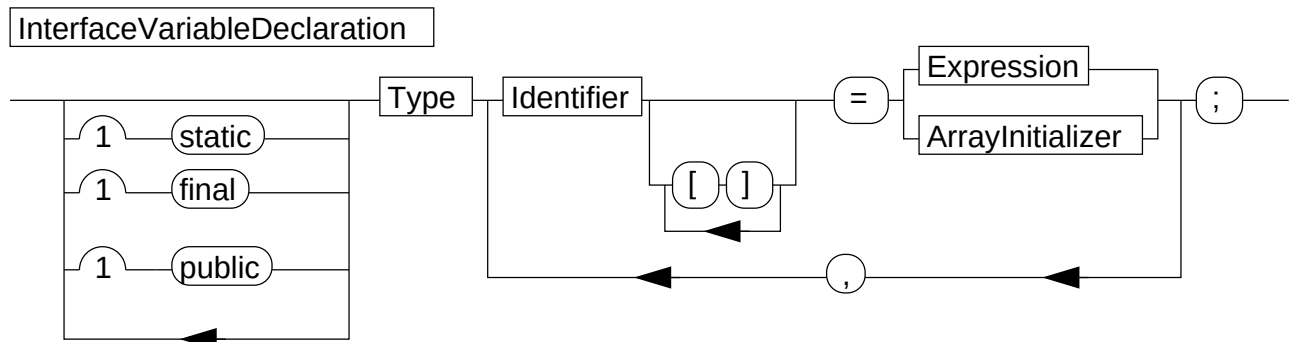
The keyword *implements* can be followed by the names of one or more super-interfaces. There exists no default super-interface.

Interface members are variables and method-signatures (there are no constructors, static-initializers, instance-initializers, nested classes)



InterfaceMemberDeclaration := *ConstantDeclaration* | *AbstractMethodDeclaration* | *ClassDeclaration* | *InterfaceDeclaration* | ; — *ConstantDeclaration* see below, *AbstractMethodDeclaration* 184, *ClassDeclaration* 146, *InterfaceDeclaration* 187.

Member-variables of interfaces are named constants, but their variable-initializer need not be a constant expression! (Compare that to instantiation of *final* variables, as indicated on page 182):



ConstantDeclaration := { *ConstantModifier* } *Type* *VariableDeclarator* { , *VariableDeclarator* }

ConstantModifier := one of `public static final` — Type 76, VariableDeclarator 121, Identifier 81, Expression 90, ArrayInitializer 236.

Interface variables can be deshadowed by casting a reference. Methods in interfaces are anyway implicitly abstract. Interfaces containing method-signatures and modifiers can be seen as templates to be filled with code when implemented in a class. (Variables are quasi-constants because they are implicitly `final`).

The following Java-interface only requires one single method to be implemented: A `public void run()`-method, which in the context of multithreading holds the source-code of threads.

```
java.lang Runnable{ public void run(); }
// tells "Java compiler, here is a run()-method of some thread!" (page 208)
```

Another interface of that kind has been introduced with Java Version 1.2:

```
java.lang.Comparable{ public int compareTo( Object o ); }
```

When implemented, the single method is supposed to return zero (0) if the objects are deemed “equal”, otherwise there has to be returned an `int`-value larger or smaller zero, indicating that the objects have a “larger than”-relation.

interface: tag

Interfaces that have an empty implementation-block serve as tags. Those tags keep attached to the objects derived from classes that implement these interfaces. These tags can serve as indicators for an agreement between programmers or the programmer and the Java compiler about the structure of elements that often are too complex to be realized as a simple method-declaration.

Examples are

```
java.io.Serializable
/* the source-code says to the Java compiler:
   "this class is serializable" (around page 309)
*/
java.util.EventListener
/* Java Runtime Environment:  "this class is an event-listener within an
   event-routing-infrastructure" (before page 288)
*/
java.rmi.Remote
/* Java Runtime Environment: "this class or object is part of a Remote Method
   Invocation (RMI)-infrastructure" (page 417)
*/
java.lang.Cloneable
/* "Java compiler, this class produces objects which are allowed to be cloned"
   (page 192)
*/
```

Extension, abstract classes, interfaces are just a method of structuring code hierarchically, there exists no deeper meaning hidden somewhere!

Interface, usually a means to make communication possible: 1. This may be A CONVENTION like in the Application Programming Interface (API) that furnishes the programmer with a library of programming-tools. 2. An interface may also be called the IMPLEMENTATION OF A CONVENTION to communicate, like a Graphical User Interface (GUI) with its software library. 3. Seldom, “interface” may be the name of a PIECE OF HARDWARE, like an Ethernet-card that forms the technical basis of some inter-computer communication-process. When time-dependent relations and communication are to be stressed refer to the notion of a $\rightarrow$ protocol.

In Java: (COLLECTION) A collection of method-signatures together with their modifiers and return-types AND constant-declarations. Classes announcing to implement interfaces define the methods and may assign values to blank finals (page 182). Classes can implement multiple interfaces.

(TYPE) As a type-identifier, a Java-interface can be used to PARTIALLY HIDE THE BEHAVIOR OF A CLASS. In this way an object or class of the interface-type only is seen as an implementation of the interface’s method-signatures and constants. If, for example, an object’s reference is casted to that interface-type, then the full complexity of a fully-implemented class or object is ignored.

This becomes especially useful in dynamic loading, where often only selected functionalities are needed to be known and not the full class-structure. (Therefore these Java constructs are named “interfaces”).

As a concrete example, consider the use of Java-interfaces in the source-code on page 404.

Also when defining the parameters in method-signatures as generally as possible, the parameters may be given a suitable interface-type; thus requiring only the absolutely necessary information and functionalities of the arguments! An example is given beginning with page 391, there the methods `setURL-StreamHandlerFactory()` and `setContentHandlerFactory()` only take arguments of interface-type.

(SORT OF MULTIPLE INHERITANCE) A Java interface gives a means to describe a class as an extension of multiple unimplemented super-class-like structures. Since the “super-classes” (=interfaces) are unimplemented, there exists no confusion about what implementation takes precedence if two “super-classes” declare the same identifier for a class-member (variable or method). Though a Java class can be the extension of only one Java (super-)class, it can implement multiple Java interfaces; which can be used to implement the concept of MULTIPLE INHERITANCE, TO A CERTAIN DEGREE.

(TAG) A TAG, a Java interface without any informational content, except its name. For example a Java interface (`java.io.Serializable`) can be used to tag (classes and) objects, to mark them as being serializable (transformable to a sequence of bytes). Or consider the interface `java.util.EventListener`, which indicates an underlying infrastructure (delegation event-model).

The notion of an adapter-class, as an abstract class having defined all its methods by empty blocks of code, appeared past the source-code on page 186.

The notion of an Adapter-class appears in two contexts:

(HELPER) Concerns only Java interfaces which declare many methods and those implementations that use only few of them. — An adapter-class implements such interface by defining all methods as empty (methods implemented with an empty-block `{}`). Then various extensions of such an adapter need only to override the desired methods (Various implementations of that interface would have to implement all the many methods of the interface, emptily). For examples concerning event-handlers, see page 580.

(CASTER) Suppose an object is given, being only cast to an interface-type. In these cases the interface is also called adapter. (example page 404)

Adapter-classes always implement an interface or are used anonymously for a temporary overriding.

Remember that interfaces are just aggregates of method-signatures, modifiers and return-types which indicate the methods to be implemented by the programmer. Or interfaces contain no method-signatures at all, then they remind the Java compiler or a fellow programmer that some rules or conventions have to be observed in working with the class that implements such an interface⁸².

Java: Comparing (2) and Copying

Java: Comparing (2) and Copying — Apart from implementing the interface `java.lang.Comparable`, as hinted at on page 189, comparing Java objects usually means asserting their equality. As could be seen, beginning with page 162, this task proved to be non-trivial, producing notions such as shallow- and deep-equal. Remember the `equals()`-method of the class `java.lang.Object` which, by default, tests objects on shallow equality. The parameter of this method has been defined to be of class-type `Object` because at that level of generality, this type represents the only information available about possible class-types. When needing to override the `equals()`-method use the following standard code to assure a proper comparison process:

⁸²Interfaces and abstract classes are also treated in the TechTips of Nov. 6th 2000 and the one of June 12th 2001.

```

public boolean equals( Object ref )
{ if( ref instanceof specialType )
  { specialType s = (specialType)ref;
    compare();
  }
  else return false;
}

```

If necessary, make sure that the `equals()`-method returns true always when the values of the method `hashCode()` are the same for both objects in question. So much for comparing.

Having in mind deep- and shallow equality, remember that assigning variables of primitive-type means duplicating their values (page 83). Assigning the identifiers of reference-type means duplicating a pointer (page 144).

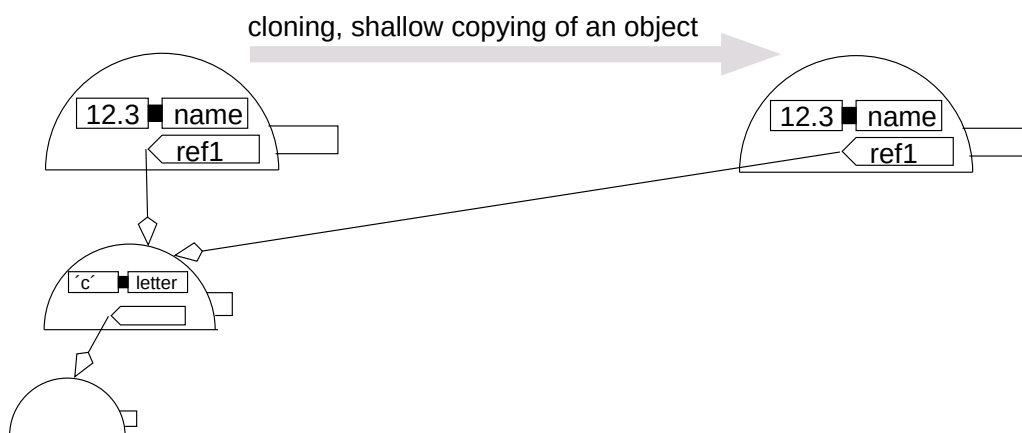
— Clone: 1. (object-model) One of two objects differing only in their identity-label, thus having the same behavior and the same state. 2. (Java) One of two objects; both created having the same class-type and having the same state. The same state means, having the *same values* within class-members of primitive-type and having the *identifiers of reference-type pointing to the same code* outside the objects. — Thus, instance-members of reference-type, which are given by pointers to other objects, make both objects share the *same* object-graph — the same graph of references. A cloned object is also called a shallow copy of a given object. (see graphics below)

— Cloning: Duplicating, copying an object. Also called shallow copying, since references to other objects - though copied - remain pointing to the same objects. Cloning or shallow copying extends the object-graph.

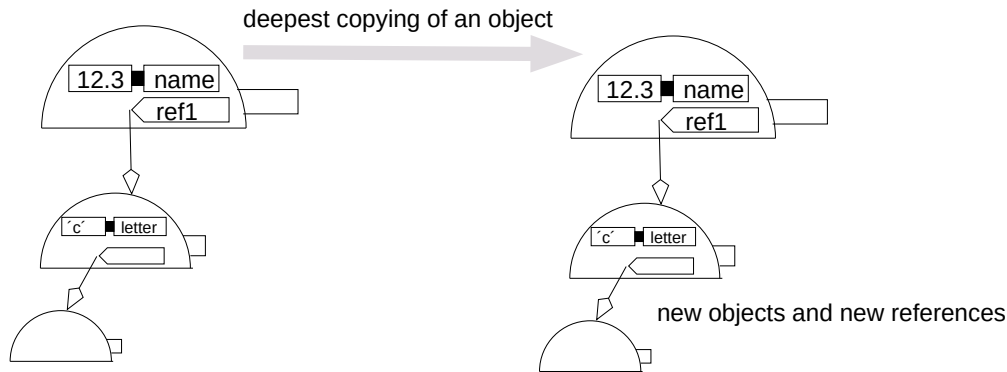
— Deep cloning, Deep copying: Duplicating the entire object-graph.

Remember the notion of so-called reference-types that have been introduced in Java (page 143): The value of a variable with reference-type is an address of data (the value of a variable of primitive-type is the data itself); therefore an assignment to the identifier of a variable of reference-type changes the address (an assignment to the identifier of a variable with primitive-type changes the data).

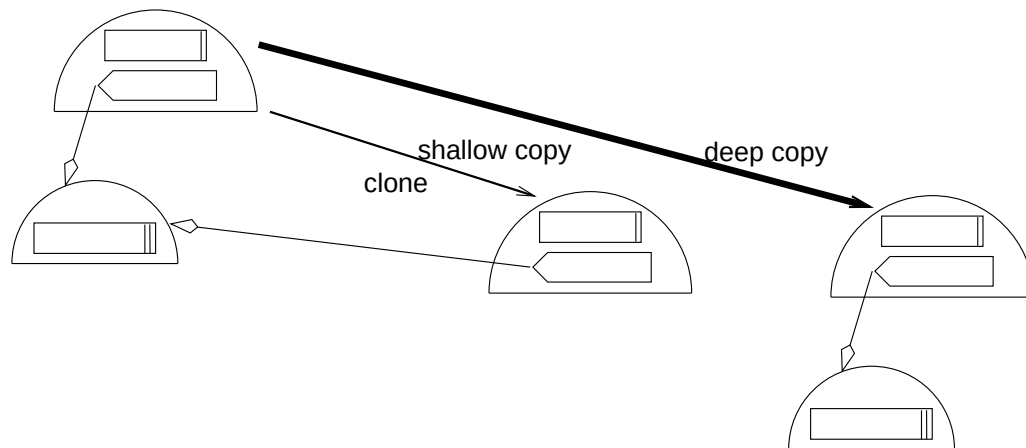
Consider a concrete example: An object can be copied by re-instantiating the object and filling it with the same information, data and references. This process is called cloning the object, sometimes also “shallow copying” the object.



References that are copied remain pointing to the original object. — But an object can be copied another way: If all objects of the object-graph (or rather the graph of references) are re-instantiated and the references changed accordingly, then that copying-process is called the deep(est possible) copying of an object:



Between simple cloning or shallowest copying and the deepest copying, as presented in the graphics above, can be imagined intermediate ways of copying. — Assessing the equality of objects has the same possibilities. The choice of how to copy and how to compare his objects, deep or shallow or in between, remains entirely with the programmer.



In the graphic above, for the case of shallow copying or cloning, can be seen again: Assigning identifiers of variables that are of reference- or class-type means only copying the references; the referenced object and its data is left untouched! In the case of deep copying, the entire object-graph has to be reproduced.

Relating the copying above, to the simple assessment of equality in comparing objects, there can be seen the following: Shallow equality of object(-graph)s follows from any kind of established deep equaling! And objects that are some kind of deep equal, always are shallow equal, because even if the references differ, the contents of the referenced objects have to be identified as equal.

The concepts, introduced in the previous paragraphs, have to be given some implementation-basis in Java: The super-class of all classes in Java is the `Object`-class (`java.lang.Object`, as introduced on page 164). An instance-method of this class is called `clone()`-method. This method can be used only if the class-extension implements the empty `java.lang.Cloneable`-interface, otherwise its use generates a `CloneNotSupportedException` (-object) at run-time. Thus the `Cloneable`-interface is used as a flag to signal the Java compiler the permission to clone the object and to furnish a method that actually produces the clone. This means, the programmer of the class has to consent to the cloning attempts on derived objects.

The protected method `clone()` of the `java.lang.Object`-class creates a new object (the clone) and returns a reference of this object; this makes it a factory-method (page 151). The created object is referenced by a returned identifier of class `Object`, to keep the `clone()`-method overridable along the entire class-hierarchy.

```
class CloneCandidate implements Cloneable //extends java.lang.Object (automatically)
{ int i;
  public Object clone() // public or protected
  { try
    { return super.clone();
    } catch( CloneNotSupportedException e ){ return null; }
  } }
class Clone
{ public static void main( String unused[] )
  { CloneCandidate clone, test = new CloneCandidate();
    test.i = 1;
    clone = (CloneCandidate)test.clone();
    System.out.println( test.toString() + " " + test.i );
  } }
```

```

        System.out.println( clone.toString() + " " + clone.i );
        clone.i = 2;
        System.out.println( test.toString() + " " + test.i );
        System.out.println( clone.toString() + " " + clone.i );
    } }

/* consider varying this source-code in two respects:
--- OMITTING THE ENTIRE CLONE METHOD-DECLARATION
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > javac Clone.java
Clone.java:12: clone() has protected access in java.lang.Object
clone = (CloneCandidate)test.clone();
^
1 error
--- OMITTING THE TRY-CATCH-CLAUSE produces
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > javac Clone.java
Clone.java:6: unreported exception java.lang.CloneNotSupportedException;
        must be caught or declared to be thrown
return super.clone();
^
1 error
--- OMITTING THE CLONEABLE INTERFACE
produces a CloneNotSupportedException which is handled by the catch-clause
producing a NullPointerException at run-time

andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > javac Clone.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 > java Clone
CloneCandidate@2f6684 1
CloneCandidate@738798 1
CloneCandidate@2f6684 1
CloneCandidate@738798 2
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCObjectOrientation2 >
*/

```

Why has the `clone()`-method to be re-implemented trivially in the `CloneCandidate`-class above (Its original protectedness should guarantee it to be called along the class-extension path)? Exactly, the compile-time error criticizes trying *to access* a protected method from OUTSIDE the extension-hierarchy and the `java.lang` package⁸³. Only if the `clone()`-method is elementarily overridden inside the no-name package, then it becomes *accessible* from within the no-name package. Furthermore, if the method is overridden with the modifier `public` inside a public class, then the `clone()`-method, properly qualified with package and reference to the object, becomes *accessible* from the entire Java Runtime Environment!

Why does the `clone()`-method copy the member of primitive-type (`int`-variable) properly? In the form above, the `clone()`-method instantiates a new object as a shallow copy of the original (values of primitive-types are duplicated as well as references, but not the referenced objects!) This explains why the members of primitive-type are cloned. Extend the example above by adding a member of class-type `String` to the class `CloneCandidate` and test the result of the given `clone()`-method. default
clone()
copies shallow

OUTLOOK: Comparing objects and copying and cloning objects involves a lot of further intricacies, for example cloning arrays, which have been omitted in this text. Cloning and comparing classes becomes often of relevance in database-related programming tasks⁸⁴.

Summary: Object-Orientation and Java Classes

Summary: Object-Orientation and Java Classes — What setting suggests the structuring of code in an object-oriented way? Generally, when larger pieces of code are to be written, and when the kind of problem to be solved allows structuring in advance, closely related to the real-life problem. Object-oriented programming usually can be seen as a form of top-down approach to programming, a strategy-first-(programming)-tactics-second-method. The object-oriented approach of describing and formulating programming tasks often is chosen by groups of programmers working on one large project. Apart from the usefulness of pre-structuring larger pieces of code, especially when developing software in groups of more than one person, the object-oriented approach may give a frame of reference that eases coordination in coding. This contrasts an approach which first solves individual problems and then integrates them. Actual coding usually oscillates between the pure concepts of structuralizing the whole and developing functional elementary parts.

⁸³Many pieces of source-code have no package-declaration as introduced on page 240. Source-code without any such package-declaration runs in the so-called default-package (introduced on page 241). Furthermore, all classes of the `java.lang`-package are always automatically “imported” into the code running in the current Java Runtime Environment, so that `java.lang`-classes can be accessed from the default-package (and all other packages) without qualification (`Object` instead of `java.lang.Object`, `String` instead of `java.lang.String`). Nevertheless, the `java.lang`-package and the default-package are distinct packages. An introduction into using packages is given beginning with page 239 and page 431.

⁸⁴For more on cloning see also the TechTips of March 6th 2001.

Object-oriented analysis includes the concept of proceduralizing code, thus giving it well-defined programmer interfaces (input/output parameters in methods, methods and public variable in objects and classes) so that the code becomes a black box (encapsulation) which can be easily re-used. The idea of offering services and hiding implementation-related details (encapsulation) can be applied to methods and even to entire classes.

Object-oriented structuring of tasks is treated in many books, independently of any programming-language.

Java and the Class-Concept

index Classes, especially when given with their class-structure, also constitute hierarchically-organized subroutine libraries. Classes can be seen as something like an hierarchical index to subroutines, like an index to topics of a book (`java.lang.Math`-class). Classes allow to aggregate primitive-types or other objects and classes may thus form a class-specific data-collection (instance- or member-variables) and classes may define class-specific operations (`java.lang.String`, `java.lang.Double`).

type Classes can be seen as collections of functionalities attached to their data, thus implementing custom-definable types (class `VarInteger` in source-code on page 142). Thus classes give flexibility to the type-concept. And therefore variables — variables that define the state of an object or static variables of a class — can be objects themselves. Thus variables of an object can be objects derived from other classes (page 141).

program A class with a `main()`-method holds a trigger for outside activation, for starting a flow of commands, that can be transferred to other constructors, methods, static or non-static initializers as well as finalizers and `catch`-clauses. Collections of classes referencing each other — one of those classes having implemented a static `main()`-method — are what comes closest to a Java program.

Some of the Java-specific class-properties are compiled below:

- Classes can be predefined by given class-libraries, called packages, like the `java.lang.String`-class, or classes have to be defined by the programmer as shown by the program on page 142.
- In some cases, classes or class-hierarchies even may be definable locally inside a method.
- Aside from the `static` modifier, other modifiers like `public`, `protected` and `private` control the accessibility of identifiers.
- In Java the existence of class-hierarchies is indicated by defining class-structures using the formulation `class childClass extends parentClass` (The notions class-hierarchy and class-structure has been introduced and described according to their relevance on page 135.)
- A modifier `abstract` `class className` preceding a class-declaration marks the class as non-instantiable. Nevertheless an abstract class may contain fully implemented methods. (Within the example in the page-reference above, a `Car`-object must belong either to the ones geared mechanically or geared automatically. A `Car`-object without that specification is a “prototype”, a mere template, a precursor of the `Car`-model and should not be instantiable.)
- Multiple inheritance in Java can be realized by so-called interfaces that furnish constants and method-signatures plus return-types of methods to be implemented. A declared class can implement multiple interfaces `class className implements interface1, interface2, ...`. But interfaces also act as types of their own: If necessary, identifiers of objects can be cast into an interface-type!

Concept: Polymorphism/Monomorphism

Polymorphism is the name for cases in which the same identifier can be used for multiple tasks. That especially concerns reusing identifiers in overloading and overriding or in implicit type-casting.

- Overloading methods: Within a class, methods with the *same name* but *different parameter-lists* can be declared. (presented in the context of method-declarations on page 124)
- Overriding methods: If there are methods with identical signatures in a class-hierarchy, then the method is used which is implemented inside the class itself or inside the closest super-class. (presented in the context of class-extension on page 167)

- Widening references of objects: Objects, which have been instantiated from some sub-class-type, always can be used in contexts that work with a super-class-type (sub-type polymorphism). This is the result of the automatic- or implicit casting between identifiers of reference-type. (presented in the context of type-casting on page 169)

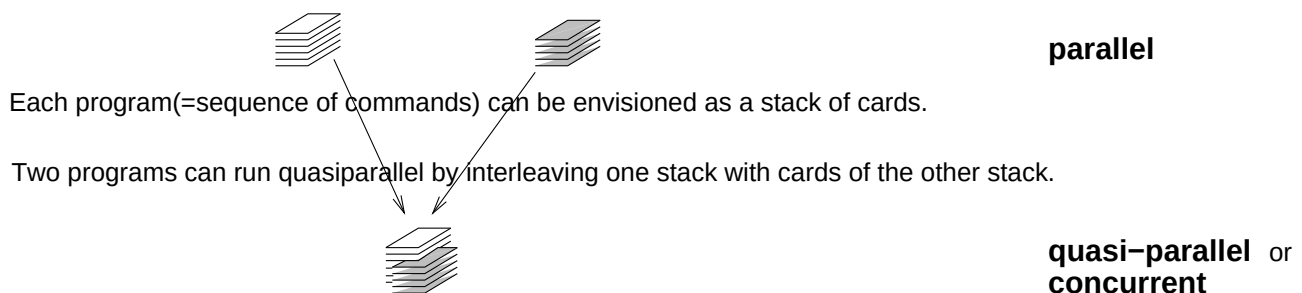
A language that is statically typed is said to have type-monomorphic identifiers. The type-casting possibilities of Java make it an in-between.

Concept: Encapsulation

Sometimes real-life-problems can be solved in the top-down-approach by simply cutting them into a set of sub-problems. Solving those sub-problems independently from each other constitutes an encapsulation strategy. (Observe the closeness to the object-oriented approach and proceduralization.) Encapsulation is another word for black-box-strategy. Encapsulation may mean collecting code in methods that communicate with the rest of their environment only over values in their parameter-list and return-values, without accessing variables non-local with respect to the method's block. For example a sorting algorithm may be written as a method with an array as an input and the sorted array expected as the output. Then the method can be used by anybody without knowing its interior. Another form of encapsulation uses an private instance-variable of an object to be only accessible through the object's methods.

Structuring Code: Multithreading

Structuring Code: Multithreading — Generally data-manipulation by an algorithm means working step by step through a sequence of commands. This can be seen as the origin of the notions “task”, “thread” or “process”. Especially for computers with multiple processors the idea of running code truly parallel in time may be tempting, but even on a computer with one processor, code may be run quasi-parallel as the following graphic suggests:



As simple as it might look conceptually when using *non-interacting* pieces of code, the aspect of relating different pieces of *interacting* code relatively in time often causes more difficulties than could be expected.

Concept: Concurrency and Parallel Computing

Concept: Concurrency and Parallel Computing — Some application-related (high-leveled) examples may demonstrate the usefulness of concurrent programming: In a graphics-program the rendering of a picture on the screen may take some time; during that time-interval the user might like to open a drawing-toolbox. Or an Internet-browser loads large amounts of data (files of text, code or images) and the user wants to fill out some template or go to another page. Writing program-code with multiple threads gives an application-program higher reactivity to the end-user’s input and thus saves the end-user waiting-time. (In a one-processor-system the original task may slow down only unnoticeably.) Even when things do not happen faster — this is especially the case in one-processor-systems — things may run smoother, better organized, when structuring code to run (quasi-)parallel in time.

A simple computer, a simple CPU with a single processor, does not process code truly parallel: Programs are sequences of commands, maybe containing branches or loops, but essentially each program is worked through sequentially, that means step by step.

Nonetheless the speed of the computer often can be used to let actions *seem* to look happening parallel: In a terminal-window or a WhatYouSeeIsWhatYouGet text-formatting program, the keyboard input generally appears ‘immediately’ on the screen. An audio-file can be listened to and edited ‘at the same time’. Because of the speed of the computer, any programming-language can be used for coding programs that give the impression of a quasi-parallel running of tasks:

```
while(true)
{
    if(keyBoardInput()) processKeyBoardInput();
    if(imageLoading()) loadNextSmallSectionOfImage()
}
```

Contrary to such simple looping, as indicated above, programming-languages may furnish more elaborate programming-constructs for (quasi-)parallel code structuring. A programming-language with constructs which allow (quasi-)parallel program-runs, may use the resources on machines with multiple processors more efficiently. Thus the language relieves the application-programmer of concurrent programming on the level of the (native) operating-system and yields a better structured code:

single
processor
computers
work
sequentially

```

Thread keyboard =
new Thread(new Runnable()
    { public void run(){ processKeyBoardInput() } } );
Thread loadImage =
new Thread(new Runnable()
    { public void run(){ loadImage() } } );
keyboard.start();
loadImage.start();

```

(Above, the code of the thread is found within the `run()`-method of an anonymous object of interface-type `Runnable`, this object is instantiated from an anonymous implementation/extension of the interface-type `Runnable`.) Therefore, a programming-language which supports concurrency, only adds to its features another method of structuring code. Methods for structuring code, so far presented for Java, have been proceduralization and object-orientation.

another
method of
structuring
source-code

Observe that these features of a programming-language are offers to the programmer. Using these features shouldn't be accepted as mandatory, but if properly used, these language-features may reveal some distinct advantages.

What kind of programming tasks are eased by using the feature of concurrency in a programming-language? INDEPENDENT TASKS ARE THE CANDIDATES FOR ACQUIRING OWN THREADS IN CODING. "Independent" means here that the results can be produced with few relations to the rest of the program and that the result may not be needed right away. Such tasks can be user-related-tasks as described above (interaction with computer) or problem-related tasks as, for example, many parallel and independent calculations in pixelwise image-enhancement or simple matrix-multiplication. Data-flow diagrams, with few interrelations often reveal the potential for concurrency. USUALLY ANY KIND OF COMMUNICATION-PROCESS FAVOURS A FORMULATION USING PROGRAMMING STRUCTURES THAT SUPPORT PARALLELISM. Dividing a programming-problem into relatively independent, isolated parallel threads of source-code constitutes another form of encapsulation.

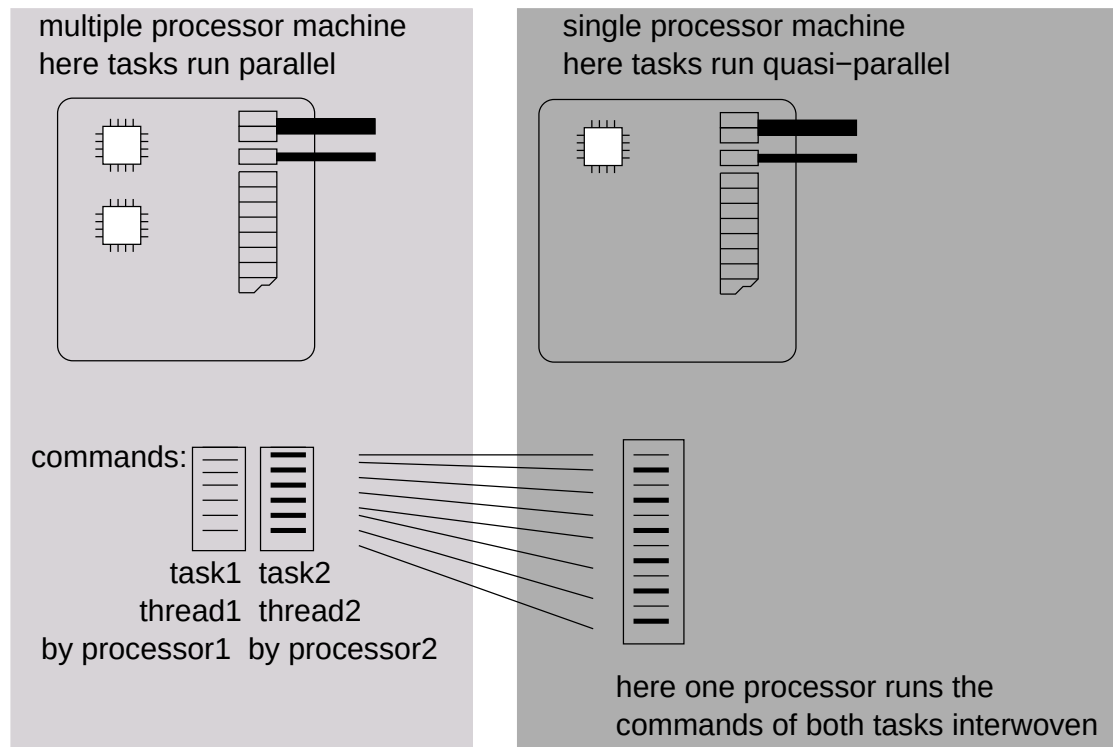
Further aspects of concurrency in general are considered in the sections about networks of computers.

Concept: Threads

Concept: Threads — Often some tasks have to be accomplished in (quasi-)parallel manner: Analyzing mouse-clicks, getting keyboard-information, organizing the graphics on the screen and maintaining an Internet-connection. That is possible by using multiple processors. (Active modems and active graphics-cards have their own microprocessors and usually their own Random Access Memory (RAM).)

Instead of using multiple processors, the tasks may be broken down into chunks that can be processed alternately (like closing a zipper, where each line of teeth represents the flow of commands of another task.) and thus working on them quasi-parallel. The organizational overhead for storing and restoring the processor-states usually becomes acceptable⁸⁵ with programming-problems that naturally support a parallel approach:

⁸⁵This organizational overhead, related to the overhead of re-formulating the original problem in a sequential form, is called granularity of the original problem.



Quasi-parallel running of threads involves granting each thread a share of computing-time (also called thread-scheduling), therefore quasi-parallel running of programs is also described by the denotation “**concurrency**”.

In **multithreading** the tasks run (quasi-)parallel using the **same memory-space**. If there exists more multithreading than one processor, then the tasks may run truly parallel, still using the same memory-space; but if there exists only one processor, then the tasks run *quasi*-parallel (using the same memory-space).

Later, at the end of this section, the concept of **multiprocessing** assigns each task a **separate space in multiprocessing the Random Access Memory**.

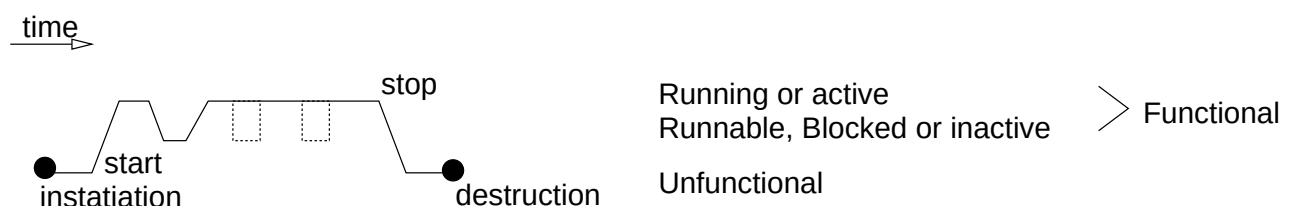
Both concepts usually occur in computing, for example one single CPU may let run several programs independently with their own assigned memory-space (multiprocessing) and have some of these programs exhibit quasi-parallel functionalities (multithreading).

THE JAVA LANGUAGE PROVIDES *Multithreading* LANGUAGE-CONSTRUCTS, so each thread has access to all other objects that are accessible within the current Java Runtime Environment (JRE).

Unlike any other programming structure presented up to now (proceduralization, class/object-model, programs as sequences of commands), threads are time-critical constructs:

First, already running code has to instantiate a thread-control object for each thread to be used; then the new thread can be started making it a functional- or running thread. The two basic states of a thread, unfunctional and functional can be divided into various sub-states, depending on the notions and concepts used in multithreading. Here the functional thread is given two sub-states: Interrupted (inactive) and Running (active).

A Thread's Time- and State-Table



Thus up to now, there are three basic states of a thread (Unfunctional, Runnable and Running). An inactive thread may wait for results transferred from other running threads. In this case the thread has to do some purely problem-related waiting, this kind of waiting has to be fully controllable from the source-code of the programming-language. Threads having been inactivated from source-code are also called **Blocked**. During the waiting-time another thread may use CPU-time, memory or network- Blocked thread

connections. In the graphic above, the blocking of the thread is indicated by the indentation of the line just after the starting of the thread.

Another case of temporary thread-deactivation is indicated by the dotted boxes in the graphic. This case can be exemplified by considering a central processing unit (CPU) with a single processor: In this context running multiple threads quasi-parallel may mean that a thread has to take turns with other threads in using CPU-time. So the threads have to be deactivated temporarily from the computer-platform's side. The operating-system has to distribute the system-resource fairly among the existent threads; this process on the platform's side, independently from any application-related influence, is also called **thread-scheduling** and is discussed in the following section. Limiting the existent threads to the number of available processors proves not useful, even on multiprocessor-computers; therefore, the operating-systems have to practice thread-scheduling. Threads that have been temporarily inactivated by the system's thread-scheduling are also called **Runnable** (those threads could run if there were enough processor-resources available).

That adds another name for the inactive state of a thread, namely **Inactive**. Depending on whether the inactivity is induced from the side of the operating-system or even programmer-induced thread-scheduling, the thread-state is called **Runnable**. Or if the inactivity has been made necessary by the running of the source-code, then the inactive state is called **Blocked**.

concurrency on
system-level

As can be seen, the implementation of threading and multiprocessing involves tasks close to the level of the hardware. Therefore, to make application-programming easier, the basic tasks of thread-management, the distributing of CPU-time among threads (also called **basic scheduling**) ideally should be done by the operating-system or the platform-software (programmed by the system-programmers).

The **higher level thread-management** within the source-code on the level of a higher programming-language may mean

concurrency on
level of
programming-
language

- giving a thread a higher priority than others,
- making one thread wait for another or
- giving one thread exclusive access to a variable, or
- making a piece of code uninterruptable by the scheduling-process

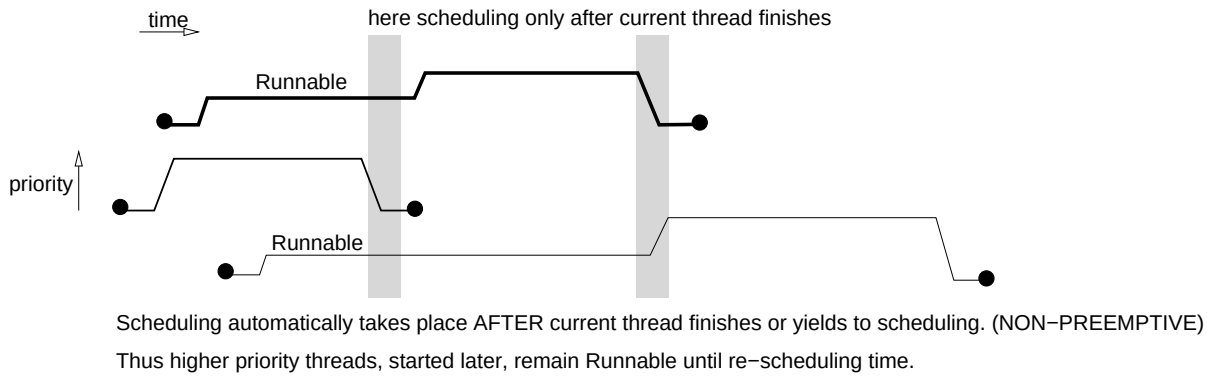
These tasks have to be made controllable by the programming-language and often are used by the application-programmer.

Concept: Scheduling of Threads (Non-/Preemptive and Priorities)

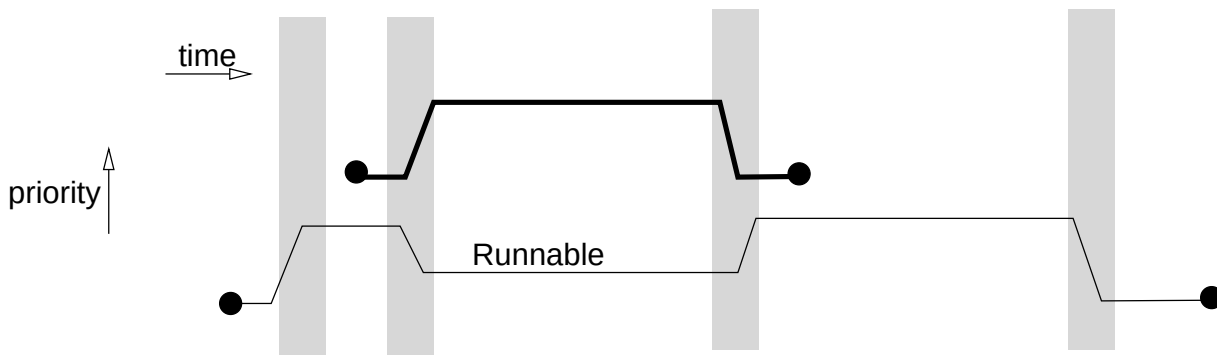
Concept: Scheduling of Threads (Non-/Preemptive and Priorities) — This section presents ways to answer the question: How can operating-systems support multithreading, what forms of thread-scheduling are imaginable? (priority-based, time-slicing). And due to deficiencies of some older operating-systems what pieces of thread-scheduling may even need to be contributed by the application-programmer?

A set of threads {threadA, threadB, threadC} could be treated like a set of programs: The threads would be given an order and worked through sequentially: First threadA, then threadB and finally threadC. The application-programmer could influence that order by giving different priorities to the threads, then the thread with the highest priority could come first, the computer would be left to chose the order of the threads of same priority. If another thread of paramount priority appeared (maybe created by the first thread), then the re-arrangement of the order could take place after the first thread has finished. This presented an example of a **priority-based thread-scheduling that acts overall non-preemptive**. (All the following scheduling-mechanisms should be ideally given as readily implemented mechanisms of the operating-system!)

priority-based
scheduling

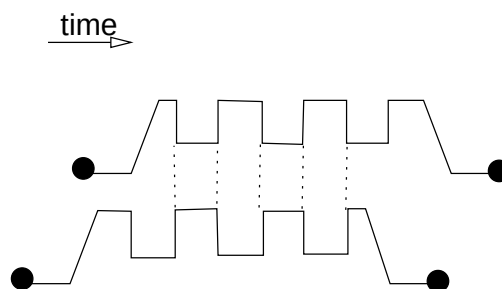
A form of Priority-based scheduling:

If the simple thread-scheduling mechanism, that has been mentioned above, can be extended so that a thread of higher priority CAN INACTIVATE A RUNNING THREAD of lower priority, then the scheduling-mechanism is called **PREEMPTIVE priority-based scheduling-mechanism**.

Preemptive Priority-Based Scheduling:

A **preemptive scheduling-mechanism** for processor-time can inactivate a thread from outside, there is no waiting needed until the questioned thread yields his resources to other threads. preemptive scheduling

A preemptive scheduling-mechanism on threads of the same priority, giving each thread an equal share of processor-time, is called **time-slicing**. time-slicing

Time-Slicing of threads with same priority:

On the level of operating-systems an overall preemptive scheduling generally is preferred, because one thread cannot block all others and thus block the entire system. With preemptive scheduling the thread advantage of preemptive multithreading

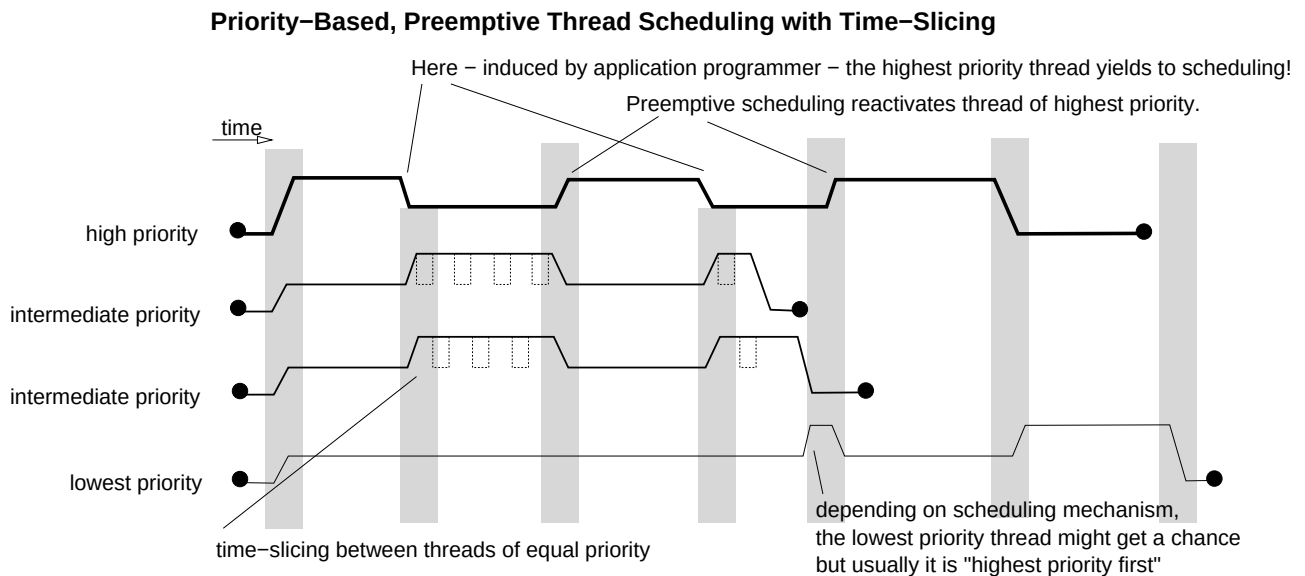
that blocks all other threads can be inactivated by the operating-system and does not block the entire system.

When using a non-preemptive scheduling of processor-time, each thread *alone* decides whether to yield the control over the processor to some other thread, *or not*. So, even application-programmers may be confronted with scheduling tasks in multithreading, this means application-programmers would have to organize the work which actually should be done by the operating-system.

A **preemptive priority-based scheduling-mechanism with time-slicing** can be described as follows: Threads with the highest priorities are always the first to become Running, even if they have to inactivate threads of lower priority (= turn them from Running into Runnable). Threads of highest priority have the right to start whenever they want. If necessary, threads with lower priority have to yield immediately to threads of higher priority. Unless the thread with the higher priority yields or pauses or unless, on a multiprocessor-system, there becomes a processor available, threads with lower priority have to wait in the Runnable-state. When two threads of same priority are “running”, time-slicing means that they are broken into pieces of code that take turns in running.

An example may give an illustration: Consider each thread as a stack of cards, in which each card represents a smaller list of processor-commands and the order of the cards gives the order in which the thread is worked through. The stack of cards with the highest priority comes first. If a stack with a higher priority occurs while being halfway through a stack, then the current stack is abandoned and the new one is worked over. Only after that stack of higher priority has been finished, the old half stack is processed further. That represents a preemptive (interruption of current work) priority-based (stacks with higher priority come first) mechanism.

If the stacks are of the same priority, then (round robin) time-slicing means that first are processed five cards of the first stack, then five of the second, and so on until after the last stack, the first stack is processed further. This is repeated until all stacks have vanished.



If there are more than two threads of the same priority in the state of being Runnable, a simple strategy for distributing processor-time is given by a round-robin-scheme. The **round-robin-scheme** means that the currently Runnable threads are arranged into a circle-order and after each time-slice the current thread is inactivated and the thread that stands next in the line becomes activated.

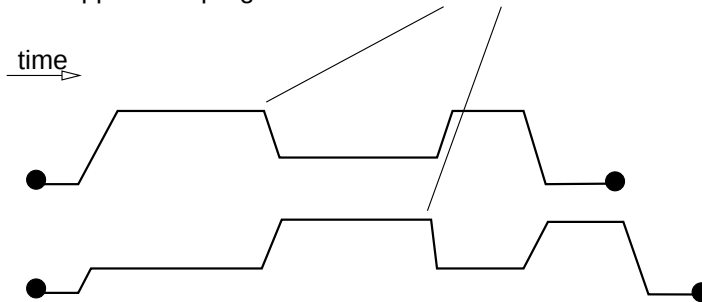
The example above has been given for a computer with a single processor. The graphic above has to be changed when considering a computer with two processors: First, the highest priority thread would continuously run on a processor of its own. Both threads of intermediate priority would occupy the resources of the second processor, working continuously on a time-sliced basis. Finally, the thread of the lowest priority would be run on that processor which terminates its threading-services first. (Try to draw a related variant of the graphic above.)

Usually some kind of priority-based multithreading with or without time-slicing is already implemented in programming-languages or operating-systems that support multithreading. In many programming-environments (that includes Java Runtime Systems), the application-programmer has to set priorities and to trigger thread-changes. Triggering thread changes becomes especially important in environments without time-slicing, otherwise one thread could monopolize a processor.

The Importance of Yielding in non-preemptive Thread-Scheduling

Threads with same priority – NOT time-sliced:

The application programmer has to initiate thread state changes

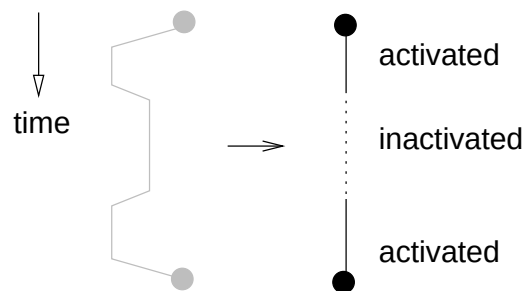


The Java language has a built-in priority-based thread-management. But the Java Language Specification [JLS] does not fix whether threads of the same priority are time-sliced or whether they get processor-time in turns when the Running-thread yields.

Java and thread-scheduling

At this point the application-programmer has to write multithreaded code with enough yield-points to avoid, for example, delayed or interrupted audio output or sluggish keyboard reactions. If necessary, the application-programmer even may have to devise an own version of time-slicing.

The graphical notation of thread-states will be used also in a vertical notation as indicated by the graphic below:



Java: Basic Thread Resource Distribution on a Java Platform

Java: Basic Thread Resource Distribution on a Java Platform — The (software-)implementation of the Java Virtual Machine (JVM) emulates a processor. On a single-processor computer, multithreading within the Java Runtime Environment (JRE, which contains the implementation of the Java Virtual Machine) means that the threads run quasi-parallel. That means at one point in time exactly one thread is Running, all others remain Runnable or Blocked. If the implementation of the Java Virtual Machine (JVM) works on a multiprocessor machine, then various Java-threads *should* be able to run truly parallel.

The implementation of the Java Virtual Machine uses native code within the platform's operating-system⁸⁶ to realize some of the multithreading functionality. That makes the virtual machine faster but gives a Java-untypical platform-dependence in scheduling threads of the same priority. — Some UNIX implementations of Java are not preemptive on threads of the same priority, threads of the same priority have to yield or to stop to give other threads a chance to run. Some *Windows*-implementations of Java use the time-slicing mechanism on threads of the same priority, thus much better realizing the quasi-parallelism of multithreading!

Due to this platform-dependence of multithreading services, it can happen that, for performance reasons, an application-programmer would have to extend the system's scheduling-mechanism. So the programmer can use the threading system of the native platform, which promises fast and more effective results or the programmer can write own Java-routines for lower-level threading-support (own time-slicing routines) that guarantee portability.

Such considerations are only necessary when programming processor-intensive code (large mathematical calculation, extensive symbol manipulation in strings or images). Whereas other code may not be in need of optimized threading functionalities: For example code, handling a lot of input/output operation, usually

⁸⁶Circumventing a conventional programmers interface especially to invoke low level methods is also called trapping.

has to do a lot of waiting anyway, maybe with some burst of activity reading or writing files to disks or network-sockets or some communication with other programs. User-interactive code usually waits even more time for user-input, time enough to usually achieve the necessary threading-services by simply calling a `yield()`-method from time to time.

The Java program below allows to determine whether the Java platform uses a time-sliced scheduling-mechanism or not:

```

/*- Two threads with a loop run concurrently in producing terminal output:
--- An interwoven output approximating (12121212121212121212)
    indicates that the platform furnishes time-slicing or multiple processors.
--- A sequential output approximating (11111111112222222222)
    indicates that the platform leaves part of the scheduling to the Java programmer;
    for example in the form of calling yield()- or interrupt()- or sleep()-methods,
    which have been outcommented below.
*/
class TheThread extends java.lang.Thread
{ int i;
  TheThread(int i)
  { //super(); // not needed, because implicitly called! See page 166
    this.i = i;
    System.out.println( "Thread no." + i + " starts:" );
    this.start();
  }
  public void run()
  {
    for( int c = 1; c <= 10; c++ )
    {
      System.out.println( "Thread no." + i + " step " + c );
      //try{ Thread.sleep(1);}catch( InterruptedException e ){ System.exit( 0 );}
      //this.interrupt(); // Mac OS, Linux no difference
      //this.yield(); // Mac OS, Linux no essential difference
    } }
  public class SchedulingOfJavaPlatform
  { public static void main( String unused[] )
    { new TheThread(1); new TheThread(2);
    } }
}
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] marycosway% javac SchedulingOfJavaPlatform.java
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] marycosway% java SchedulingOfJavaPlatform
Thread no.1 starts:
Thread no.1 step 1
Thread no.2 starts:
Thread no.1 step 2
Thread no.1 step 3
...
Thread no.1 step 9
Thread no.1 step 10
Thread no.2 step 1
Thread no.2 step 2
Thread no.2 step 3
...
Thread no.2 step 9
Thread no.2 step 10
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] marycosway%
*/

/*
C:\WINDOWS\JavaCodeFiles\Basic1\SCMultithreading>javac SchedulingOfJavaPlatform.java
C:\WINDOWS\JavaCodeFiles\Basic1\SCMultithreading>java SchedulingOfJavaPlatform
Thread no.1 starts:
Thread no.2 starts:
Thread no.2 step 1
Thread no.1 step 1
Thread no.2 step 2
Thread no.1 step 2
Thread no.2 step 3
Thread no.1 step 3
...
Thread no.2 step 8
Thread no.1 step 7
Thread no.2 step 9
Thread no.1 step 8
Thread no.2 step 10
Thread no.1 step 9
Thread no.1 step 10
C:\WINDOWS\JavaCodeFiles\Basic1\SCMultithreading>
*/

```

An output that gives 10 times thread no.1 and then 10 times thread no.2 may not use any automatic scheduling in the Java Runtime Environment (JRE). Whereas a thread with a mixed printout of thread no.1 and thread no.2 uses either time-slicing or the platform uses multiple processors. To keep multi-threaded code in Java truly portable, a Java programmer might consider to incorporate a rudimentary scheduling into the Java code. A time-slicing CPUScheduler written in Java is introduced in [JT].

Java: Handling Single/Isolated Threads

Java: Handling Single/Isolated Threads — Where, in Java source-code, does an application-programmer produce threads? How are threads run in Java? First consider two standard threads that already have been used: The main- or `main()`-thread in which runs the `main()`-method of any Java-application and the event-dispatch-thread in which runs the `paint()`-method of an applet.

application's
main-thread

A Java-application, a program which is given by a class with a static `main()`-method, uses a system-specific thread which is triggered by the external “`java className`”-command. Code of this main-thread is found inside the `main()`-method and all blocking methods called from there. Only one single “`main()`”-thread can be running in a given Java Runtime Environment (see also page 65).

For the basics on applets, see page 67. An applet (in need of an additional supporting software-environment, like a Java enabled Internet-browser) uses Java-threads furnished by the supporting software-environment. The applet-thread is called from the main-thread of the applet runtime environment. In such a specific applet-thread, applets run their central methods like the `init()`, `start()`, `stop()` and `destroy()`, for more on those methods see page 396.

applet-thread

Another, and one of the more prominent threads appears to be the event-dispatch-thread of the Graphical User Interface (GUI), which is used by source-code found inside the `paint()`-method and used by source-code within other event-handling methods. For basics on the notion Graphical User Interface, see page 507, for the role of the `paint()`-method see page 575.

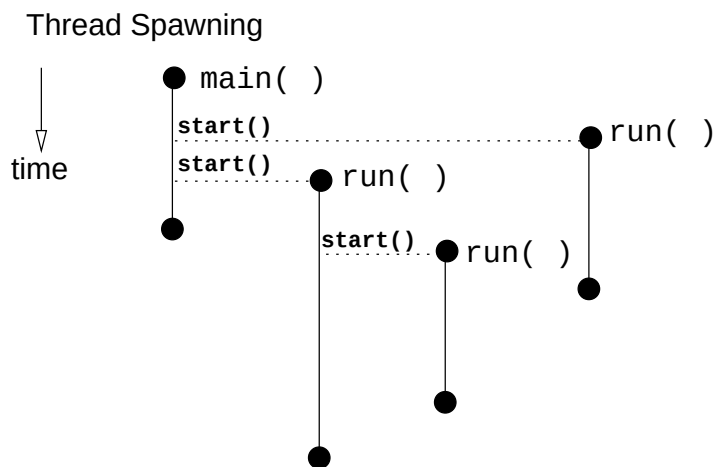
event-
dispatch-
thread

Furthermore, every Java Runtime Environment (JRE) uses a garbage-collection mechanism to automatically erase unreferenced objects – and thus unused objects – from the system's Random Access Memory (already hinted at on page 152, introduced on page 444). This mechanism runs as a thread of its own in the Java Runtime Environment.

garbage-
collection-
thread

Thus the Java Runtime Environment (JRE) and its applet-running extensions furnish a set of already implemented threads. Nevertheless Java enables every application-programmer to implement own threads, as has been already demonstrated by the source-code within the previous section.

In Java-applications, threads can be started from the `main()`-method as illustrated below:



Methods or constructors of classes may start threads. Or those classes may declare a `run()`-method which contains the code of a thread and thus can be run (quasi-) parallel to other threads. Threads can be started from other threads as indicated by the illustration above. But the seed that started this activity is the `main()`-method and the associated thread. Threads can outlive the `main()`-method or any other thread from which they were started. In Java, threads can be instantiated or/and started from any block or allocation expression.

The basic infrastructure necessary for running programmer-defined threads in the Java Runtime Environment (JRE) is given by one class and one interface: The `java.lang.Thread`-class allows to instantiate and control a `Thread`-object, this object represents a kind of dashboard- or control-object for the associated thread. The `java.lang.Runnable`-interface⁸⁷ furnishes a tag for indicating the existence of a `run()`-method in the implementing class. Finally, the implemented `run()`-method contains the source-code of the thread.

⁸⁷Unfortunately the thread-state “Runnable” has the same name as the Java-interface “`java.lang.Runnable`”, which happens to be pure coincidence and has no further meaning. The distinction should be clear from the context in which the denotation appears.

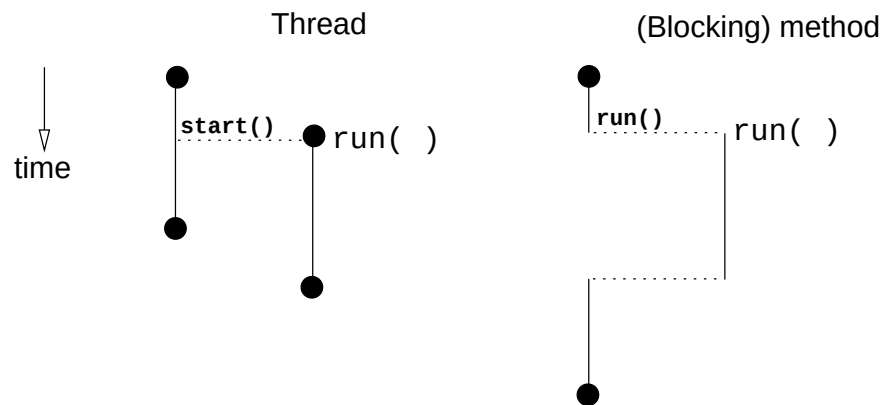
java.lang.Thread a class; the instantiated Thread-object creates the thread and furnishes means of control over the thread (“dashboard-object”). The `start()`-method is given by a Thread-object, this method triggers the concurrent or parallel run of the `run()`-method. The `start()`-method returns immediately after having initiated the code-processing by the thread. (The `start()`-method is a non-blocking method as introduced on page 231.)

java.lang.Runnable an interface; all classes of this type implement the `run()`-method containing the source-code of a thread. The Thread-class itself implements the Runnable-interface trivially. So either an extension of the Thread-class has to override the `run()`-method non-trivially; or the Thread-object can be made to control the `run()`-method of some other Runnable-object (A reference to that Runnable-object in the Thread-object’s constructor delegates that method-call. Details are given in the next section.).

So the thread’s source-code is given within a `run()`-method. This `run()`-method is triggered automatically by calling the `start()`-method of a Thread-object. What if the `run()`-method is called directly? A direct call of the `run()`-method would let the code within the `run()`-method run within the calling thread! No new thread would have been created, that call would be a regular so-called blocking method-call. — The code within the `run()`-method runs in a thread of its own ONLY if triggered by the `start()`-method of an associated Thread-object!

`run()`-call =
(blocking)
method-call

`start()`-call =
thread
activation



Since in Java, the control of the thread and the source-code of a thread may be separated, there are different ways to spawn (= instantiate a thread object, start the `run()`-method) a thread in Java, these alternatives are presented in the following section.

Java: Spawning a Thread

Java: Spawning a Thread — Here are presented four variations of getting a thread to run in Java; later are given implementations of these ideas as Java source-code. Essentially there are two procedures of thread-declaration:

- The class which extends the `java.lang.Thread`-class has to contain the `run()`-method⁸⁸. This may be indicated symbolically by

class *className* extends Thread.

This is a clean procedure, but problematic if the to be extended class already extends a (super-)class. Those special cases can be treated as follows:

- The argument-list of the Thread-constructor contains a reference to an object. Therefore that object has been derived from a class which implements the `java.lang.Runnable`-interface — the object is of type Runnable — and contains a `run()`-method of its own. Instead of using its own `run()`-method, the Thread-object now takes the thread’s code from the Runnable-object. This may be indicated symbolically by

new Thread(new class implements Runnable).

There are various ways to let the argument of the constructor implement the `run()`-method:

- The argument is a reference to an object with a `run()`-method. (The class from which the referenced object has been instantiated, implements the `java.lang.Runnable`-interface.)

⁸⁸Since the Thread-class implements the Runnable-interface by default, an extension of the Thread-class has to contain the definition of the `run()`-method.

extending
Thread,
overriding `run()`

Thread-
constructor
references a
Runnable-object

- Inside the argument the Runnable-interface with its run()-method is implemented anonymously.
- The argument of the Thread-constructor is the this-keyword and thus refers to the enclosing - instantiated - class that implements the Runnable-interface and therefore contains the run()-method. This makes it possible to localize all thread-relevant declarations and commands in one object.

Some of these thread-definition techniques are also used with applets (see page 401). Revisit the list above when considering the source-code of the following subsections:

Thread-Object contains run() -Method

Each Thread-class (of course including its extensions) implicitly implements the Runnable-interface, so a run()-method can be implemented directly within the (extended) Thread-class.

```

/* Thread class-extension: conceptually clean,
   but problematic if an already derived class is to become threadable
   (for those cases omit this example and consider the next ones!)
*/
class ThreadWithRun extends Thread
{ int i;
  ThreadWithRun( int i )
  { //super(); // unnecessary because implicitly called, see page 166.
    this.i = i;
    this.start();
  }
  public void run()
  {
    for( int n = 0; n < 4; n++ )
    { System.out.println( "thread no." + i + " active" );
      //try{ Thread.sleep( 10 ); }catch( InterruptedException e ){ System.exit( 0 ); }
      // sleeping may be seen as a faint emulation of time-slicing
      //Thread.yield();
      // above, Mac OS no difference, Linux, main thread ends after both threads ended
      //this.interrupt();
      // above, Mac OS no difference may act like a premature System.exit( 0 ),
      // above, Linux no difference
    } } }
public class ThreadExtension
{ public static void main( String unused[] )
  { new ThreadWithRun( 1 ); // anonymous or
    ThreadWithRun X = new ThreadWithRun( 2 ); // with unused identifier
    System.out.println( Thread.currentThread() + " main()-thread ends." );
  } }

/*
--- this.start() could be omitted from the constructor; but then
    the threads have to be started in the main()-method, for example like this:
    X.start(); or new ThreadWithRun(1).start();
--- Observe that the sequence of statements in the main()-method isn't reflected
    in the sequence of the output.
*/
/*
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% javac ThreadExtension.java
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% java ThreadExtension
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
Thread[main,5,main] main()-thread ends.
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro%
*/

/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > javac ThreadExtension.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > java ThreadExtension
thread no.1 active
thread no.1 active
thread no.1 active
thread no.1 active
thread no.2 active
thread no.2 active
thread no.2 active
thread no.2 active
Thread[main,5,main] main()-thread ends.
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading >
*/

/*
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>javac ThreadExtension.java

```

```

C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>java ThreadExtension
thread no.1 active
thread no.1 active
thread no.1 active
thread no.2 active
Thread[main,5,main] main()-thread ends.
thread no.1 active
thread no.2 active
thread no.2 active
thread no.2 active
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>
*/

```

Extending the Thread-object often complicates matters, because the extending class cannot be a sub-class of any custom build class-hierarchy any more; except by implementing some interfaces. Observe also that the `start()`-method of the Thread-class starts the `run()`-methods.

run() -Method in Anonymous Class

The constructor of the Thread-class can take an argument which is a reference to an object that implements the Runnable-interface: A simple use offers itself by implementing and instantiating an object of type Runnable as an anonymous class (and object):

```

/* Here the constructor of the thread contains an object,
   this object is an on-the-spot anonymous implementation
   of the Runnable interface.
*/
public class ThreadAnonRunnable
{
    public static void method( String s )
    {
        for( int i = 0; i < 4; i++ )
        { System.out.println( s );
          Thread.yield(); // Mac OS, Linux no difference
          //Thread.currentThread().interrupt(); // Mac OS, Linux almost the same as above
          //try{ Thread.sleep( 10 ); }catch( InterruptedException e ){e.printStackTrace();}
          // above, a sort of time-slicing effect
        } }
    public static void main( String unused[] )
    {
        //----- two anonymous threads:
        new Thread
        ( new Runnable()
          { public void run(){ method( "thread no.1 active" ); }
          }
        ).start();
        new Thread( new Runnable()
                    { public void run(){ method( "thread no.2 active" ); } } ).start();
        System.out.println( Thread.currentThread() + " main()-thread ends." );
    } }
// Observe that the sequence of statements in the main class isn't reflected
// in the sequence of the output
/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > javac ThreadAnonRunnable.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > java ThreadAnonRunnable
thread no.1 active
thread no.1 active
thread no.1 active
thread no.1 active
Thread[main,5,main] main()-thread ends.
thread no.2 active
thread no.2 active
thread no.2 active
thread no.2 active
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading >
*/

/*
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>javac ThreadAnonRunnable.java

C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>java ThreadAnonRunnable
Thread[main,5,main] main()-thread ends.
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>
*/

```

Here the Runnable-interface is instantiated into being an anonymous inner class, where the block of the `run()`-method contains just a method-call. This presents a very compact way of producing threads from already written methods! (see the description of non-blocking methods on page 231)

Thread-Object and run() -Method in Different Classes

The Thread-object acts as a sort of control- or dashboard-object. Another object of interface-type Runnable contains the thread-implementation as a run() -method. The Thread-object's constructor references the Runnable-object. This means that control and code have to communicate between different classes. Thus CLASSES STARTING A THREAD, WITHOUT EXTENDING THE CLASS Thread IMPLEMENT THE Runnable-INTERFACE⁸⁹ and let this thread be controlled by another object of class type Thread:

```
/* The constructor of the thread contains a reference to an object.
   This object contains, its class marked by the Runnable-interface, a run()-method!
*/
class ContainsRun implements java.lang.Runnable
{
    String aString;
    ContainsRun( String s ){ aString = s; }
    /* this.start(); not possible in constructor
       because Thread-object is member of another class! */
    public void run()
    {
        for( int i = 0; i < 4; i++ )
        { System.out.println( aString );
          try{ Thread.sleep( 100 ); }catch( InterruptedException e ){ System.exit( 0 ); }
          //Thread.yield(); // Mac OS, Linux no time-slicing
        } } }

public class ThreadRunnable
{ public static void main( String [] unused )
  { new Thread( new ContainsRun( "thread no.1 active" ) ).start();
    new Thread( new ContainsRun( "thread no.2 active" ) ).start();
    System.out.println( Thread.currentThread() + " main()-thread ends." );
  } }
/*
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% javac ThreadRunnable.java
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% java ThreadRunnable
thread no.1 active
thread no.2 active
Thread[main,5,main] main()-thread ends.
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro%
*/

/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > javac ThreadRunnable.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > java ThreadRunnable
thread no.1 active
Thread[main,5,main] main()-thread ends.
thread no.2 active
thread no.1 active
thread no.2 active
thread no.2 active
thread no.1 active
thread no.1 active
thread no.2 active
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading >
*/

/*
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>javac ThreadRunnable.java

C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>java ThreadRunnable
Thread[main,5,main] main()-thread ends.
thread no.2 active
thread no.1 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>
*/
```

The class, that contains the main() -method, instantiates two objects of type ContainsRun (thus also of type Runnable), then each of these two objects is wrapped with a Thread-class thus instantiating the associated Thread-objects. The start() -method of the Thread-class starts the run() -methods. — For the sake of shortness two threads of the same kind have been implemented.

Thread-Object is Member in Class of run() -Method

Another way of avoiding to use a class that extends the Thread-class, can be seen when declaring a class-member of type Thread and associate it with the run() -method. Thus a java.lang.Thread-object is declared within the class that implements the java.lang.Runnable interface:

⁸⁹Observe how the use of the notion Runnable-interface oscillates between being a(n interface-)type of a given object, a template for defining a method or a tag or marker of an object. The interface is all that at the same time.

```

/* in this example
   all explicit threading procedures
   are encapsulated in a single class: Compound
*/
class Compound implements Runnable
{
    Thread localThread;
    String aString;
    Compound( String s )
    { aString = s;
      localThread = new Thread( this );
      // as usual, "this" refers to the enclosing object
      localThread.start();
    }
    public void run()
    {
        for( int i = 0; i < 4; i++ )
        { System.out.println( aString );
          try{ Thread.sleep( 10 ); }catch( InterruptedException e ){ System.exit( 0 ); }
          //localThread.interrupt(); // Mac OS, Linux no difference
          //Thread.yield();           // Mac OS no difference, Linux main-thread ends last
        } }
    public class CompoundThread
    { public static void main( String unused[] )
      { new Compound( "thread no.1 active" );
        new Compound( "thread no.2 active" );
        System.out.println( Thread.currentThread() + " main()-thread ended." );
      } }
}
/*
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro% javac CompoundThread.java
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro% java CompoundThread
thread no.1 active
thread no.2 active
Thread[main,5,main] main()-thread ended.
thread no.1 active
thread no.2 active
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro%
*/

/*
andreadipietro@linux:~/JavaCodeFiles/Basic1/SCMultithreading > javac CompoundThread.java
andreadipietro@linux:~/JavaCodeFiles/Basic1/SCMultithreading > java CompoundThread
thread no.1 active
Thread[main,5,main] main()-thread ended.
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
andreadipietro@linux:~/JavaCodeFiles/Basic1/SCMultithreading >
*/

/*
C:\WINDOWS\JavaCodeFiles\Basic1\SCMultithreading>javac CompoundThread.java

C:\WINDOWS\JavaCodeFiles\Basic1\SCMultithreading>java CompoundThread
Thread[main,5,main] main()-thread ended.
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
thread no.1 active
thread no.2 active
C:\WINDOWS\JavaCodeFiles\Basic1\SCMultithreading>
*/

```

Observe that the thread now can be started in the constructor because the Thread-object is a member of the class in question.

Basics of Controlling a Single Thread

Basics of Controlling a Single Thread — A Thread-object may be instantiated from the Thread-class or from a sub-class of the Thread-class. Since the Thread-class implements the Runnable-interface implicitly, every Thread-object has a default run()-method which is empty. Every Thread-object of this kind represents a complete yet trivial thread, unless the run()-method has been overridden or the Thread()-constructor contained an argument referencing an object of (interface-)type Runnable. The ensuing alternatives of coupling a Thread-object to non-trivial thread's code were demonstrated in the section before; a simple solution may be just to override the run()-method when extending the Thread-class.

The simple model of thread-states may begin with states like Unfunctional/Functional and may be extended to a set of states Unfunctional/Inactive/Active; which may be further refined to the states Unfunc-

tional/Blocked/Runnable/Running. That model of thread-states becomes even more refined in the context of the Java programming-language, there a thread can take five different states:

Initialized The instantiation of the Thread-object makes the thread existent. The existence can be checked by calling the Thread-object's `isAlive()`-method.

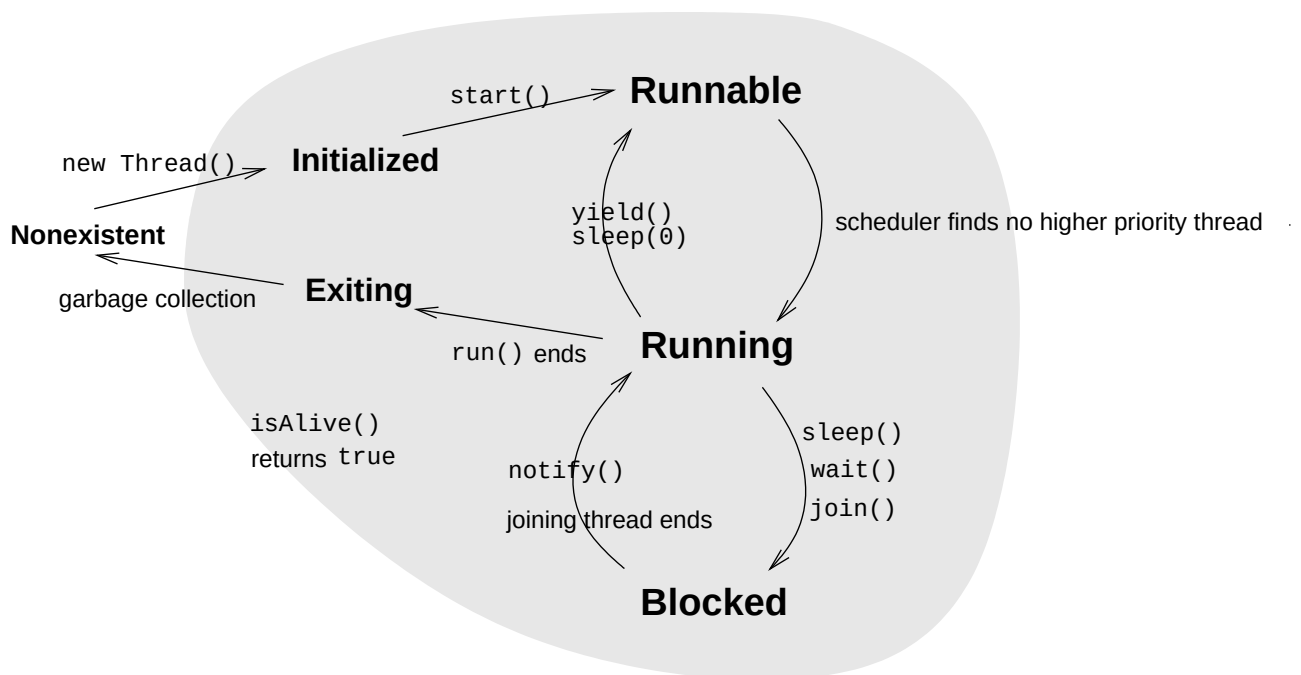
Blocked Method-calls from the Thread-object like `sleep()`, `wait()`, later `join()` turn the thread's state into being Blocked. After a default waiting-time in the case of `sleep()`, after an indeterminate waiting-time by method-calls like `notify()`, or when another thread finished its work in the case of a `join()`-call, the thread changes into being Running or Runnable.

Runnable If the computer had arbitrarily many working processors at its disposal, then each Runnable thread would immediately become Running on one of the processors. A thread that is Runnable is able to run but has been blocked by the scheduler of the implemented Java Virtual Machine (JVM). Usually the programmer or the user of the program shouldn't see how the scheduler of the implemented Java Virtual Machine distributes processor-time among the threads that are Runnable, oscillating them between being Running and Runnable giving the impression of quasi-parallelism. A Java programmer may change a Running thread into being Runnable by calling the methods `yield()` or `sleep(0)`.

Running The `start()`-method, inherited from the Thread-class, makes the thread Running or Runnable. A running thread has a processor working through its code.

Exiting Waiting until the thread finishes the `run()`-method renders the thread Exiting. Then, the thread cannot be re-started, only the Thread-object remains.

Nonexistent The garbage-collector de-references the Thread-object and erases it from the computer's Random Access Memory.

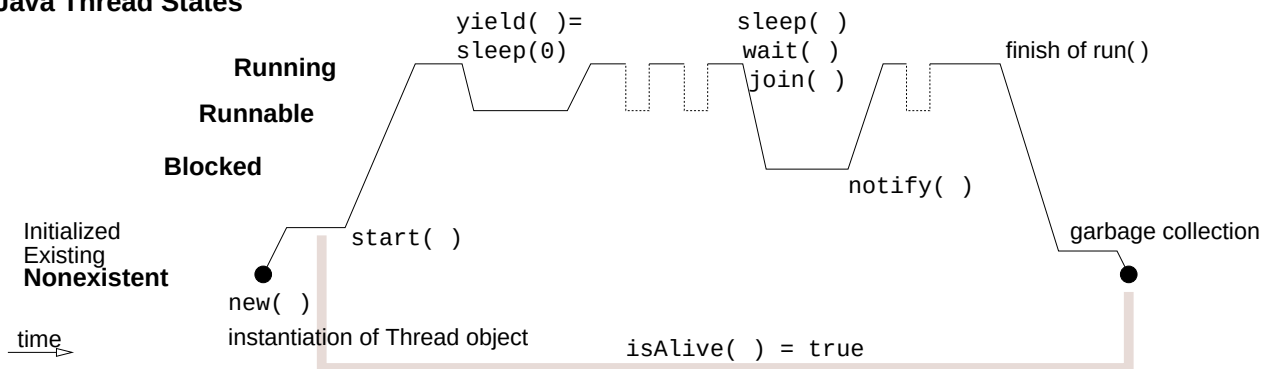


The Thread-class and its object furnish methods that inform about the state of the thread:

- `public String getName()` The default object referenced is "Thread"+*n* (where *n* represents an integer).
- `public final native boolean isAlive()` returns true sometime before(!) the call of the `start()`-method, until sometime after the end of the `run()`-method for the thread referenced by the qualifier.
- `public final int getPriority()` returns the priority of the specifying thread. A Java-thread can have a priority ranging from 1 to 10 in integer steps.

- **public boolean `isInterrupted()`** This instance-method returns true if the qualified thread's `interrupt()`-method has been called. This method just returns the boolean value of a thread-specific "interruption"-flag, nothing more! The thread's state has not been changed!
- **public static boolean `interrupted()`** This class-method returns true if the CURRENTLY running thread has experienced a call of its `interrupt()`-method. AND calling the static `interrupted()`-method resets the currently running thread's "interruption"-flag to false. The thread's state has not been changed!
- **public static native Thread `currentThread()`** returns a reference to the Thread-object associated with the `run()`- or `main()`-method that made this call. This static method of the class Thread is called within pieces of source-code (usually methods which have been called) to get a reference to the Thread-object in whose thread they are running. This call can be seen as a sort of "Where am I?"-question.

Java Thread States



The graphic illustrates the thread-states in Java and some of the methods for controlling the thread, others are listed below:

- **public final void `setName( String )`** The default object created is "Thread"+*n* (where *n* represents an integer).
- **public final void `setPriority( int )`** Sets the priority of the qualifying-thread. The Thread-class has three static variables `MIN_PRIORITY=1`, `NORM_PRIORITY=5` and `MAX_PRIORITY=10`.
- **public synchronized native void `start()`**, qualified by the Thread-object's identifier, starts the associated `run()`-method and thus introduces a new Runnable or Running thread.
- **public static native void `sleep( long )`** Blocks the qualifying-thread for the number of milliseconds given by the argument. (Keeps all locks, as described later on page 223.) Many operating-systems cannot resolve individual milliseconds!
- **public static native void `yield()`** Thread-class-method that changes the state of the currently Running method into Runnable, thus allowing another thread to become Running.
- **public void `interrupt()`** A call of that instance-method of a Thread-object sets a thread-specific "interrupted"-flag to true. The exceptions thrown are or have to be so-called checked exceptions (page 280), which have to be caught in a try-catch-clause. In that process the thread's "interrupted"-flag automatically is set to false. On calling the `interrupt()`-method, usually the Java programmer has to make the called thread test on being interrupted and let the thread itself initiate the change of thread-state. An exception to this rule is given by a Blocked state caused by invoked `sleep()`, `wait()` or `join()`-methods, there the call of the `interrupt()`-method automatically throws a `java.lang.InterruptedException`.

Interrupt: 1. Signal, often coming from the hardware, that forces the processor to stop with its current task, save its status and proceed with another program, the so-called interrupt-handler. Interrupts are used for registering mouse-movements, mouse-clicks, keyboard activity, process-scheduling and/or other network-tasks. Hardware interrupt handling usually has top-priority. 2. Stopping a thread's run from outside, rendering the thread Runnable, Blocked or Unfunctional. 3. In Java: A thread's internal flag which can be set to true or false.

(Consider scrutinizing the TechTip from April 2000 for further information on interruptions.)

Concept: Handling Multiple/Communicating Threads

Concept: Handling Multiple/Communicating Threads — This section assumes that low-level mechanisms like preemptive priority-based scheduling with or without time-slicing are working well, because, for example, none of the implemented Java-thread is considered to be processor-intensive. No time-slicing with several threads of the same priority anyway makes it necessary to insert `yield()`-commands at points where other threads of the same priority can take over. Some older Java versions and operating-systems may not effectively interpret the `yield()`-command or some third thread, like the main-thread, may render the `yield()`-call ineffective. To assure portability it may also be assumed at this point that the Java programmer has written own Java code for organizing the scheduling of threads. The mechanics of this task is described in specialized books like [JT].

This should — to a large extent — justify the basic assumption of this section: All threads are working parallel, they are either Running, Blocked or Nonexistent. The distinction between the states Running and Runnable fades in the context of high-level thread-communication. **IN THIS SECTION, ALL THREADS MAY BE THOUGHT AS RUNNING TRULY PARALLEL, EACH ON A MACHINE OF ITS OWN. AND THE THREAD-SCHEDULING IS TAKEN CARE OF BY THE OPERATING-SYSTEM!** Setting priorities in this context just means to inform some black-box scheduling-mechanism about personal preferences. That can be seen as the outset of the APPLICATIONAL-LEVEL APPROACH to the coordination of multiple threads.

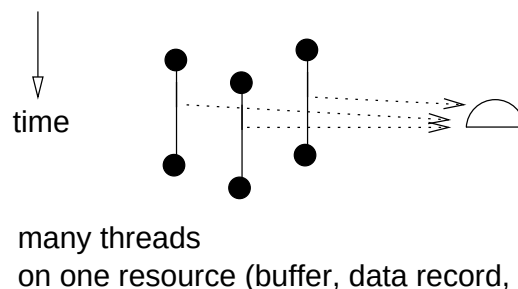
Remember another aspect of multithreading: Basically, multithreading presents just another way to structure source-code! Compare that to the proceduralization of code which is made possible by using methods. Or reconsider the indexing and aggregation of code which can be done using class-hierarchies. When programming with one thread, the code-statements are always coordinated as a simple sequence in time, whereas programming with multiple threads has the added complexity of coordinating code-segments that run truly parallel in time. Some of this organization-task may be taken over by the computer-platform, but much of this coordination may prove to be problem-related and therefore has to be done by the application-programmer.

Otherwise a lack of cooperation of threads may lead to inconsistent results: For example two airline offices could sell one seat on an airplane to two different customers because the update mechanism in the database does not coordinate the database-access properly. On the side of thread-programming observe that threads share the same memory, each thread can change objects, especially variables, of another thread without difficulty. If unwanted interferences are not coordinated on the side of the source-thread or preventable on the side of the target-thread, then things could become extremely confusing.

So there should be mechanisms for furnishing defined entry-points in some other thread (`yield()`, `wait()/notify()`, `join()`, `interrupt()`) and mechanisms that prevent interference with code and objects (synchronized).

Concept: Atomicity and Exclusive Access (Locking)

Concept: Atomicity and Exclusive Access (Locking) — In this conceptual section examples from the Java language are used to effectively illustrate the content. Consider the source-code on page 217, there is shown how two Java-threads access one object:



Threads in general have access to a common share of data and code, in short: **THREADS SHARE MEMORY.** Means of access-control as they have been presented up to now for Java source-code were identifiers, qualifying identifiers, using access-control modifiers in declarations, casting. These means of access-control can be supplemented by a programming construct which controls thread-communication or -interference:

(DATA: Exclusive Access/Locking) Other threads can be prevented from accessing the classes or objects used by one thread by “locking” the respective resources.

(CODE: Atomicity) Another resource that becomes relevant in multithreading is processor-time. It may be useful to let entire pieces of source-code run exclusively on a processor, uninfluenceable by other threads.

atomic blocks
of code

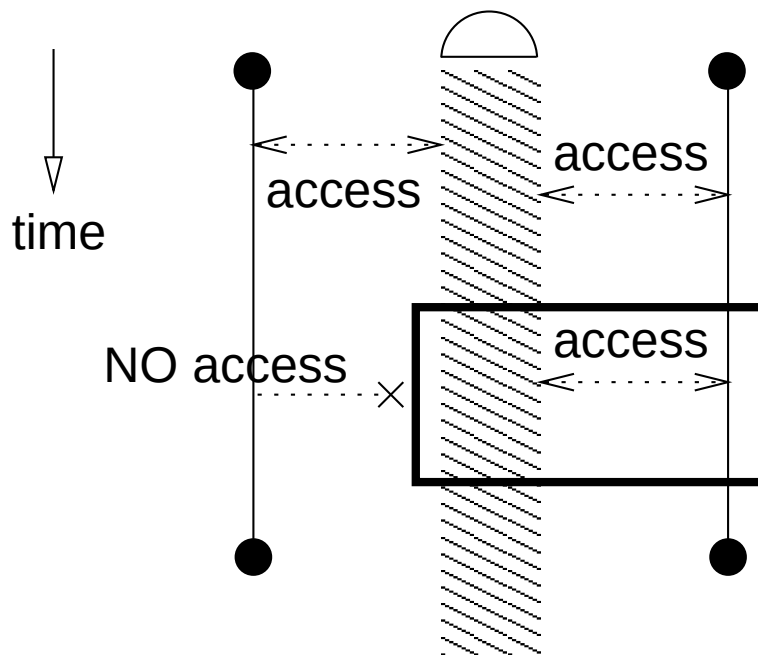
Thus, blocks of source-code which are run un-interrupted on one processor are called atomic. **Atomic blocks of code** possess the (virtual-)processor exclusively. Atomic blocks of code cannot be interrupted by any other kind of scheduling-mechanism or interruption-command. Making a block of code atomic guarantees exclusive right to scheduled processor-time until the code-block has finished. Within a one-processor computer this guarantees an uninfluenced result, within multiprocessor-environments this guarantees only the one resource, processor-time. Therefore, in view of the existence of multiprocessor computers or network-access, atomic language-constructs are usually associated ALSO with locking functionalities (treated some paragraphs later).

Atomic Operation: An atomic operation may consist of a sequence of commands that cannot be seen or are not accessible from a parallel operation. An atomic operation cannot be interrupted and may even be given exclusive access to data. Usually an atomic operation acts free from any influences of other parts of a program or the entire computer-system and its environment.

In Java, atomic pieces of code are given, roughly spoken, within code-blocks headed by the `synchronized-keyword`.

Multiple threads may try to access an object or the static members of a class at almost the same time. Some commands, like assignments or references or elementary arithmetic operations on those objects, remain uninfluenced by other threads; the different operations cannot overlap when triggered from different threads. Such operations on objects or classes are called atomic operations. In Java the elementary operations on primitive-types and reference-types are atomic (assignment, copying and cloning, comparison), they can be seen as being handled by the implementation of the virtual machine in one step. (For an exception see the outlook of the section on page 86.) Other operations, mostly those compounded by several other elementary operations usually become non-atomic.

The other aspect of making resources exclusively available to one thread was the accessibility of data. The graphic below shows the principle of binding exclusive access to an object to the code of a thread:



locking data

This kind of access-control in the context of multithreading is also called **locking data** or **locking objects/classes**.

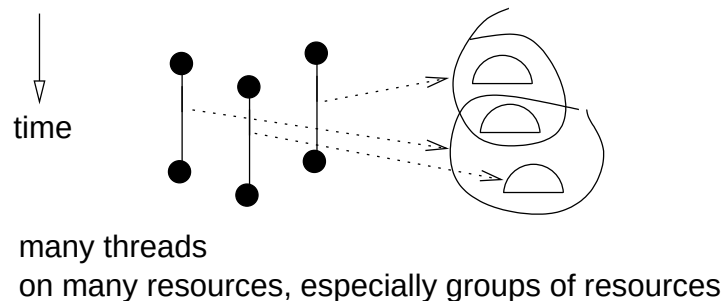
Locking Data: Locking an object or a class with respect to some code segment in a thread means that the thread or better its code segment is granted an exclusive access-right to manipulate the object or the class (for example reading or writing data). So when the code segment is worked over, no other thread has access to that object and has to wait until access is possible again. (Since scoping and access-control concern themselves only with the validity of identifiers, a mechanism that temporarily denies access to an object or a piece of code deserves a name of its own: Locking.)

In Java, static synchronized methods lock their enclosing class, non-static synchronized methods lock their object, identifiers or method-signatures cannot be locked!

Remember private instance-variables that are exclusively accessible from objects instantiated from the same class. In the context of multithreading, locked data has some aspects of temporarily declared private variables.

Mini-Outlook: Next to (1) access-control with modifiers like `public`, `protected` and `private`, by qualifying and casting, and (2) the temporary denial of access to all other threads by locking, there can be imagined a third case (3) of access-control, where the denial of access depends on the identity of the accessor and an explicitly given permission; this case is treated in the part of this text on security beginning with page 449.

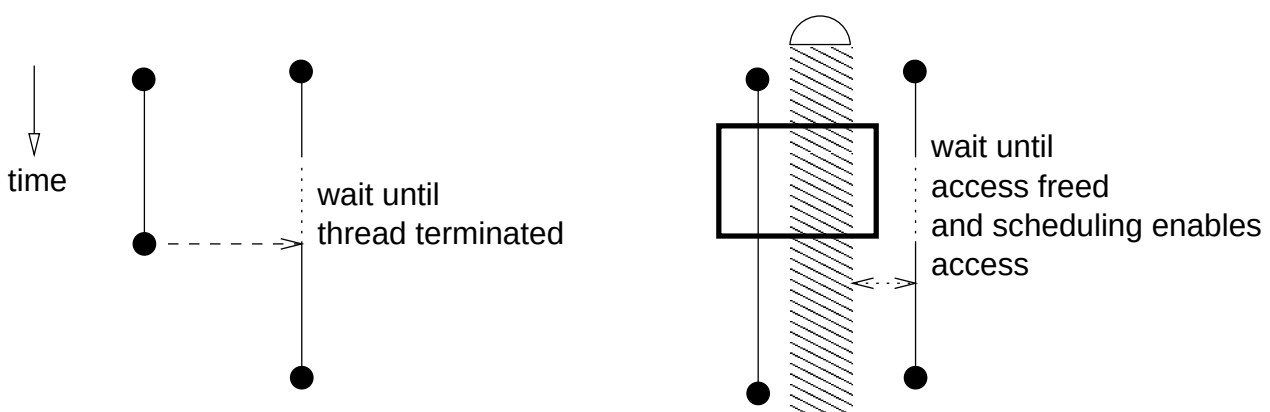
Many problems in practical programming with multiple threads, atomic code-blocks and exclusively accessible objects and classes arise from constellations like the one depicted in the graphic below:



Here several threads try to access several objects or groups of objects. The respective operations have to be coordinated by the application-programmer, this generally involves a careful choice of making code-blocks atomic and in conjunction to reserving — in proper sequence — exclusive access to the groups of objects. A more concrete kind of typical problem in that context is given by the “Dining Philosophers Problem” or rather “dining humans problem”: Five persons (threads), four forks (resource1) and three knives (resource2). How to make a fair time-dependent distribution at the dinner table? Fair means: Everyone gets his share of food at approximately the same time as the others do. Unfair means: Discriminating persons or even letting one or more persons starve. (This reflects a standard resource-distribution-problem of application-related programming with multiple threads.)

Concept: Waiting

Concept: Waiting — Problem-related organization of resource distribution, as presented above, naturally involves threads that have to wait until they can access resources:



The two basic cases are depicted in the graphic above: A thread waits until another thread terminates (joining) or a thread waits until another thread releases (wait/notify) its rights for exclusive access to resources.

OUTLOOK: Another form of waiting is realized by daemon-threads inside a runtime-environment: Exactly at the time the last non-demon-thread terminates, all daemon-threads are terminated automatically. Daemon-threads are used for supporting services, like listening to input-devices (keyboard, mouse, network-connection), and further for monitoring and signaling.

Java: Handling Multiple/Communicating Threads

Java: Handling Multiple/Communicating Threads — How exactly are realized the concepts of atomicity of code-blocks, locking resources, and waiting in the multithreading context of the Java language? This question should be answered by the Java language-construct `synchronized( exclusively accessible data/code ){ code becoming atomic }` an instance-method of the `java.lang.Thread`-class: `join()`, and instance-methods of the class `java.lang.Object`: `wait()/notify()`.

Java: Isolating Code and Data from other Threads

Java: Isolating Code and Data from other Threads — Remember that in Java the elementary operations on primitive-types and reference-types are atomic (assignment, copying and cloning, comparison), they can be seen as being handled by the implementation of the virtual machine in one step. There are two exceptions to that rule: With variables of the primitive-types `long` and `double`, elementary operations are non-atomic. Why? Because, due the implementation of the Java Virtual Machine⁹⁰ (JVM), the handling of variables of these types, needs several independent steps of virtual processor-commands. And these command-sequences have been given no protection in the Java Virtual Machine Specification. Another thread accessing a variable of type `long` and `double` at the same time thus can create problems; these can be avoided by shielding those operations:

```
synchronized( null ){ double x = 1.00000000000000001 };
```

This kind of shielding can be generalized for other pieces of code:

In Java, the source-code of a thread is either found in the thread's `run()`-method or in some method called from there. From those places, any code-segment gets exclusive access to a given object or class with the following Java-construct:

```
synchronized( objectOrClass ){ codeSegment }  
// locked object or class with associated atomic piece of code
```

Effects:

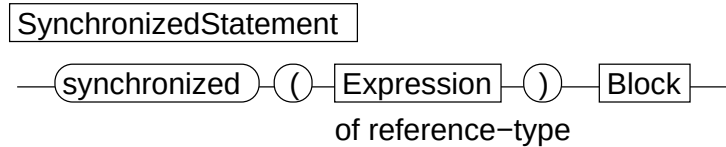
- No other thread can influence the code segment by letting it sleep or by interrupting it. The code is left unbroken by any scheduling-mechanism.
- When the code-segment is worked over, no other thread can access any members of object or class referenced by the argument.
- Variables of primitive-type, used within the code segment, are locked only if they're local with respect to the code segment.

Thus, objects or classes can be exclusively bound to some atomic piece of code in a given thread, this makes the `synchronized`-statement above a combination of atomic operation with a locking capability. If multiple objects or classes are to be locked at the same time, then they may be aggregated as introduced in the next chapter beginning with page 233.

Apart from the `synchronized`-statement above, the `synchronized`-keyword also can be used as a `synchronized`-modifier on a method. This indicates again that, once these blocks are run on the implementation of the Java Virtual Machine (JVM), these blocks have the top-priority AND will not respond to any scheduling-mechanism. That means, once those `synchronized` blocks or methods are run on the virtual processor, they will not share it with any other part of the code until they are finished. Ideally, that means the processor (or a processor on a multiprocessor machine) dedicates itself solely to the block of code in question; this is also called **machine-synchronization**.

Entering a `synchronized` block of Java source-code also means acquiring exclusive access to the object referenced by the argument. For the duration of the running of the block this object is "locked" to that code, or the code has acquired to lock of that object denying other code access to the object.

⁹⁰Java's virtual processor works with 32bit words at a time, variable of primitive-type `long` and `double` encompass 64bits or 8bytes. Thus, variables of that type need a two-step procedure to be transferred into the (virtual) Java-processor.



SynchronizedStatement := synchronized "(" *Expression* ")" *Block* — Expression 90, Block 105

The expression in parentheses needs to be of reference-type; the respective object remains inaccessible during the work on the block. A synchronized(this) term locks the enclosing object. Otherwise any other referred object or class is locked.

A statement of the kind synchronized(null){...} represents a purely atomic block, without any locking aspect. Again: The synchronized-statement comes to use if other threads of control may try to interrupt a block. And the synchronized-statement reserves a (virtual-)processor exclusively for this block, that means it **machine-synchronizes the block**. Below are presented examples of different usage of the Java-keyword synchronized:

```
synchronized( objectName ){ code }
synchronized( className ){ code }
```

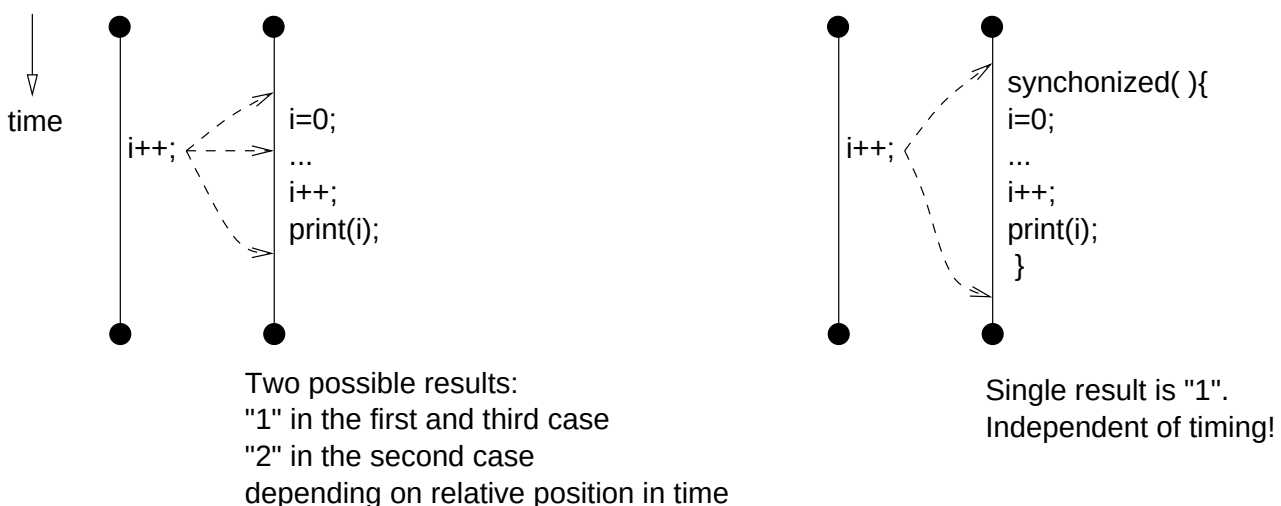
If a synchronized *method* has been called (by some thread), then, until the method returns, the whole referred object or class is shielded against intrusion from any other thread. So synchronizing instance-methods locks the enclosing object.

```
// the non-static case:
synchronized returnType methodName( parameterList ){ code }
// essentially the same as:
returnType methodName( parameterList ){ synchronized( this ){ code } }

//
// the static case:
static synchronized returnType methodName( parameterList ){ code }
// essentially the same as:
static returnType methodName( parameterList ){ synchronized( ClassName ){ code } }
```

CAUTION: Calling unsynchronized methods from synchronized environments annihilates the synchro- CAUTION!
nization effect!

In the graphic below has been indicated the effect of making a code segment atomic: The keyword synchronized protects the associated block of source-code from being influenced by the incrementation done inside the other thread.



The graphic above over-simplifies matters with respect to Java: In Java, on the level of the implemented Java Virtual Machine (JVM, the Java byte-code-interpreter), threads access the same address-space, but represent different processor-states (stacks). Since each local variable is part of the processor-state (stored on the thread's stack), in Java, threads may share field-variables and class-variables but THREADS CAN NEVER SHARE LOCAL VARIABLES. The following program essentially implements the phenomenon shown in the graphic above by using objects:

local variables
are private to
threads

```

/* First there is given an object of type VarInteger.
   This wraps an integer-value that can be
   changed with a set()- and retrieved with the get()-method.

   Then two threads are at work:
-- The main()-thread tries to decrement the value as much as possible.
-- A Service thread tries to increment the value four times EXCLUSIVELY.

   To deny access to the main-thread, the Service thread synchronizes the
   adding-code and the object of class-type VarInteger.
   The process of incrementing works exclusively only
   if the get()-method has been synchronized too.
*/
class VarInteger
{ int i; VarInteger( int x ){ i = x; };
  public void set( int i ){ this.i = i; }
  //public synchronized int get(){ return i; }
  // the line above keeps atomicity exclusive, SYNCHRONIZED or the line below:
  public int get(){ return i; } // breaks atomicity
}
/* Primitive-Type Object-Wrappers like Integer, Boolean, ...
   let their values remain fixed, the class above allows changing the value */

class Service extends Thread
{ VarInteger count; int millis;
  Service( int milliSeconds, VarInteger count )
  { this.count = count; millis = milliSeconds;
    this.start();
  }
  public void run()
  { synchronized( count )
    { for( int i = 0; i < 4; i++ )
      { count.set( count.get() + 1 );
        System.out.println( count.get() );
        try{ sleep( millis ); }catch( InterruptedException e ){ e.printStackTrace(); };
        // Thread.yield(); // sleep(0)=yield()
      }
    }
  }
}

public class Race
{ public static void main( String unused[] )
  {
    VarInteger counter = new VarInteger(0);
    Service adder = new Service( 100, counter );
    while( adder.isAlive() )
    { counter.set( counter.get() - 1 );
      Thread.yield(); // enforces thread change if no time-slicing done
    }
    System.out.println("adder not alive");
  }
}
// The main()-method outraces the adder-thread if the adder-thread sleeps too much.
// The 'set()' -method above should be synchronized too.
/*
   Methods called from a synchronized environment that are unsynchronized
   can produce race-conditions.
   Atomic Java constructs are either synchronized blocks or calls of synchronized methods.
   1. The synchronized-keyword and modifier act as pre-atomifiers:
      Enclosed method-calls have to call synchronized methods too!
   2. The synchronized-keyword also grants exclusive access to objects and classes.
*/
/*
(output with synchronized VarInteger.get()-method: SYNCHRONIZED)
(ideally the VarInteger.set()-method should be synchronized too):
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] marycosway% javac Race.java
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] marycosway% java Race
1
2
3
4
adder not alive
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] marycosway%

(output if VarInteger.get()-method is not synchronized)
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] marycosway% javac Race.java
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] marycosway% java Race
1
-34
-81
-133
adder not alive
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] marycosway%
*/
/*
(output with synchronized VarInteger.get()-method: SYNCHRONIZED)
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>javac Race.java
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>java Race
9

```

```

1
2
3
adder not alive

C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>

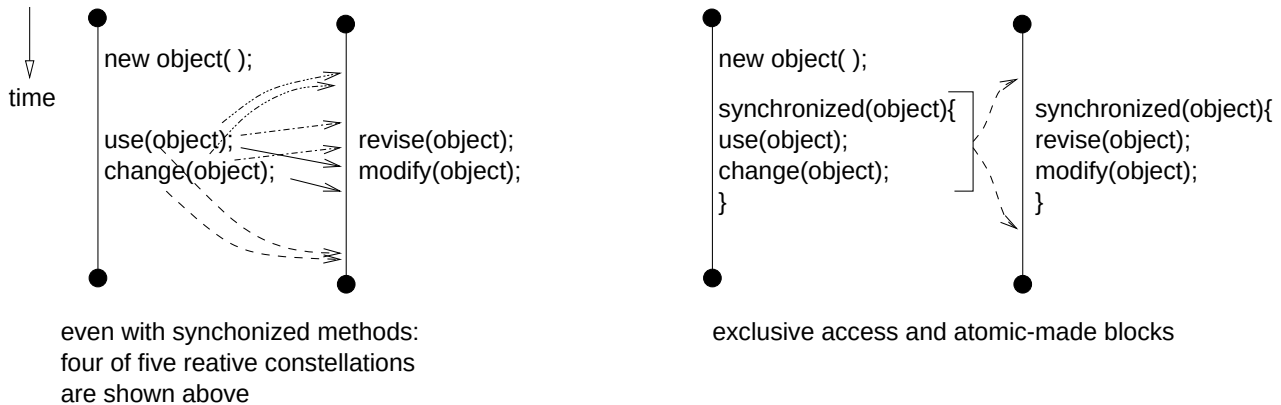
(output if VarInteger.get()-method is not synchronized)
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>javac Race.java

C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>java Race
0
-12418
-27344
-46192
adder not alive

C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>
*/

```

The graphic below demonstrates that the synchronized-keyword can be used to assure proper **RELATIVE sequencing of source-code** when accessing an object or class from different threads:



(**Absolute sequencing**⁹¹ of source-code, when accessed from different threads, becomes possible by using a wait() and notify() pair or the join()-method, as introduced beginning with page 220.) A practical implementation of the constellation depicted in the graphic above is given by the program below:

```

/*
The central object is given by an instance of class-type java.util.Vector
This object is accessed from the run()-method of a Thread-object "Together"
adding two strings to the vector, both either synchronized or unsynchronized.
But the Vector-object also is accessed from a run()-method of a Thread-object "Disturb",
adding a string of "plus"-signs.
Randomizing sleeping-times and looping ten times, to reproduce different settings,
gives the output as printed below.
*/

import java.util.* ; // contains the predefined Vector class
class Together extends Thread
{
    java.util.Vector object;
    Together( Vector arg ){ object = arg; this.start(); } // constructor
    void to( Vector arg ){ arg.addElement( "to" ); }
    // java.util.Vector.addElement() already has been synchronized!
    void gether( Vector arg ){ arg.addElement( "gether" ); }
    //
    // Now two alternative run()-methods: One with UNSYNCHRONIZED access:
    /* public void run()
    { sleeps( 30 ); to( object ); sleeps( 30 ); gether( object ); }
    */
    // OR the other run()-method below with SYNCHRONIZED access:
    public void run()
    { sleeps( 30 );
      synchronized( object ){ to( object ); sleeps( 30 ); gether( object ); }
    }
    //
    void sleeps(int s)
    { try{ this.sleep( (int)( s*java.lang.Math.random() ) ); }
      catch( InterruptedException e ){ e.printStackTrace(); }
    }
}
class Disturb extends Thread

```

⁹¹In some cases, the necessity of absolute sequencing could reveal the threaded problem as being not truly parallelizable at that special point.

```

{
    java.util.Vector object;
    Disturb( Vector arg ){ object = arg; this.start(); }
    void enterPluses( Vector arg ){ arg.addElement( "+++" ); }
    public void run(){ sleeps( 1 ); enterPluses( object ); }
    void sleeps( int s )
    { try{ this.sleep( (int)( s*java.lang.Math.random() ) );}
      catch( InterruptedException e ){ e.printStackTrace(); }
    }
}
public class Synchrobject
{ public static void main( String unused[] )
  {
    for( int i = 0; i < 10; i++ )
    { Vector object = new Vector();
      Together thread1 = new Together( object );
      Disturb thread2 = new Disturb( object );
      try{ thread1.join(); thread2.join(); }
      catch( InterruptedException e ){ e.printStackTrace(); }
      System.out.println( object.toString() ); // OUTPUT
    }
  }
}

/* output: SYNCHRONIZED
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > javac Synchrobject.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > java Synchrobject
[+++, to, gether]
[to, gether, +++]
[to, gether, +++]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[to, gether, +++]
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading >

output: UNSYNCHRONIZED
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > java Synchrobject
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > java Synchrobject
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[+++, to, gether]
[to, +++, gether]
[+++, to, gether]
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading >
*/

```

ultra-atomic
computing

The most extreme case to assure atomicity in Java would be to declare all non-method-members private and all non-private methods synchronized.

Some practical ideas how the synchronized-keyword may be used are given below:

```

synchronized(Account)
{showBalance; manageDrawingProcess; balanceAccount; showBalance}
/* Otherwise another person, that has access to the account,
   could interfere by drawing money from it and producing thereby confusion.
*/

synchronized void useLoudSpeaker(){...};
/* Using one loudspeaker as an output-device for multiple channels of sound
   at the same time, would generates a language- or music mix.
*/

```

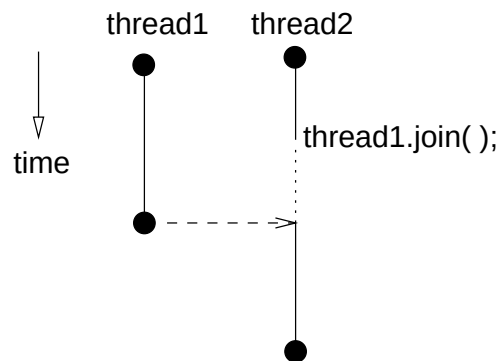
Remember that in Java, the change of local variables generally doesn't need to be synchronized because local variables, stored in the thread's stack cannot be changed by other threads. Exceptions are variables of primitive-type long and double as mentioned on page 216.

Java: Waiting for other Threads

Java: Waiting for other Threads — Making code-segments atomic and locking objects or classes for exclusive access by the current thread gives the programmer control over access from other threads. As can be seen above, in the graphics, there still remains some ambiguity, the parallel operation can come into effect before or after the synchronized code-segment (relative- or absolute code-sequencing). If the programming-problem requires to resolve that ambiguity⁹², then the following mechanism can be used to relate the threads:

⁹²Multithreaded programming requires parallelizable problems, the relating constructs presented here should be identified already before the design of the program. Too many relations of this kind between threads may question the parallelizability of their code.

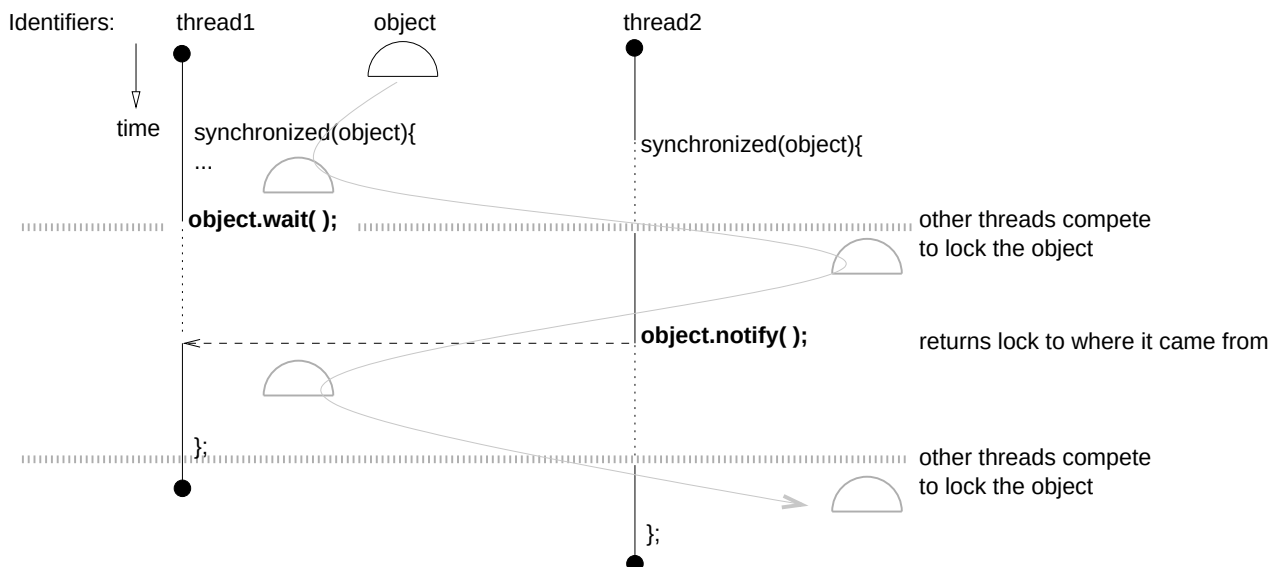
In the first case, one thread waits until the other thread has finished before proceeding: In Java, the CALLER-THREAD WAITS UNTIL THE QUALIFYING-THREAD HAS FINISHED. If the joining-thread already stopped, then the statement is ignored.



qualify.join() caller-thread stops, until qualifying thread finishes

A number of primitive-type long may become an argument of the overloaded instance-method join() of the Thread-class: This argument specifies the number of milliseconds of maximal waiting-time; if the qualifying thread hasn't finished by then, the waiting thread resumes its work without further delay.

In the second case, using a wait()-call, a thread “waits” until the thread is “notified” to proceed: In Java, a thread locks an object within a synchronized-statement; then, from within the synchronized block, calling the wait()-method of the locked object releases the lock of the object and renders the thread blocked, now another thread may access the object. A call of the notify()-method of the object makes it possible to reschedule the initial thread for re-acquiring the lock on the object.



The following piece of source-code demonstrates such a waiting- and notifying-process:

```

/* The object of type VarInteger stores a number.
   A thread "Decimals" increases that number by 10 five times.
   The other thread "Units", increases that number by 1 three times.
   and then resets it.
   The thread "Decimals" should wait until the other thread ("Units")
   has reset the number.
*/
class VarInteger
{ int i; VarInteger( int x ){ i = x; };
  public void set( int i ){ this.i = i; } public int get(){ return i; }
  public void increment(){ ++i; } public void decrement(){ --i; }
}

class Units extends Thread
{ VarInteger object;
  Units( VarInteger arg )

```

```

    { object = arg;
      this.start();
    }
    public void run()
    { for( int d = 0; d < 4; d++ ) /*change the 4 loop to 7 and you have a deadlock*/
      { synchronized( object )
        { for( int i = 0; i < 3; i++ )
          { object.increment();
            System.out.println( "thread units: " + object.i );
          }
          object.i = object.i - 3; // reset unit count from 3->0, 13 -> 10, ...
          try{ object.wait(); }
          catch( InterruptedException e ){ e.printStackTrace(); } // NEW DECIMAL
        }
      }
    }
}

class Decimals extends Thread
{ VarInteger object;
  Decimals( VarInteger arg )
  { object = arg;
    this.start();
  }
  public void run()
  { for(int k = 0; k < 5; k++ )
    { synchronized( object )
      { object.i = object.i + 10; // decimal count
        System.out.println("thread decimals :"+ object.i );
        object.notify();
      }
      //try{sleep(3000);}catch(InterruptedException e){ e.printStackTrace(); }
    }
  }

  public class Waiting
  { public static void main( String unused[] )
    { VarInteger object = new VarInteger( 0 );
      Units      ones   = new Units( object );
      Decimals   decims = new Decimals( object );
    }
  }
}

/*
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% javac Waiting.java
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% java Waiting
thread units: 1
thread units: 2
thread units: 3
thread decimals :10
thread units: 11
thread units: 12
thread units: 13
thread decimals :20
thread units: 21
thread units: 22
thread units: 23
thread decimals :30
thread units: 31
thread units: 32
thread units: 33
thread decimals :40
thread decimals :50
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro%
*/

/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > javac Waiting.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > java Waiting
thread units: 1
thread units: 2
thread units: 3
thread decimals :10
thread units: 11
thread units: 12
thread units: 13
thread decimals :20
thread units: 21
thread units: 22
thread units: 23
thread decimals :30
thread units: 31
thread units: 32
thread units: 33
thread decimals :40
thread decimals :50
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading >
*/

/*
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading> javac Waiting.java

C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading> java Waiting
thread units: 1
thread units: 2
thread units: 3
thread decimals :10
thread decimals :20
thread decimals :30
thread decimals :40
thread units: 41
thread units: 42
thread units: 43
thread decimals :50
thread units: 51
thread units: 52
thread units: 53
(Ctrl-C)

```

C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading>
*/

The methods `wait()`, `notify()` and `notifyAll()` are instance-methods of the `java.lang.Object`-class (compare also page 164). So every object has these (and other) methods as default instance-methods. When a `synchronized`-statement locks the object with respect to some code segment, calling of the `wait()`-method from that segment frees the object and blocks the thread. Other threads may now compete for that lock and one of them locks the object with respect to one of its code segments (see the upper dotted gray line in the graphic above). — Within such a segment, a statement `object.notify()` advises the Java Runtime Environment to select one of the waiting threads and let this one resume to run, locking the object again. Instead, the `notifyAll()`-method wakes them all up for continuous competition for that lock. (other dotted gray lines would have to be inserted in the graphic above, one when calling this method others when finishing `synchronized` code-blocks.)

Observe that the `wait()`-method blocks the thread from which it was called and releases locked objects, whereas the static method `sleep()` of the `java.lang.Thread`-class does not release any locked objects! In pseudo-code:

`sleep()`
holds rights —
`wait()` gives
rights away

```
Thread.sleep( long millisec ) // can be seen identical to:
synchronized( null ){ this.wait( long millisec ) }
```

- **public final native void `wait()`** used in code segments locked to its qualifying object by a `synchronized`-statement. Blocks its thread and lets other interested threads compete for locking that object. An argument of primitive-type `long` can let this overloaded method assure that its calling thread resumes competing for the object's lock after the indicated time-period, whether or not it has been notified.
- **public final native void `notify()`** chooses to return to an arbitrary (depends on the implementation of the Java Virtual Machine (JVM)) waiting thread; not necessarily the one that called. The `notify()`-method is used in code segments locked to its qualifying object. The method releases the Block of a thread that called the object's `wait()`-method.
- **public final native void `notifyAll()`** activates the thread-scheduling mechanism for all waiting threads to return to one of those. Like its counterpart above, but releases the Blocks of ALL threads that called the object's `wait()`-method.

No qualifier preceding the method-call means `this.wait()` or `this.notify()`.

Constructs like `while( amount() == 0 ) wait();` test whether the notification(s) furnish the right conditions to proceed.

Java: Daemon-Threads And Shutting Down of JRE

Java: Daemon-Threads And Shutting Down of JRE — As demonstrated by the graphic on page 205, threads can outlive their calling thread without being terminated prematurely. So-called daemon-threads differ in that respect from regular threads: Daemon-threads cannot hinder the Java Runtime Environment (JRE) from closing down. In the moment the last regular thread has been terminated, which is the moment the Java Runtime Environment is populated only by daemon-threads — the Java Runtime Environment shuts down.

Daemon-threads usually are spawned at the same time the Java Runtime Environment starts⁹³, they may run in some infinite loop, and close down together with the Java Runtime Environment. Daemon-threads usually run with a low priority, offering services to regular threads (of higher priority). System-side examples of daemon-threads are:

(GC) Garbage-collection-thread of the Java-interpreter which can be explicitly made Running by the programmer with the call `System.gc()`. (See also page 444.)

(EDT) A thread that processes mouse- and keyboard-events, the event-dispatch-thread, introduced on page 586.

⁹³Triggered by starting an application or an applet, also see page 431.

If there run only daemon-threads on the Java Virtual Machine (JVM, the byte-code interpreter), then the Java Runtime Environment will terminate its work, together with all remaining daemon-threads.

A Java programmer can produce auxiliary daemon-threads by invoking the Thread-object's `setDaemon()`-method, thus any Java-thread can be tagged as a daemon-thread.

boolean **isDaemon()** tests a thread to be a daemon-thread.

void **setDaemon(boolean)** sets the daemon-attribute to true or false.

Problems specifically arising with Thread-Communication:

Problems specifically arising with Thread-Communication: — The (quasi-) parallel running of multiple threads, their mutual access to data and their communication can cause new types of problems in programming:

Dangers: Starvation

Dangers: Starvation — The danger of starvation becomes relevant when there are fewer processors than threads, which usually is the case. The problem arises when higher priority threads become or are processor-intensive and there are no alternate processors available. Then, lower priority threads receive no processor-time, they are worked over slowly and thus cannot finish within an acceptable time-period, they starve. The user wonders why the computer reacts sluggish or seems to act autistically. A general solution may be given by letting potentially processor-intensive threads have a lower priority, this benefits one-processor platforms and doesn't hurt multiprocessor platforms.

Real-life example: A bully in a kindergarten who monopolizes the wooden toy train.

```

/*
    Two Thread-objects of type ThreadWithPriority are created.
    They are created with differing priorities.
    The thread which finishes first terminates the run of both.
*/
// the threads communicate using an instance of this class
final class Sign{ public boolean sign; }

class ThreadWithPriority extends Thread
{
    int p; Sign halt;
    ThreadWithPriority( int p, Sign halt )
    { super(); this.p = p; this.setPriority(p); this.halt = halt; this.start(); }
    public void run()
    { for( int i = 1; i <= 10; i++ )
        {
            if( halt.sign == true )
            { System.out.println( "total halt by thread of priority " + this.getPriority() );
              System.exit( 0 );
            }
            System.out.println( i + " Thread with priority " + this.getPriority() );
            Thread.yield(); // yielding in Mac OS
            //this.interrupt(); // works like yield() in Mac OS
            /*try{ Thread.sleep( 10 ); }
               catch( InterruptedException e ){ System.out.println("interrupted"); }*/
            // sleep does not respect priorities and has a time-slicing effect
        }
        halt.sign = true;
        System.out.println( "Thread with priority " + this.getPriority() + " ended." );
        // System.exit( 0 ); // termination without communication
    } }

public class Starve
{
    public static void main( String delayInMilliSec[] )
    {
        Thread.currentThread().setPriority( Thread.MAX_PRIORITY );
        System.out.println
            ( "Priority of main()-thread is " + Thread.currentThread().getPriority() );
        Sign halt = new Sign(); // same result as being a static member of Starve
        System.out.println( "Thread with low priority starts: " );
        new ThreadWithPriority( Thread.MIN_PRIORITY, halt );
        try{ Thread.sleep( Integer.parseInt( delayInMilliSec[0] ) ); }
        catch( Exception e ){ e.printStackTrace(); };
    }
}

```

```

        System.out.println( "Thread with higher priority starts: ");
        new ThreadWithPriority( Thread.NORM_PRIORITY-1, halt ); // the bully
    } }
}
/*
(may depend on platform and chosen stream(java Starve or java Starve>>>Starve.java))
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro% javac Starve.java
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro% java Starve 50
Priority of main()-thread is 10
Thread with low priority starts:
1 Thread with priority 1
2 Thread with priority 1
3 Thread with priority 1
4 Thread with priority 1
5 Thread with priority 1
6 Thread with priority 1
7 Thread with priority 1
8 Thread with priority 1
9 Thread with priority 1
10 Thread with priority 1
Thread with priority 1 ended.
total halt by thread of priority 1
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro%
*/

/*
andreadipietro@linux:~/JavaCodeFiles/Basic1/SCMultithreading > javac Starve.java
andreadipietro@linux:~/JavaCodeFiles/Basic1/SCMultithreading > java Starve 50
Priority of main()-thread is 10
Thread with low priority starts:
1 Thread with priority 1
2 Thread with priority 1
3 Thread with priority 1
4 Thread with priority 1
5 Thread with priority 1
6 Thread with priority 1
7 Thread with priority 1
8 Thread with priority 1
9 Thread with priority 1
10 Thread with priority 1
Thread with priority 1 ended.
Thread with higher priority starts:
total halt by thread of priority 4
andreadipietro@linux:~/JavaCodeFiles/Basic1/SCMultithreading >
*/

/*
C:\WINDOWS\JavaCodeFiles\Basic1\SCMultithreading>javac Starve.java

C:\WINDOWS\JavaCodeFiles\Basic1\SCMultithreading>java Starve 25
Priority of main()-thread is 10
Thread with low priority starts:
1 Thread with priority 1
2 Thread with priority 1
3 Thread with priority 1
4 Thread with priority 1
5 Thread with priority 1
6 Thread with priority 1
7 Thread with priority 1
8 Thread with priority 1
9 Thread with priority 1
10 Thread with priority 1
Thread with priority 1 ended.
Thread with higher priority starts:
1 Thread with priority 4
2 Thread with priority 4
3 Thread with priority 4
4 Thread with priority 4
5 Thread with priority 4
6 Thread with priority 4
7 Thread with priority 4
8 Thread with priority 4
9 Thread with priority 4
10 Thread with priority 4
Thread with priority 4 ended.
total halt by thread of priority 4
C:\WINDOWS\JavaCodeFiles\Basic1\SCMultithreading>
*/

```

How can information be shared globally between threads? Primitive-types as arguments are transferred by-value and therefore variables of such type change only locally (see the variable priority p, in the code above) therefore use reference-types (see the object of type Sign above).

The third output right above shows that time-slicing also may happen across different priorities! Intending to avoid starvation, time-slicing may distribute processor-time weighted according to the priority of the thread.

Dangers: Race-Conditions

Dangers: Race-Conditions — “Race-condition” is the name for those thread-interactions that cause problems with non-appropriate timing. For example, two threads may write contradicting data into one object, by which latter may confuse both threads. For a programmed example of a race-condition see the demonstration of the synchronized-statement on page 217. But using synchronized-statements unskillfully, still can cause race-conditions. Generally, results or operations should be independent of the order of thread-activation. Thus, a race-condition is said to occur if the application is influenced by the order of activating the threads. Race-conditions are generally given by unforeseen cases that influence the threads in unwanted ways. Severe race-conditions introduce particularly pathologic communication problems often together with termination difficulties.

A real-life example maybe given by a Laurel-and-Hardy action: The Laurel-thread puts a hammer at his workplace and leaves to get the nails. The Hardy-thread takes the hammer away. The Laurel-thread arrives with the nails and wonders about the vanished hammer.

Dangers: Deadlock

Dangers: Deadlock — Deadlock is a form of a race-condition which causes the program to prematurely terminate, due to mutual blocking threads:

```
class TheThread extends Thread
{ String name;
  TheThread( String s ){ super(); name = s; this.start(); }
  public void run()
  { System.out.println( "Thread " + name + " started." );
    synchronized(this)
    { try{ wait(); }catch(Exception e){ e.printStackTrace(); };
      notify();
      System.out.println( "Thread " + name + " ended" );
    } } }

public class MiniDeadlock
{ public static void main( String unused[] )
  { new TheThread( "One" ); new TheThread( "Two" );
    // above: TWO threads, each one waiting for the other.
    System.out.println( "main()-method-call ended" );
  } }
/* None of the threads stops waiting,
   because both are waiting to be notified by the other thread.
   This inactivates the terminal from where the code has been run.
*/
/*
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro% javac MiniDeadlock.java
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro% java MiniDeadlock
Thread One started.
Thread Two started.
main()-method-call ended
^C (Ctrl-C)
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro%
*/

/*
andreadipietro@linux:~/JavaCodeFiles/Basic1/SCMultithreading > javac MiniDeadlock.java
andreadipietro@linux:~/JavaCodeFiles/Basic1/SCMultithreading > java MiniDeadlock
Thread One started.
main()-method-call ended
Thread Two started.
(Ctrl-C)
andreadipietro@linux:~/JavaCodeFiles/Basic1/SCMultithreading >
*/
```

A deadlock is a terminal race-condition. Two threads wait for signals from each other that enable them to signal each other. Or more explicit: The first thread waits for the signal of the second that enables the first thread to send the signal to the second thread. But exactly that signal is the one the second waits for before sending the very first. (The unsuspecting reader may be left fully confused by these sentences.)

Real-life example: The infamous argument about getting out of some place only to be accomplished by proving to be mentally disturbed. But the wish to leave that place being a clear sign of mental sanity.

Dangers: Lifelock

Dangers: Lifelock — A lifelock produces an eternal ping-ponging of two or more threads, which may prove as destructive as a deadlock. Real-life example: In an auction betters can find themselves in a lifelock of increasing their bids.

```
/* Two objects of type ThreadLife are acquainted.
   One finishes only if the other has finished.
*/
class ThreadLife extends Thread
{ Thread t; int n;
  ThreadLife( int n ){ this.n = n; }
  public void acquaintWith( Thread t ){ this.t = t; };
  public void run()
  { try{ Thread.sleep( 5 ); }catch( Exception e ){ e.printStackTrace(); };
    while( true )
    { System.out.println( "thread " + n + " alive, waiting for the other to finish");
      //try{ t.join(); }catch(InterruptedException e){} // makes another deadlock
      if( !t.isAlive() ){ System.exit( 0 ); } // this makes the lifelock
      yield(); // works well with Mac OS, Linux; unnecessary with Windows!
    } } }

public class MiniLifelock
{ public static void main( String unused[] )
  { ThreadLife a,b;
    a = new ThreadLife(1); b = new ThreadLife(2);
    a.acquaintWith(b); b.acquaintWith(a);
  } }
```

```

        System.out.println( "before start " + a.isAlive() + " " + b.isAlive() );
        a.start(); b.start();
        System.out.println( "main()-method-call ended" );
    } }
/* None of the threads stops running because both are waiting to be notified.
   Thus the terminal from where the code has been run, continuously prints messages.
*/

/*
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% javac MiniLifelock.java
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% java MiniLifelock
before start false false
main()-method-call ended
thread 1 alive, waiting for the other to finish
thread 2 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 2 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 2 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
....
^C (Ctrl-C)
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro%
*/

/*
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > javac MiniLifelock.java
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading > java MiniLifelock
before start false false
main()-method-call ended
thread 1 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
...
thread 2 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 2 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 2 alive, waiting for the other to finish
...
(Ctrl-C)
andreadipietro@linux:~/JavaCodeFiles/Basics1/SCMultithreading >
*/

/*
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading > javac MiniLifelock.java

C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading > java MiniLifelock
before start false false
main()-method-call ended
thread 2 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 2 alive, waiting for the other to finish
thread 1 alive, waiting for the other to finish
thread 2 alive, waiting for the other to finish
....
(Ctrl-C)
C:\WINDOWS\JavaCodeFiles\Basics1\SCMultithreading >
*/

```

Concept: Pragmatics in Thread-Specific Programming

Concept: Pragmatics in Thread-Specific Programming — Do structure source-code by using threads only if the threads can be made independent of each other or if the threads interact only minimally. Try to avoid synchronization-constructs or use them as little as possible. Performance suggests to implement as few synchronized blocks as possible, security suggests as many as necessary. Do strive to test multithreaded configurations as prototypes. If possible, try hard to write the source-code to be easily re-assessable! Methods, which only change their local variables, need not lock the object of which they are a member, they need not to be synchronized. Examples: `Math.sqrt()` and the `length()`-method of the `String`-class or constructors that modify only the members of their object. But consider carefully those synchronized methods that are calling each other and let them lock objects in the same order, to prevent deadlocks. Reusing the same hierarchy of synchronized-blocks often helps.

Atomify a piece of code with exclusive access to an object with the `synchronized(object)`-statement only when needed to, and do this rather in smaller pieces than in a single large one. The statement `while(thread.isAlive()){try{Thread.sleep(100);}catch(InterruptedException e){}}` assures that the thread in question is really Nonexistent. As an alternative approach, a Java programmer may call the static method `Thread.currentThread()` to see whether the thread exists.

Usually, solving performance problems proves easier than finding thread-conflicts; therefore organizing and structuring threads usually precedes optimization. Constructs with little sense are `this.isAlive()` which is always true, and `this.join()` which never returns.

Aspire to change byte-code interpreter (implemented Java Virtual Machine) and the underlying scheduling-environment during development, to test the robustness of the threaded code! Especially for producing multithreaded source-code, use print-outs or a debugger.

How can Threads interact elementary on a Level above Scheduling?

Monopolizing their access to objects. Within the code segments that are “synchronized ” to the object.

Freeing their exclusive access. Calling the object’s `wait()`-method. Or by leaving a synchronized code-segment.

Threads can only compete to lock an object. By calling a synchronized method of an object or a class, the respective object or class has been locked by the caller until the method returns.

Creating another thread. Using the `java.lang.Thread`-class, the `Runnable`-interface and the `start()`-method.

De-/Reactivating thread Calling an object’s `notify()` or `notifyAll()`-method to unblock the thread that called the object’s `wait()`-method.

Waiting for another thread to finish. By calling the other thread’s `join()`-method, that returns when the other thread has stopped.

There are also ways to use some of these methods on entire groups of threads. This functionality is not presented here. Thread-groups are used by the Java Runtime Environment, for example, when organizing applet runs. Or the `System`-thread-group contains the main-thread, the thread in which the `main()`-method is run, the garbage-collection-thread, the `finalize`-thread (runs the `finalize()`-method) and other threads. The Abstract Window Toolkit (AWT) uses a thread-group containing the event-dispatch-thread and other GUI⁹⁴-related threads.

Though the following list is by far not complete, it gives another impression of the complexities to expect when trying to produce safely working multithreaded Java source-code. Multithreaded access-protection can be built in by following a few rules:

- When requiring consistent retrieval and setting of values, use synchronized methods for access. Within the `java.util` package, classes like `Vector`, `Stack` and `Hashtable` have been designed to use synchronized methods.
- Implement event-listening methods as synchronized, so even multiple calls of one handler are worked over successively.
- Do not machine-synchronize the calling of listener-methods, to avoid a deadlock if the listener makes a callback.
- Fire an event-object (or rather its references) using a synchronizably cloned vector of targeted event-listeners, so no listener unregistering from a different thread can interfere.

A Multithreaded Application

A Multithreaded Application — The part of the source-code below which has been called “producer” produces and stores `Integer`-objects in a buffer of the class `java.util.Vector`. (This `Vector`-class is part of the `java.util`-package, a pre-written code library, that is part of the Java Runtime Environment.) The “consumer” fetches the `Integer`-object by calling “`getMessage()`”. The two threads decouple the Producer- and Consumer-objects which solely communicate by a “`getMessage()`”-call. This presents a kind of source-message-drain setting with features (Pull-model, client-server) considered later around page 274.

```
/* An object of type Producer spawns a thread,
   that contains a non-terminating loop (while(true)).
   With each passage of the loop a counter is incremented (count++) and
   the value is pushed onto a buffer (messages) of type Vector.
   The relevant public method of this class is given by getMessage()
   which returns the counter-value.
*/
import java.util.Vector;

class Producer extends Thread // ----- Producer
```

⁹⁴GUI is an acronym for Graphical User Interface


```

{ static final int MAXQUEUE = 5; int count = 0;
  private Vector messages = new Vector(); // ..... the buffer
  private String s, empty="EMPTY";
  public void run()
  { while( true )
    { synchronized( messages )
      { if( messages.isEmpty() ) s = empty;
        else s = ( new Integer( messages.size() ) ).toString();
        System.out.print( ">>>" + s + ">>>" );
      }
      generateMessage( count++ );
      try{ sleep( (int)( 1000*java.lang.Math.random() ) ); }
      catch( InterruptedException e ){ e.printStackTrace(); }
    } }

  private synchronized void generateMessage( int count )
  { try
    { if( messages.size() == MAXQUEUE ) // buffer full
      { System.out.print( "producer reports: buffer full, waiting..." );
        wait(); // until consumer read from buffer (**)
      }
    }catch( InterruptedException e){ e.printStackTrace(); }
    synchronized( messages )
    { messages.addElement( new Integer(count) ); // counting detects lack in transfer
      System.out.print( "[" + count + "> " );
    }
    notify(); // Informs other waiting thread to proceed; a yield to waiting threads (*).
  } // If outcommented, the case of an undisturbed writing in the buffer will stop.

  public synchronized Integer getMessage() // ..... called by consumer
  { try
    { if( messages.size() == 0 )
      { wait(); // until notified that buffer not empty (*)
        System.out.print( " consumer reports: buffer empty!" );
      }
    }catch( InterruptedException e){ e.printStackTrace(); }
    Integer message = (Integer)messages.firstElement();
    messages.removeElement( message );
    notify(); // in case that buffer had been full (**),
    // if outcommented blocks on buffer full.
    return message;
  } }

/* --- The Consumer class has a main()-method which instantiates a Consumer-object,
    being also of type Thread.
    --- The run()-method contains a non-terminating loop in which the
        getMessage()-method of a Producer-object is called irregularly.
*/
class Consumer extends Thread // ----- Consumer
{ Producer producer;
  Consumer( Producer p ){producer = p;}

  public void run()
  { while( true )
    { Integer message = producer.getMessage();
      System.out.println( " >" + message.toString() + "]" );
      try{ sleep( (int)( 1000*java.lang.Math.random() ) ); }
      catch( InterruptedException e ){ e.printStackTrace(); }
    } }

  public static void main( String unused[] )
  { Producer producer = new Producer();
    producer.start();
    new Consumer( producer ).start();
  } }
/*
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% javac Consumer.java
[localhost:~/JavaCodeFiles/Basics1/SCMultithreading] andreadipietro% java Consumer
>>>EMPTY>>> [0> >0]
>>>EMPTY>>> [1> consumer reports: buffer empty! >1]
>>>EMPTY>>> [2> consumer reports: buffer empty! >2]
>>>EMPTY>>> [3> >3]
>>>EMPTY>>> [4> consumer reports: buffer empty! >4]
>>>EMPTY>>> [5> >>>1>>> [6> >>>2>>> [7> >>>3>>> [8> >5]
>>>3>>> [9> >>>4>>> [10> >6]
>7]
>8]
...
>>>2>>> [96> >>>3>>> [97> >94]
>>>3>>> [98> >>>4>>> [99> >>>5>>>producer reports: buffer full, waiting... >95]
[100> >>>5>>>producer reports: buffer full, waiting... >96]
[101> >97]
>>>4>>> [102> >98]
>>>4>>> [103> >99]
>100]

```

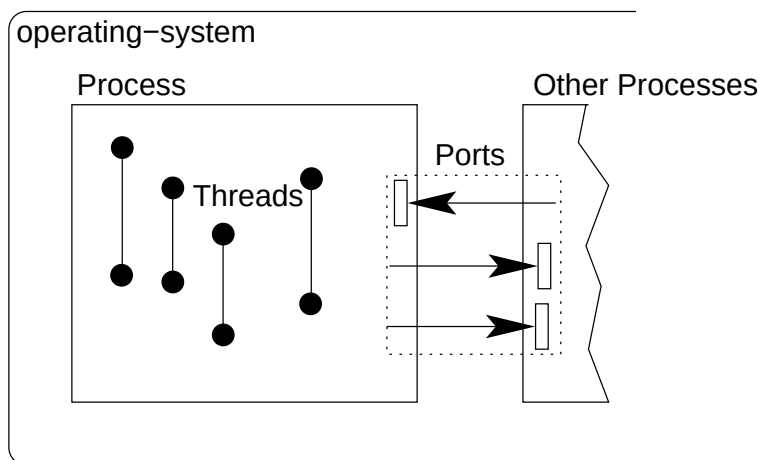
```
[localhost:~/JavaCodeFiles/Basic1/SCMultithreading] andreadipietro%
^/
```

On a single-processor computer, a drain-thread with `NORM_PRIORITY + 1` would readily deplete the FIFO row of the buffer. A drain-thread with `NORM_PRIORITY - 1` would let the source-thread fill the FIFO row of the buffer before emptying it. A drain-thread with `NORM_PRIORITY` lets the scheduling-mechanism of the Java byte-code interpreter take over: Without time-slicing the source-thread fills the buffer first, then the drain-reads. With time-slicing or on a multiprocessor computer — supported by the Java Runtime Environment —, the source- and drain-threads ideally should take turns in filling the buffer. The buffer above has been implemented by an object of type `java.util.Vector`; but the buffer also could have been implemented by a piped stream, as introduced on page 292 and around page 300.

Concept: Multiprocessing and Multithreading

Concept: Multiprocessing and Multithreading — Remember the concepts of concurrency, of threads and thread-scheduling. Within that context recall that different threads (also called lightweight processes) use the same memory-space in the computer's Random Access Memory. In other words: All threads within a Java Runtime Environment (JRE) have access to all objects or classes within that JRE, unless some other access-protection mechanism takes over (access-modifiers, access-restrictions given by a security-policy). So, different Java threads in a Java Runtime Environment share most of the computer's state.

A **process** is a similar entity of activity, but a process possesses its own private RAM-space. The scheduling-mechanisms remain the same, but a process-context-switch may use more than thousand machine-instructions (=processor-commands), whereas a thread-context-switch may require less than hundred machine-instructions. Usually a process allows to run several threads that share its RAM-space:



Threads can interact by accessing objects or data within their common memory (the Java Runtime Environment). But how can processes interact? Inter Process Communication (IPC) can be realized by so-called **shared-memory**. Specially assigned shared RAM-areas, so-called memory-sectors, together with rules for its usage, are called **ports**. Processes interact by writing data (= messages) to ports or reading data from ports:

- Ports store messages (data) like a buffer, such a buffer is also called message-queue.
- A port stores either output- or input-data, therefore single ports communicate unidirectionally.
- Ports, unlike buffers, often can restrict or allow the reading or writing of data.

The messages which ports do exchange can be just streams of bytes or the messages may have a defined data-type. Furthermore, the buffer-function of the ports make it possible to synchronize processes.

Port: 1. Part of the Random Access Memory or part of the processor's address-space that contains data coming from outside the computer (keyboard, disk-drive) or to where data is sent outside the computer (disk-drive network, printer, screen). Ports are therefore unidirectional. The operating-system often transforms port-data into a more programmer-friendly format and furnishes it as → socket-data (page 273). The information about disk-drives gained from ports usually is used to extend the file-system, so that the user needs little special knowledge to access the data stored externally.

2. A piece of hardware that furnishes the ports (cards for connecting peripherals like USB- or IEEE1394/FireWire/iLink-drives)

3. Local address-numbers of the TCProtocol ranging from 0 to 65535 (65536 possibilities given by a two-byte-address, see page 33). These address-numbers are often bound to higher-leveled network-protocols like the HyperText Transfer Protocol (HTTP) on port 80, or Java-specific Remote Method Invocations (RMI) done over port 1099. For more on the TCProtocol in computer-networks see page 363.

The Java Runtime Environment (JRE) on a platform is brought to run as a process of its own. Two opened terminal-windows, with two “java”-commands dispatched at the same time, realize two different Java Runtime Environments (two processes) on the same native platform.

Inter Process
Communication

Two processes are also needed if the Java Runtime Environments are started in *non-overlapping time-intervals*. In that context, Inter Process Communication (IPC) includes ideas of how to store data for later use by a newly started Java Runtime System. This type of IPC requires to make data persistent!

Any communication-process between different Java Runtime Environments existing on the same computer, or different computers in a network, almost automatically suggests a multithreaded structuring of the communication-software. Thus parallel programming can be seen as an important part of (virtual) networking!

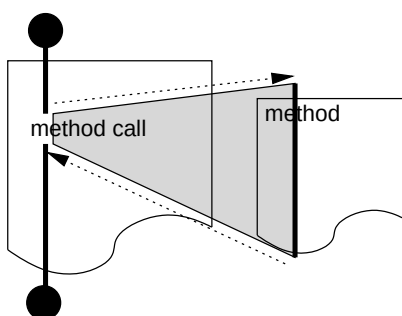
Concept: Blocking and Non-Blocking Methods

Concept: Blocking and Non-Blocking Methods — Another aspect in approaching multithreaded programming comes from considering so-called non-blocking methods. As described on page 128 and by the graphic on page 119, standard method-calls let the program work on the code-lines of the method and then return to the next operation after the method-call. From the perspective of the calling-code the program-run is blocked at the method-call until the method-run returns. This view suggests the name **blocking methods** for the standard way of calling and working through methods (also called “synchronous” methods).

blocking
method

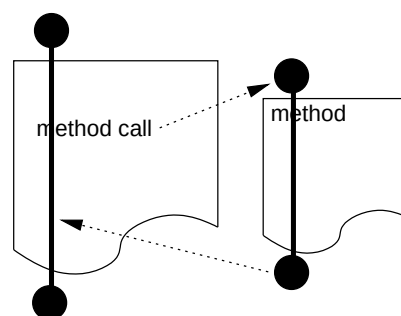
(blocking) method call

calling code has to wait
for method to finish



non-blocking method

(instantiates its own thread)



Another imaginable reaction to a method-call could be an immediate return and continuing work on the caller code, while receiving possible return-values later. This “non-blocking” behavior of a method-call clearly has a multithreaded aspect (such a behavior has to be realized by spawning a new thread, see page 208). Therefore non-blocking methods are also called “asynchronous” methods. An example of making a previously blocking method into a non-blocking method is given by the two pieces of source-code on page 588 (blocking method) and page 589 (non-blocking method).

non-blocking
method

Non-Blocking Method: Such a method runs in a thread of its own. So the calling-code can proceed immediately after the method-call. The calling-code does not wait for the method to return, like → blocking methods do.

Blocking methods will be used when the result is needed for the forthcoming tasks of the program. For example, input-streams use blocking `read()`-methods. — Non-blocking methods are useful when the results are not-immediately needed, or in cases where the results of that method-call may need time to be generated: A non-blocking method can be used to present a message on the screen or generally for any output- or listening-task. Some Java-furnished methods readily occur as non-blocking, like `Object.finalize()` (page 152), `java.awt.Toolkit.getImage()`, `java.awt.Component.prepareImage()` (both to be found in the table on page 529); these methods all are non-blocking, they spawn a thread in which the method's code is run, but meanwhile they return to the calling-code, usually without noticeably delaying the processing of the calling-code.

Summary: Multithreading

Summary: Multithreading — The `java.lang.Thread`-class has `Thread`-objects instantiated, that furnish the methods to start a thread and to control other thread-features, giving a sort of dashboard- or control-object. The thread's code can be given by the `run()`-method inside an object instantiated from an extended `java.lang.Thread`-class; or the thread's code is realized inside a `run()`-method of some object that implements the interface `java.lang.Runnable`. If an object, instantiated from a sub-class of `java.lang.Thread`, does not contain a `run()`-method, then the thread's constructor can take the reference of an object which implements the `java.lang.Runnable`-interface and therefore has implemented a `run()`-method. There have been shown other methods to connect the control-object of (super-)class `Thread` to the method `run()` implementing the interface `Runnable`.

Methods, inherited from the `Thread`-class, that give immediate control over the thread's behavior are:

```
start() sleep() wait() join() notify() notifyAll() yield() interrupt()
```

Methods that set resources are:

```
setPriority(Thread.NORM_PRIORITY-1) setDaemon()
```

Methods that inform about the thread's state are:

```
isAlive() isDaemon() getPriority() Thread.currentThread()
Thread.interrupted() isInterrupted()
```

There are other methods in the `Thread`-class with a lot more background-information given in the ➡📖 Java Language Specification [JLS]. Apart from user initiated threads, the implementation of the Java Virtual Machine (JVM), the Java byte-code interpreter, always handles system threads like the

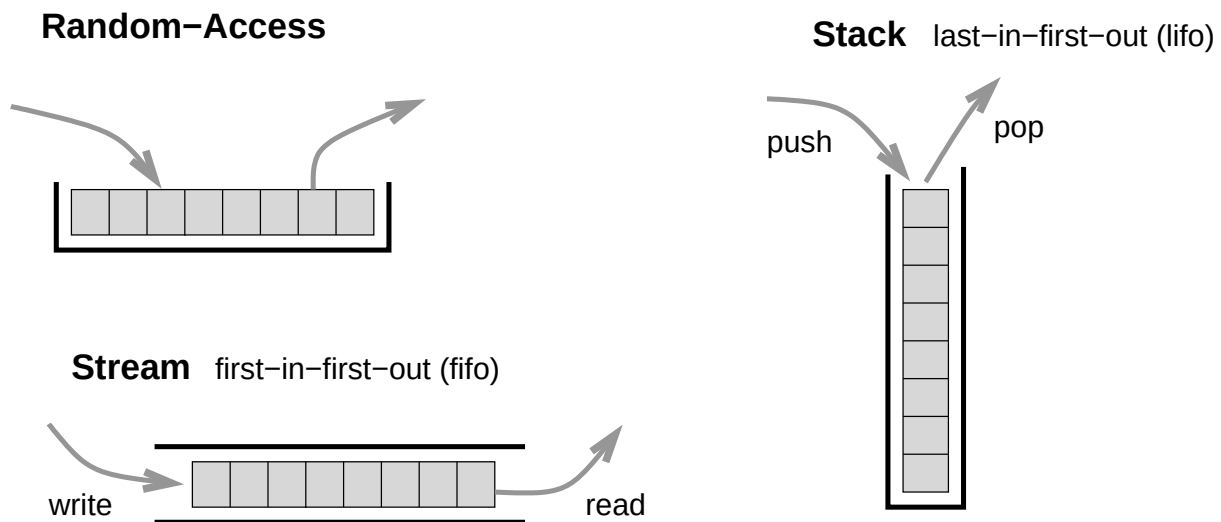
- clock handler-thread for queuing the `sleep()`- and `wait()`-requests of the user threads, a
- garbage-collector running within a low-priority thread,
- threads which are set up on calling a `finalize()`-method and in which is run the code of the `finalize()`-method.
- the main-thread of an application, in which is run the `main()`-method
- an applet-thread that organizes calling the applet's method `java.applet.Applet.start()` and the method `java.applet.Applet.stop()`
- various input and output threads for keyboard and mouse-input and screen-output

OUTLOOK: The interfaces `java.awt.image.ImageProducer`, `java.awt.image.ImageConsumer` and `java.awt.image.ImageObserver`, as implemented by the package `java.awt`, together with non-blocking / asynchronous methods (methods that take references to `ImageObserver`-objects) like `Toolkit.getImage()` and `Graphics.drawImage()`, produce an implicit multithreading context. These three Java reference-types are introduced on page 606. Omitted has been the class `java.lang.ThreadGroup`, which aggregates `Thread`-objects and serves as a uniting dashboard-object in several thread-controlling aspects.

Concept and Java: Aggregation of Data

Concept and Java: Aggregation of Data — This chapter presents repositories for collecting objects, mostly objects that contain data of some similarity. These collections also can be seen as little non-persistent databases.

Arrays and vectors reserve a numbered sequence of storage-space, so that a stored object can be retrieved by giving the number of its storage place. Storing and retrieving happens independently of the status of any other storage place, and therefore this way of storage is called **random-access storing and retrieving**.



Non-random-access storing and reading is presented later with streams and stacks. Streams⁹⁵ realize **fifo (first-in-first-out)** storage, the concept is introduced on page 272, a first Java programming example is given on page 291. The so-called **stacks** realize **lifo (last-in-first-out)** storage.

Java-arrays have a fixed number of individual storage-places, this is meant when speaking of the “fixed length of arrays”, therefore array-enlargement means creating a new, larger array and then copying the elements of the original array into the larger one. This functionality has been pre-programmed into Java-“vectors”, thus Java-“vectors” represent array-like containers of automatically and dynamically adjustable length.

Then there are also Java-hashtables, where the number that marks an element has been substituted by a so-called **key**. Accessing an element in a hashtable then needs the key-data instead of the number.

There are also methods for producing or analyzing such collections of data: For example, objects of Java-type `StringTokenizer` search for spaces in given strings, cut them out, and serve the remaining substrings as a sequence of individual strings.

Aggregation of objects may also be realized as a graph of references. Such an approach may open the wide field of recursive data-structures, tree-structures and the like. These aspects of aggregation are not treated in this text. Also the Java collection-framework which bundles further variants of aggregating data is not treated here.

⁹⁵Streams are introduced on page 272 and re-considered extensively beginning with page 291. The notion of a stream reveals several aspects of itself when scrutinizing the gray box on page 273.

Java: Arrays and Array-Type

Java: Arrays and Array-Type — Arrays gather multiple variables of the same type. Within an array, each variable is given an index and thus individual variables can be stored and retrieved by just knowing their index. Java arrays are variables of reference-type (page 76) or more precise of array-type (NOT of class-type, neither of interface-type, as can be seen when considering the syntax-diagram on page 143). Java arrays are declared and instantiated and referenced, like variables of class-type, but the array-type cannot be extended and may not have a class-loader⁹⁶. So, a Java array is a variable of reference-type but **A JAVA ARRAY IS NO OBJECT**, even if the array contains objects of class-type! Arrays have been made part of the  Java Language Specification [JLS] like the concept of “primitive-types” for being able to write faster programs.

In Java an array is a special aggregation of variables:

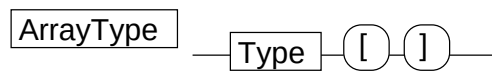
- All variables of that aggregation are of the same class-type or of the same primitive-type.
- Each variable of that aggregation is indexed by one non-negative integer, an index or number 0, 1, 2, ..., $n - 1$. The length of the array associated with that index is given by the non-negative integer n . Observe that the counting begins with zero⁹⁷!
- There may exist an array with elements having more than one index — this can be seen to implement a kind of array of array — such array is said to be multidimensional.

How can be declared a variable of array-type in Java?

```
PrimitiveType nameOfArray[] ... [];           // declaration
ClassOrInterfaceName nameOfArray[] ... [];    // declaration
```

The number of pairs of square brackets (“[]”) gives the number of indices which are to be attached to the name of the array for accessing the respective array-element. That number of indices is also called the **dimension of the array**.

dimension of
the array

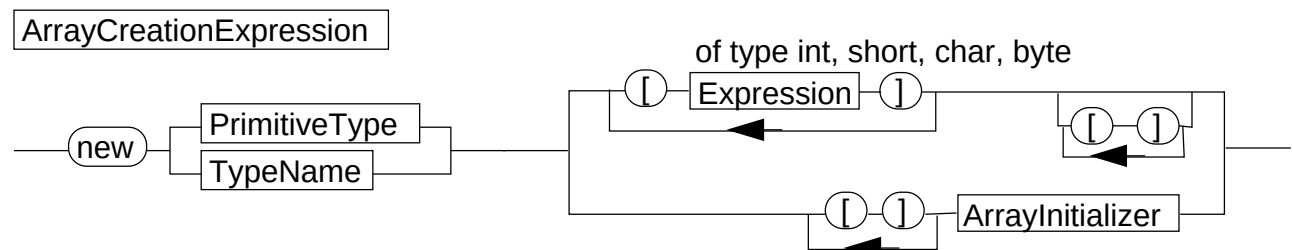


ArrayType := *Type* “[]” — Type 76

The instantiation of an array-variable can take the following form (*Type* in italics stands for some primitive-type, some class-type or some interface-type):

```
Type name[][]...;           // declaration
name = new Type[ dimension1 ][ dimension2 ]...; // allocation
// or shorter:
Type name[][]... = new Type[ dimension1 ][ dimension2 ]...;
```

Each index can take a maximal number $n - 1$, the number n is called the **length of the dimension** represented by that index variable. As can be seen above, THE LENGTH OF A DIMENSION IS FIXED ON ALLOCATION, NOT ON DECLARATION. The lengths of the dimensions can of type byte, char, short but not long! *Lengths of dimensions of an array cannot be changed by assignment after creation*⁹⁸.



⁹⁶If a class-definition is not known to the Java Runtime Environment (JRE), then a class-loader makes the class known to that Java Runtime Environment. This basic mechanism is described beginning with page 433.

Arrays with entries of primitive-type do not have a class-loader. Arrays with entries of class-type are associated with the class-loader of their entries.

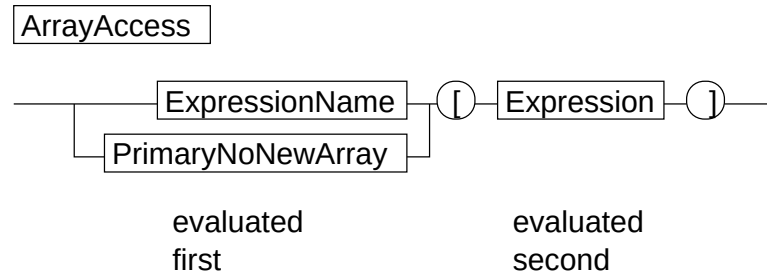
⁹⁷Beginning to count n pieces with zero and thus having to end with $n - 1$ often may lead to confusion if not accustomed to it. Since on hardware-level counting begins with zero, this counting method spares the platform (Java compiler, runtime-environment) an additional shifting operation thus improving performance in array-access intensive code. (Beginning to count with zero needs not to be deemed outright counter-intuitive: The chimes of a clock for a full hour 7 begin at zero and usually end after 6 seconds. So this counting method may be called **chime-like counting**.)

⁹⁸Aggregations with changing length after instantiation can be realized with the standard class `java.util.Vector`. (Remember that the values of `String`-objects also cannot be changed after instantiation, just like the lengths of arrays.)

ArrayCreationExpression := new *PrimitiveType* < "[" *Expression* "]" > {"[" "]" } | new *TypeName* < "[" *Expression* "]" > {"[" "]" } | new *PrimitiveType* < "[" "]" > *ArrayInitializer* | new *TypeName* < "[" "]" > *ArrayInitializer* — *PrimitiveType* 84, *Expression* 90, *TypeName* 82, *ArrayInitializer* 236

After declaring and allocating an array, the empty places of the array-elements can be filled:

```
name[index1][index2]... = Value or variableName;
                                // assignment, instantiation
```



ArrayAccess := *ExpressionName* "[" *Expression* "]" | *PrimaryNoNewArray* "[" *Expression* "]" — *ExpressionName* 82, *Expression* 90, *PrimaryNoNewArray* 91.

If such an array is referenced without checking its length first, then there may occur an exception of the kind `java.lang.ArrayIndexOutOfBoundsException`. Either a try-catch-construct that catches the `OutOfBoundsException`-exception or a check of the length-variable of the array can avoid an interruption of the program.

Some aspects like, array-initializers, the length post-qualifier for getting the length of an array, and the defining of arrays of arrays, are demonstrated in the source-code below: (Array-initializers are used in a syntactic structure which has not been anticipated by the syntax-diagram above, but the program below works well (Java Version 1.3)! This stresses the meaning of experimentation in programming.)

```
public class Arrays
{ public static void main( String unused[] )
{ int [] intArray = new int[ 3 ];
  // Arrays of primitive-type get default initialization intArray[0] = 0...,
  // or arrays are initialized elementwise:
  // (chime-counting: numbering begins with zero!):
  intArray[ 0 ] = 1; intArray[ 1 ] = 2; intArray[ 2 ] = 3;

  // Instead of initializing elementwise by assignment; use an array-initializer:
  float floatArray[] = { 0f, 1, 2f, (float)3. };
  Integer[] I = { new Integer(2), new Integer(4), null };
  Object[] o = { new Integer(2), null, new Float(3.1415) };

  // Arrays of arrays can be built (realizing multidimensionality):
  byte[][] B = { {1,2}, {1,2,3} };
  byte[][] C; C = new byte[2][]; C[0] = new byte[2]; C[1] = new byte[3];

  // The FIXED length (or lengths) of an array is given by the length-constant:
  System.out.println
  ( "Array Test C[0][0]=" + C[0][0] + " C.length=" + C.length +
    " (2) C[1].length=" + C[1].length + " (3)"
  );
} }

/*
C:\WINDOWS\JavaCodeFiles\Basics1\Aggregation>javac Arrays.java

C:\WINDOWS\JavaCodeFiles\Basics1\Aggregation>java Arrays
Array Test C[0][0]=0 C.length=2 (2) C[1].length=3 (3)

C:\WINDOWS\JavaCodeFiles\Basics1\Aggregation>
*/
```

An array of array-type `java.lang.Object` comes very close to a database into which almost any kind of object of class-type can be stored, even variables of primitive-type which have been properly wrapped by objects of class-type `Number`. The source-code above shows arrays declared, allocated and filled with elements in one single step using a construct called the **array-initializer**:

```

// name of the array
char a [][] = { { 'a' , 'b' }, { 'x' , 'y' , 'z' }, { '1' , '2' } }

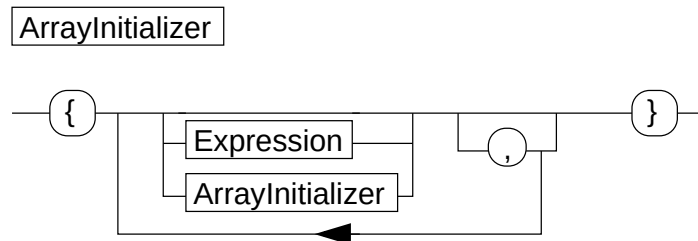
// a variable of a fixed type (primitive or reference),
// here "char"

a[1, 2] = 'z'

position of the element
position of the brace-pair that contains the element in the next higher level

```

Identifying the index-sequence of a chosen element amounts to chime-like-counting braces with a final count of the position of the element. Here, curly braces {..., ..., ...} give an array-initializer with the values inside the braces separated by commas:



ArrayInitializer := "{ [(*Expression* | *ArrayInitializer*) [,] { (*Expression* | *ArrayInitializer*) [,] } }"

— Expression 90

Array-types are reference-types, that means when the identifier of an array-variable is assigned to another identifier (`arrayNew = arrayOriginal`) only a new link to the existent array-variable is created. A copy, or rather a clone⁹⁹, of the array-variable can be produced using the `arraycopy()`-method in the `java.lang.System` class.

Now the parameter-list of the `main()`-method of Java-applications (page 72) can be properly assessed: `public static void main( String [] arrayName )` makes an array of `String`-objects available in the block of the `main()`-method. From where does this array come? This array is taken from the terminal-command that acts as an external trigger for calling the `main()`-method: Entered into the terminal-window of the native operating-system, the command "`java fileName string1 string2 ...`" "calls"¹⁰⁰ the `main()`-method with an array containing the space-separated values of that input-array. The length of the array is given by the number of `String`-arguments at the time of creating the array within the Java Runtime Environment.

OUTLOOK: Arrays containing solely variables of primitive-type have no class-loader. Arrays with class-type entries have a class-loader. For more see page 443. Further means of array-manipulation, like sorting can be found in the class `java.util.Arrays` (for more on packages see next chapter).

Java: Vectors

Java: Vectors — Arrays can only take the number of elements to which they have been initialized; for enlarging the length of an array, another array has to be allocated and has to be filled with the already existing elements.

Java furnishes a pre-written array-like class whose objects allow to extend the number of elements without explicitly re-instantiating another array. This type of class is called `java.util.Vector`¹⁰¹. Its use can be seen by consulting the ➡ Java Platform API Specification [JPAPIS] or by referencing the example on page 228.

⁹⁹For distinguishing the notions "copy" and "clone", see page 144 and page 191.

¹⁰⁰Creates a Java Runtime Environment (JRE), loads the byte-code from the file `fileName.class` into the Runtime Environment and finally calls the `main()`-method of the loaded class.

¹⁰¹The qualifier indicates that the class `Vector` is found inside the package `java.util`. Packaging gives a non-hierarchical way to name classes (contrary to path-names in a Hierarchical File System). For further information consult the next chapter.

Java: StringTokenizer

Java: StringTokenizer — A given string can be seen as a succession of individual words or tokens that are separated by space-characters. For a given string in Java, instances of the class `java.util.StringTokenizer` automatically furnish access to the individual words or tokens as substrings. In that way this class furnishes a sort of aggregation of substrings. For more on that class consult the   Java Platform API Specification. The `java.util.StringTokenizer`-class is used in a program on page 386.

Java: Collection-Framework

Java: Collection-Framework — Variables of array-type and Vector-objects implement aggregation of elements by numbering them. Maps aggregate elements by assigning each element a key-element (maps are also called hashtables). A general approach to aggregation or “generalized mapping” is attempted with the “Collection-Framework” inside the package `java.util`.

The concepts applied here are closely related to those found in elementary mathematics when considering mathematical sets and classifying mathematical functions and relations (functions as one-to-one relations and the like). For the Java-specific implementation see the   Java Platform API Specification [JPAPIS] beginning with the interface `java.util.Collection`. The Java Collection Framework essentially bases its structure on three notions: Set, List, Map. A set represents a collection of elements which all differ from each other. A list can be seen as a precursor to a Vector-object or an array. For hashtable-like structures consider implementations of the interface `java.util.Map`.

Framework: Elementary frameworks are single abstract classes or interfaces. Their implementations can differ but their basic function remains the same. A non-elementary framework is taken to be a package containing abstract classes and interfaces making it possible to insert or exchange implementations easily. The security package represents such a framework because various encryption modules can be used without changing the higher level package functionalities.

The interface `java.util.List` is mentioned here because it serves demonstrational purposes: — This interface gives an example of a sub-interface because it extends the interface `java.util.Collection`. — The method-signatures of the interface `java.util.List` are given mostly as implemented methods in the abstract class `java.util.AbstractList`.

Class-names beginning with the word “Abstract” OFTEN indicate that the rest of the name that suffixes the word `Abstract` is the name of an interface! This can be checked by referencing the   Java Platform API Specification [JPAPIS]. — When considering the programming of a Graphical User Interface a Java programmer encounters a class named `List` (`java.awt.List`), how to avoid a possible naming conflict is described in the next chapter.

a rule for
class-names: Ab-
stractInterface

Transforming Aggregates

Transforming Aggregates — Differing concepts of aggregation like vectors, arrays and streams and collections produce the need of transforming them into each other. This transformation represents just another way of higher-level non-destructive casting.

A typical problem concerning ARRAYS AND STREAMS¹⁰² may be formulated as follows: How to read streams (no knowledge about their length) into arrays (length must be fixed)? Most probably the streams have to be buffered somewhere until the length of the data can be determined.

For example, if the stream transfers several serialized objects, choose a `java.util.Vector` as buffer, because the method `java.util.Vector.toArray()` generates an array of those objects.

If the stream is taken as a simple succession of bytes or chars use the `ByteArrayInput/-OutputStreams` `CharArrayReader/-Writer` `StringBufferInputStream`. A fragmented example of the wealth of possibilities of transforming aggregations in Java is given by the following table:

¹⁰²Streams are introduced on page 272 and re-considered extensively beginning with page 291. The notion of a stream reveals several aspects of itself when scrutinizing the gray box on page 273.

line to column	Array	Vector	List	Stream
Array	<code>java.lang.System.arraycopy()</code>		<code>java.util.Arrays.asList(Object a [])</code>	<code>java.io.ByteArrayInputStream(char buf [])</code> <code>java.io.CharArrayReader(char buf [])</code>
Vector	<code>java.util.Vector.toArray()</code> of objects <code>java.util.Vector.copyInto(Object [] anArray)</code>	<code>clone()</code>		
List	<code>java.util.List.toArray()</code>			
Stream	<code>java.io.ByteArrayOutputStream.toByteArray()</code> <code>java.io.CharArrayWriter.toCharArray()</code>			if interface <code>java.lang.Cloneable</code> implemented, <code>clone()</code>

Arrays of bytes can be easily transformed to Strings by calling the appropriately overloaded String-constructor. See one of the output lines in the program on page 497.

OUTLOOK: Random-access files also have aggregational traits. The file itself is accessible by its name within the Hierarchical File System (HFS) of the underlying, native operating-system. (This gives the content a persistent¹⁰³ character.) The contents of the file have to be seen as a sequence of bytes, which allow a pointer to fix a position within that sequence (the initial position of the pointer is zero). Reading takes always chunks of data beginning with the pointer-position. For more see page 299.

Above has been considered the aggregation of data; another form of aggregation that concerns the source- and byte-code of related classes means to consider Java-packaging as presented in the following chapter.

¹⁰³Persistence means that data keeps existent even after the computer has been turned off. And the data can be accessed in a later computer session without any loss of content. Typical persistent storage-devices are hard-disks and CD-ROMs.

Structuring Code: Packaging



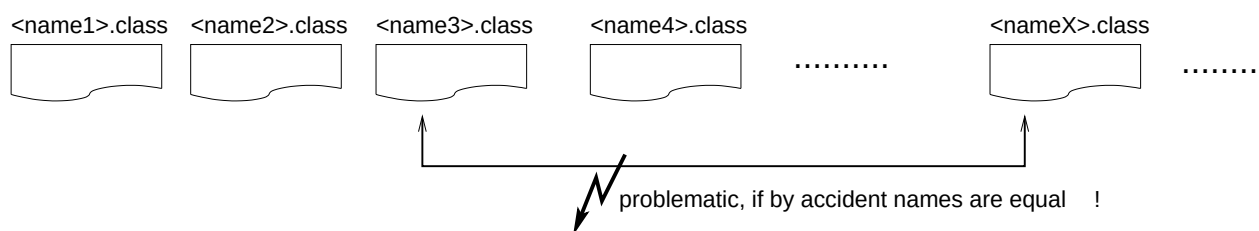
Structuring Code: Packaging — Proceduralizing code in the context of a program is possible by breaking it down into multiple methods. Remember also that, in Java, data-structures and programs can be stored as classes and interfaces. One Java program may be given by some text-files of source-code consisting of several class- and interface-definitions; compilation produces one class-file of byte-code for each class- and interface-definition. How to collect classes and interfaces of related content and functionality (program, graphics, text-manipulation, mathematical functions, ...)? How to establish a code-library of related classes and interfaces? (AGGREGATION PROBLEM)

Single files of source-code can be used to collect related class-declarations; compiling produces a set of class-files, containing byte-code. If those files are located in the current directory or found in a directory inside the CLASSPATH (remember the discussion on page 60), then the use of the classes' identifiers triggers the class-loader (page 433) in the implemented Java Virtual Machine (JVM) to use those classes; otherwise the Java byte-code interpreter returns a "Can't find class" message.



(NAMING-) PROBLEM: Some basic problem of ambiguous naming of *variables* has already been considered in the footnote on page 187, the same can happen with the naming of *classes*. — If a few classes are involved in some programming task, then an application-programmer may remember their names. If a lot of classes are involved, then the number of different identifiers may confuse the programmer, and identifiers that are chosen accidentally to have the same name may confuse the Java Runtime Environment (JRE); subsequently confusing the programmer and the end-user. (Example: The List-interface used for collecting variables, and a class-type — introduced in the context of programming a Graphical User Interface — also named List, instantiating screen-components for giving a selection to an end-user.)

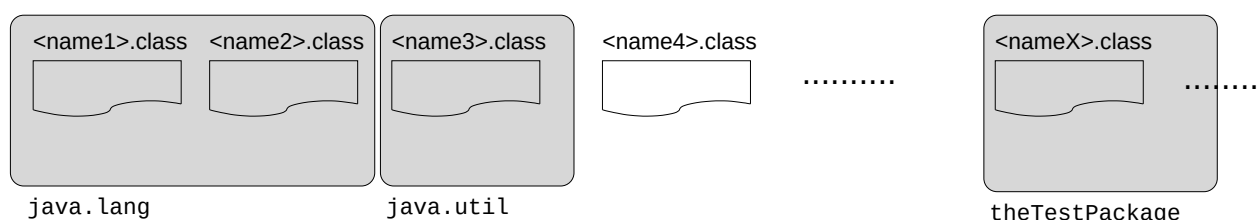
class-files loaded into the Java Runtime Environment from the current directory or in directories of the CLASSPATH:



SOLUTION: Collect a set of related classes as a package of a given name and let that name qualify the class-names of that package. Then, classes of the same name can coexist, if found within different packages. The language-construct of a "package" solves the naming problem: In Java, qualifying a class-name with its package-name **packageName.className** makes the class available for use in the calling source-code. (Example: Now the same class-/interface-names of different packages, can be discerned by using fully-qualified identifiers: `java.util.List` and `java.awt.List`)

BETTER:

Class-files in current directory or in directories of the CLASSPATH: Sorted according to their package-membership:



So there are two characteristic properties of packages in the Java programming-language:

own scope/name-space a means of avoiding conflicting names and, by using modifiers (`private`, `protected`, `public`) for classes, interfaces and members, restricting access from outside the package.



collection of classes to gather related classes, their source-code and their byte-code for easier reference in finding and using them. In Java exist no packages within packages, there exists no package-hierarchy¹⁰⁴; packaging makes a “flat” classification of code as also indicated by the graphic above.

NO packages
in packages

The Java Runtime Environment (JRE) itself makes extensive use of this package concept: For example, remember that the almost ever-present `String`-class is a class of the `java.lang` package. More about that is said later in this section.

Package: (Java) A set (collection or group) of Java-classes which is given a name. The package-name defines a name-space for its members. Otherwise classes are taken to be in the so-called default-package. A package in Java can be given in three different forms:

1. A set of text-files that contain the source-code in which the classes or interfaces of the package have been declared.
2. A set of “class”-files of byte-code inside a single directory. In many cases the text-files with the source-code are found in the same directory.
3. A JAR-file with a directory-structure and the associated class-files containing byte-code.

A package-name may consist of period(.)-separated words, the *period-separation* reflects only the *relative directory-path* under which the package-members will be found. There are no packages within packages in Java (no containment-hierarchy of packages, THE PACKAGE-SPACE IN JAVA REMAINS FLAT).

A JAR-file or a ZIP-file collects files which have been transformed so that they use less memory than before. JAR-files use the ZIP-file-format to compress files and to aggregate individual files into one single file. Such a JAR-file can bear a directory-structure and may contain signed pieces of software. The ZIP-format is further mentioned on page 326 considering its uses and, more oriented towards the file-compression-concept, on page 522.

Modularization: 1. Independently of any programming-language: Usually done in advance: Dividing a problem or system into different entities, that are hopefully loosely coupled. This process of division should improve the overall clarity and manageability. 2. Java’s packaging-infrastructure provides an implementation of the modularization concept.

Proceduralization is the strategy of breaking down pieces of source-code into named and easier manageable pieces, a bottom-up approach. In contrast to this, modularization has a more top-down aspect in structuring the entire program into modules or integrating the program into the functionality of pre-existing modules. Modularization can be seen as a coarse precursor to object-oriented structuring of a programming-problem.



In short: In the Java Runtime Environment (JRE), packages are sets of classes that are given their own name-space. How is source-code marked as belonging to a package? And on the level of the operating-system: How are the files of byte-code stored, so that they form the Java-package?

Java: How to Write and How to Install a Package

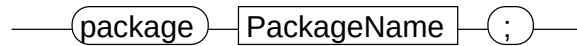
Java: How to Write and How to Install a Package — Any source-code can be rewritten to belong to a (non-default-)package just by inserting a one-line package-declaration. Then the essence of installing a package amounts to placing the class-files of the package’s byte-code inside some appropriately chosen directory. Details are found in the following sections.

Java: Writing a Package

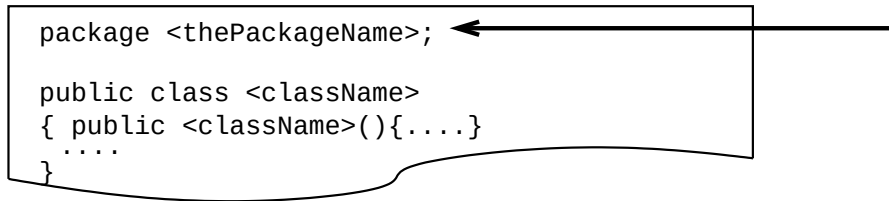
Java: Writing a Package — How to tag source-code so that its class-file(s) are recognized by the class-loader as a member of a package? A file of source-code can be made part of a package by BEGINNING THE SOURCE-CODE WITH A PACKAGE DECLARATION:

¹⁰⁴Although the Java Language Specification [JLS] uses the term “subpackage”, this term there refers only to naming convention, not to accessibility or arrangement of code! Therefore this term does not appear in this text.

PackageDeclaration



PackageDeclaration := package *PackageName* ; — *PackageName* 241
 <className>.java



The name of the file remains that of the single public class in the source-code.

```

/* In the simpler cases:
   The file containing this source-code and the byte-code compiled thereof
   are put into a folder named 'theTestPackage'.
   If this folder is in the user's classpath, for example in the current directory, then
   the source-code is found during compilation, the byte-code is found during runtime.
*/

package theTestPackage; // -----
public class Print
{
    public Print(String s){ System.out.println(s); };
    public static void main( String unused[] )
    { System.out.println( "application as package member, can be run" );
    }
}

/* [localhost:JavaCodeFiles/Basics1/SCPackaging] marycosway% javac theTestPackage/Print.java
   [localhost:JavaCodeFiles/Basics1/SCPackaging] marycosway% java theTestPackage/Print
   application as package member, can be run
   [localhost:JavaCodeFiles/Basics1/SCPackaging] marycosway%
    
```

applications
run as
package-
members

Inside a package, a class with a main() -method may be triggered into running by using the java-tool.

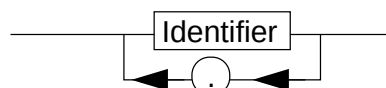
No package-directive at the beginning of the source-code, means that the source-code is part of the **default-package** or **no-name-package**, also called unnamed- or anonymous package. Thus each piece of Java source-code without a package-declaration has to be considered as being part of the so-called default-package. This is also illustrated on page 437, where the frames of all the named packages like java.lang, java.io are supplemented by an unnamed frame marked as the default-package. Thus inside a Java Runtime Environment (JRE), each class or interface is located inside exactly one package!

default-
package

every class
inside one
package

Can a package be composed of multiple files? Yes, each of those text-files of source-code has to have the package-declaration at its beginning. Can a text-file contain multiple package declarations? No, only one package-declaration at the beginning of the text-file is permitted, that package-declaration registers all classes or interfaces, which are defined inside the text-file, as members of the package.

PackageName

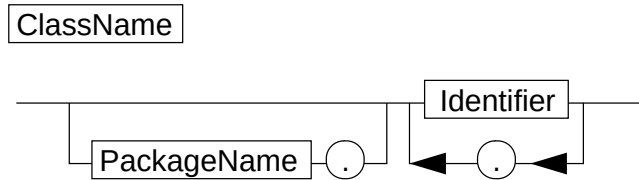


PackageName := *Identifier* { *Identifier* } — *Identifier* 81

Package-names may be composed of several identifiers separated by dots, like java.lang, java.awt, java.awt.image or com.firm.utilities, all of which are distinct and unrelated package-names. A hierarchical structure is pure naming-convention and usually helps the user in memorizing and recalling the contents of that package!

There exist **NO SUB-PACKAGES OF PACKAGES**, that means packages inside of packages cannot be defined! The name-space of a package is flat, meaning that access to a class inside any package remains to be of the form **packageName.className**, even if the package-name contains some dots.





`ClassName` := [`PackageName` .] `Identifier` { . `Identifier` } — `PackageName` see above, `Identifier` 81

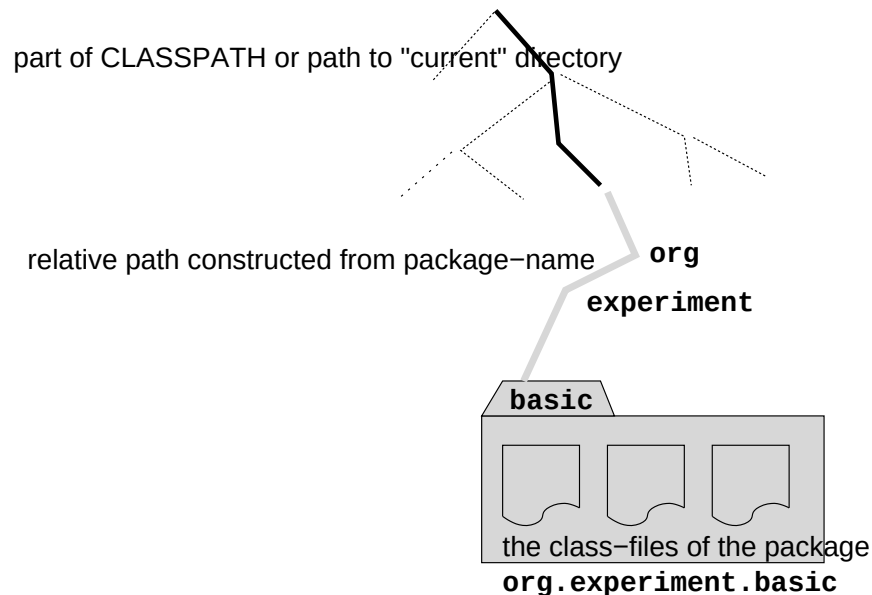
In the syntax-diagram above, the multiple dots between identifiers indicate the possible referencing of inner classes. The package-name may qualify that construct using a dot too. And the package-name itself may be given as a succession of identifiers separated by dots (useful for memorizing and for indicating a directory-structure, but not for indicating a sub-packaging structure). Here one dot has been used in three different contexts!

Installing a Package

Installing a Package — If source-code has been defined as being part of a package, and has been compiled, then how are the class-files — the byte-code — to be stored for future referencing?

1. Make a directory-path from the package-name by substituting the dots in the package-name with file-separators (past page 60) of the operating-system and
2. Gather all the class-files of that package in the last directory of that relative path
 - a) The folder that is given by the relative path of these directories and the class-files can be converted into a ZIP- or JAR-file by using the appropriate tool on the level of the operating-system. (A `jar`-tool or `-command` is part of the Java Development Kit (JDK), see page 325. Furthermore, JAR-files can contain signatures as shown on page 491.)
3. Take the folder that is given by the relative path of these directories together with the class-files in the last directory. And put that folder into a directory of the `CLASSPATH`-variable (see also page 60) or into the current directory¹⁰⁵. — To test the package, simply put its folder into the current directory!

Installing the package `org.experiment.basic`



Now the package members are referencable in any Java Runtime Environment (JRE) on that computer-platform. The only exception occurs if the package's directory begins in the current directory; to find that directory, the Java Runtime Environment has to be started from that directory! Of course, those classes of the package that are to be referencable from outside the package have to be declared `public`, and consequently they have to appear in their own file of source-code (Remember the rule: Only one `public` class per text-file of source-code)!

¹⁰⁵The current directory usually is part of the `CLASSPATH`-variable!

How to install a pre-written Package

How to install a pre-written Package — Whether a pre-written package is given on a disk or loaded with a browser via a network, a package comes as a file and, if necessary, unpacks into a directory-structure with class-files. As can be imagined, the installation procedure consists solely in doing the following:

Put the folder, containing the directory-structure with the class-files or the file, into a directory of the CLASSPATH-variable. Or add the directory, which contains the packages, to the CLASSPATH-variable. (How to extend the CLASSPATH to a folder of one's choice has been described beginning with page 60.)

Beginning with the Java Version 1.2 packages given as files in the JAR-format also can be put into an ext-folder to be found within the files of the Java Runtime System (JRS) that produces the Runtime Environments (JRE). Some pre-written packages like the Java Cryptography Extension (JCE) are required to be placed inside the ext-folder (page 496). Especially extension-packages like those beginning with a token “javax.” are to be found inside the ext-folder.



Again, having downloaded some class-files having a package-name beginning with sun.com. or some other package with the name com.somenam.e.java.package.somefunction, put them in one directory of the CLASSPATH-variable.

Often the operating-systems furnish a tool that enables the user to peek into ZIP-files; when using it, the default packages can be mostly found in the file rt.jar (Java Version 1.2 and higher) or classes.zip (Java Versions below number 1.2).

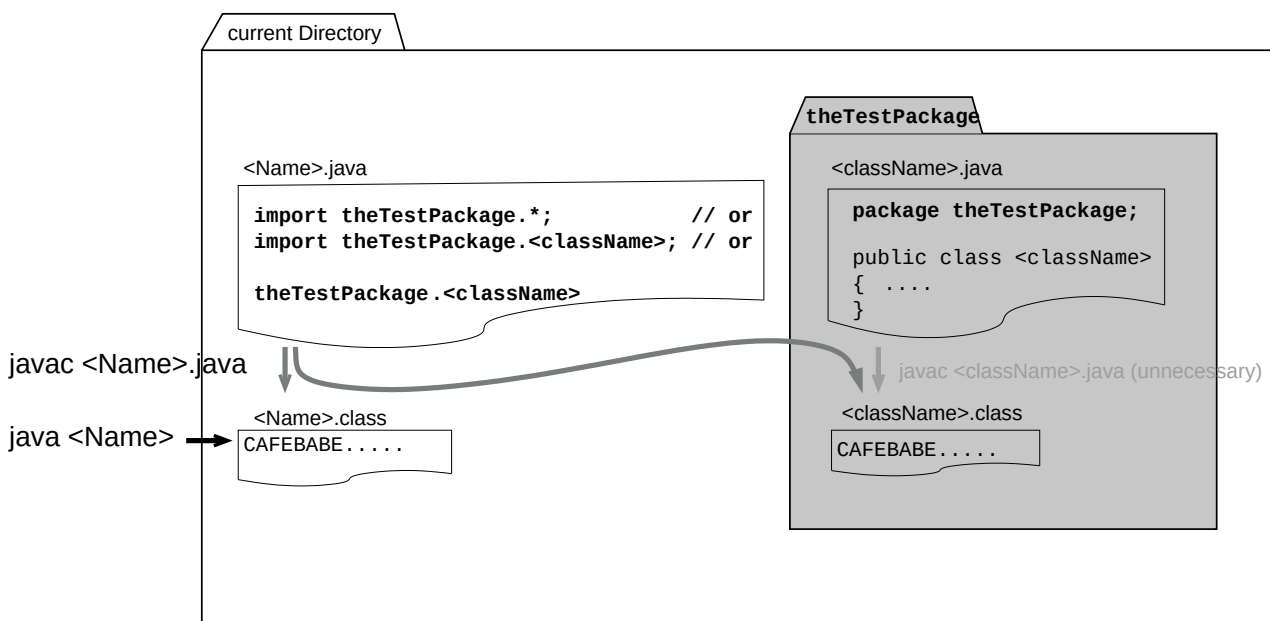
Alternatively, this peeking may be done with the jar-tool that comes with the Java

Development Kit (JDK) (page 325).

Similar to any other class-file containing Java byte-code, classes of packages are searched for in the current directory and in the directories contained in the CLASSPATH-variable of the operating-system. BUT in the case of packages that search is extended by adding the relative path, that can be built from the period(.)-separated name of the package. This can be considered a remarkable extension of the searching-range of the class-loader.

As a useful convention, considered to assure global uniqueness of the package-names, can be seen the inverted URL-naming convention (www.hostName.com -> com.hostName.www.localPackageName).

Below has been visualized a simpler example of a file- and directory-structure which incorporates a package called “theTestPackage”:



If referenced, the source-code of the package is compiled automatically, together with the calling code.

The scheme visualized above may deviate in following points: The source-code may be somewhere else; only the path-location of the class-files becomes relevant for the implemented Java Virtual Machine (JVM). But then the automatic compilation feature, as indicated above, may not work.

Again, where can packages be located? The Java Runtime Environment uses a **default class-loader** to search directories indicated by the CLASSPATH-variable (page 60) for class-, interface- or package-names.

So, the default class-loader notices class-files of named packages only if they are inside a directory of the CLASSPATH-variable (or the ext-directory as a JAR-file). Therefore either extend the CLASSPATH-variable with the directory-path of the package-class, or put the file of the package inside one directory of the CLASSPATH.

The next section shows how the Java Runtime Environment uses the package-structure itself to furnish pre-written classes. Such pre-written classes already have been used extensively in the form of `java.lang.System.out.println()` and `java.lang.String`. Why could the qualification (`java.lang.`) be omitted, why were these identifiers always recognized in the form `System.out.println()` and `String`?

Each piece of given Java-code has the package `java.lang` automatically imported, so all pre-written classes of the package `java.lang` are available without qualifying its classes with the package-name `java.lang`. This has been already indicated in the footnote on page 193.

Thus, each piece of Java source-code can be thought as being given an invisible import-declaration of the form `import java.lang.*`!

How to use an installed Package

How to use an installed Package — As already indicated, the default class-loader of the Java Runtime Environment (JRE) searches for class-files of byte-code only in the current directory and the directories given by the CLASSPATH-variable, as well as the ext-directory. Package-members are only found in a relative directory-path added to the searched directory, where the relative directory-path is given by the package-name with the dots are automatically substituted by the file-separators of the current computer-platform (“theTestPackage.” becomes “theTestPackage/”). So the class-(or ZIP- or JAR-¹⁰⁶) files of the package have to be in one of those directories to be referencable. The standard way of referencing has been mentioned before:

```
packageName.packageMember
// Useful for explicit or one-time referencing the package member.
```

The import-declaration alleviates the application-programmer from the necessity of using a fully-qualified package-reference:

```
import packageName.packageMember;
// The package-member class or interface
// can be referenced solely by its name packageMember.
import packageName.*;
// ALL package-members, classes or interfaces
// can be referenced solely by their name packageMember.
```

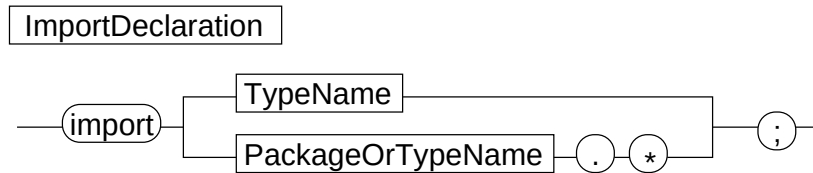
The asterisk is a wildcard. The import-declaration introduces just a writing convenience for the programmer. CAUTION: Be aware that the identifier `name.*` does not include packages or classes that have to be accessed by the identifier `name.*.*`!

Sun’s Java compiler uses an import-on-demand strategy; thus, ideally, neither of the mentioned possibilities above consumes significantly more platform resources (processor-time, RAM-space) than the other. The use of wildcards re-introduces the possibility of conflicting class-names (exactly this was the reason packaging has been introduced). But such ambiguities in the “imported” packages are solved easily by using the fully-qualified (package) class- or interface-name, as has been shown in the footnote on page 187 (member-variables of the same name) and when motivating the package-concept beginning with page 239 (`java.util.List` and `java.awt.List`).

Fully-qualified class-name: The name of a class is declared in the source-code following the keyword `class`. If, at the beginning of the source-code, a package-keyword declares a package-name, then the fully-qualified class-name is given by the period(.)-separated concatenation of the package-name and the class-name.

In those cases where only the byte-code is given as a class-file in the file-system, the file-name that gives the class-name (`String`) and the package-name (`java.lang`) may be deductible from a relative directory-path (the one that can be seen when looking into the file `rt.jar` (Java Version 1.2 and higher) or `classes.zip` file (Java Versions 1.1.x and lower) with an appropriate software tool like the `jar`-tool).

¹⁰⁶For the ZIP- and JAR-formats of files see the section about archiving Java files on page 325.



ImportDeclaration := *SingleTypeImportDeclaration* | *TypeImportOnDemandDeclaration*

SingleTypeImportDeclaration := import *TypeName* ;

TypeImportOnDemandDeclaration := import *PackageOrTypeName* . * ; — *TypeName* 82, *PackageName* 241.

<ClassName>.java

```
import <thePackageName>.*
class <ClassName>{ ....
    ....
```

```
/* (PREREQUISITE)
The directory, which contains the file of byte-code compiled from
this piece of source-code ---
this directory contains a folder with the files of byte-code
(or the source-code) of the imported package.

Alternatively the byte-code of the package may be put as a single JAR-file
into the 'ext'-folder of the current Java Runtime System.
*/

import theTestPackage.*; // <-----
public class HelloPackage
{ public static void main( String unused[] )
  { new Print( "Hello Package!", " );
    new theTestPackage.Print( "Your package class is used," );
    System.out.println( new Print("and has the name: ").getClass() );
  } }

/*
C:\WINDOWS\JavaCodeFiles\Basics1\SCPackaging>javac HelloPackage.java

C:\WINDOWS\JavaCodeFiles\Basics1\SCPackaging>java HelloPackage
Hello Package!,
Your package class is used,
and has the name:
class theTestPackage.Print

C:\WINDOWS\JavaCodeFiles\Basics1\SCPackaging>
*/
```

Remember the access-control considerations on page 175. **Default** declaration of classes or interfaces means that, after having imported the package, the identifier is only accessible for code inside the package. **Protected** declaration of classes or interfaces means that the identifier is only accessible inside the package or in sub-classes of classes from the package. **Public** declaration is ONLY POSSIBLE WITH ONE, SINGLE, UNIQUE CLASS IN A FILE OF SOURCE-CODE; therefore to have every package-member public, make the package a collection of files with each (publicly declared) package-member in one single text-file of source-code. Non-public classes cannot be accessed from outside the package.

Package Introspection

Package Introspection — Beginning with Java 1.2 there exists the `java.lang.Package`-class. This class (together with the package `java.lang.reflect`) enables the programmer to get a lot of information about the loaded packages within the current Java Runtime Environment.

A factory-method `getPackage( String packageName )` returns a reference to the `Package`-object of given name. And the factory-method `getPackages()` returns an array of all currently loaded `Package`-objects. The `getClass()`-method of the super-class `java.lang.Object` makes it possible for a Java program to get the names of the classes inside the packages.

Instance-methods allow to check for compatibility of packages.

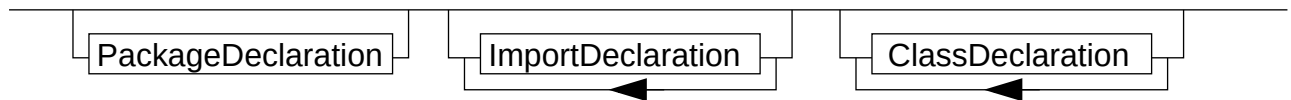
Compilation, Packages and Compilation Units

Compilation, Packages and Compilation Units — Java source-code can be found in text-files and within compressed text-files in a JAR-file. The source-code of a given programming-task may be categorized as belonging to classes and/or packages within text or JAR-files, or even within databases. Considering larger programming tasks and the numerous ways to structure and store the source-code, the idea of a compilation unit means to identify those parts of a source-code that can be compiled independently.

Compilation Unit: 1. Smallest unit of source-code that can be compiled by the Java compiler. Within the currently used Java Runtime Environment a compilation unit usually is given by a text-file of Java source-code. A compilation-unit is part of exactly one Java package (explicitly defined or default), and a compilation unit usually is given by a text-file that contains Java source-code and contains none or one public class or one public interface.

2. Final construct of the syntactic structure of the Java programming-language. Final means that this construct does not appear in any other construct of that syntactic structure.

CompilationUnit



CompilationUnit := [*PackageDeclaration*] { *ImportDeclaration* } { *ClassDeclaration* } — *PackageDeclaration* 240, *ImportDeclaration* 244, *ClassDeclaration* 146.

The following paragraphs consider the first interpretation of the notion “compilation unit”: Remember that the Java compiler (`javac fileName.java`) allows **ONLY ONE PUBLIC CLASS PER TEXT-FILE AND THE FILE-NAME MUST BE THE NAME OF THAT PUBLIC CLASS**. The larger graphic a few pages above depicted how importing a package automatically incorporates multiple (two in that concrete case) text-files of source-code into the compilation process. This automatically lets emerge *a compilation unit consisting of several text-files*, even with multiple classes with a public-modifier.

Thus respecting the requirement **ONLY ONE PUBLIC CLASS PER FILE AND THE FILE-NAME MUST BE THE NAME OF THAT PUBLIC CLASS**, a set of public classes may be compiled at the same time quite naturally: First, break the source-code down to multiple text-files each containing maximally one public class, each piece of source-code being in the same package (also no-name-package). Second, compile a class that references or imports that package, then the relevant public classes are compiled automatically. (The source-code of the no-name-package has to be stored in the current working directory.)

Also remember that Java package members with a static `main()`-method can be run as a program.

Summary: Packages

Summary: Packages — In Java, a package essentially is given by a collection of classes and interfaces. In the Java Runtime Environment (JRE) a package defines a name-space,  so that the package’s classes and interfaces can be referenced by qualifying their names with the package-name (`java.lang.String`). — A package may be given by the collection of its text-files of source-code, these may even be compressed in a single JAR-file. — Usually a package is given as a collection of the associated class-files of byte-code, which also may have been jarred.

Java Language Classes that are package-members are referenced in source-code by the package-name qualifying the class-name or by using the `import`-declaration. Packages do not contain other packages, this circumvents the complexities of an hierarchical name-space. Therefore, although the package-name itself may be composed of dot-separated identifiers, **the hierarchical structure that suggests itself has no meaning within the Java programming-language**. Packages like `java.awt` and `java.awt.image` are totally unrelated on the language-level. The name-space of packages is flat, there exist no sub-packages!

- A package defines a name-space in the Java Runtime Environment.
- A package has a package-name (Example: `org.experimental.basic`) that is used to reference a package-member in source-code (Example: `org.experimental.basic.aClass.class`). On the language-level, the dot-separation is used only as a mnemonic help for users. (Only in the file-system

of the native operating-system, the syntactic structure (dot-separation) is used as a relative path by the Java Runtime Environment when searching the byte-code of a package-member.)

Operating-System On the level of the operating-system, all the class-files that are part of a package have to be found in a single directory. HERE the dot-separated package-name is used to indicate a relative directory-path! A package on the level of the operating-system is given by a folder(=directory) that contains sub-folders named according to the dot-separated parts of the package-name. To be found by the Java Runtime Environment (namely a class-loader), that folder has to be in the current directory or one of the directories of the CLASSPATH-variable or the ext-folder. In the environment of the file-system, different packages with similar names, like `java.awt` and `java.awt.image` become related insofar as these different packages are found in the same hierarchy of files.

- In the file-system of the native operating-system the class-files of the package are gathered in a single directory. For example the directory:
`/directoryInCLASSPATHVariable/org/experimental/basic/`
- On that level the package can be used by the Java Runtime Environment, if a directory in the CLASSPATH, together with the package-name taken as a relative path, points to the classes of the package. This means, using the example above: The CLASSPATH-variable of the operating-system has to contain the directory `/directoryInCLASSPATHVariable/`. And the associated relative path, then is given by `/org/experimental/basic/`
In many cases the relative path may be hidden inside a ZIP-file or a JAR-file. In Java versions below the number 1.2, the ZIP-file `classes.zip` contains the core-classes. Beginning with Java Version 1.2, the JAR-file `rt.jar` contains the core-classes. — In special cases a class-loader may be programmed as to load classes of packages outside the CLASSPATH. (page 439)

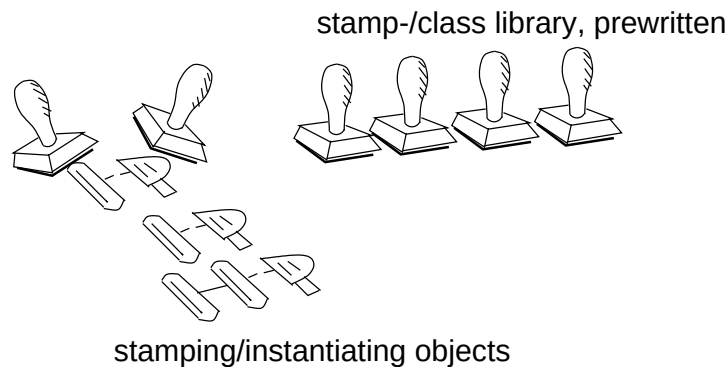
A package is a named aggregation of classes with an own name-space within the Java Runtime Environment. (A class-hierarchy is a structured index to successively extended classes. And a class itself is an aggregation of data and methods that manipulate that data.)

BEWARE of confusing the notions FILE-HIERARCHY, the PACKAGE-NAMES, and the idea of a CLASS-HIERARCHY. Although sometimes the class-hierarchy suggests names for files, the file-names need not reflect this! And package-names may differ from file-names as well as from class-names! File-hierarchies are given by a Hierarchical File System as visualized in the graphic around page 47. Class-hierarchies are a way to describe the inner structure of a class, as depicted by the graphic on page 135. Those differences in naming can be seen, when scrutinizing the applet's class-structure given by the graphic on page 615.

As a routine, the process of writing and installing a package coarsely breaks down to three simple steps:
1. Declare the package in source-code. 1a. Compile, if necessary. 2. Put the class-file of byte-code into the folder-hierarchy given by the package-name and put that folder into the CLASSPATH. Then the package is installed.

Packages have been introduced as name-space or access-control generating structures, especially in conjunction with Java class-files that are loadable by the Java Runtime Environment. The package-related files may occur in different forms: Java class-files may be archived in JAR-files (with the `jar`-utility, that compresses files into ZIP-format) or as simple folders containing byte-code (class-files) or/and Java source-code in text-files.

Standard Code-Libraries: Pre-programmed Packages



Standard Code-Libraries: Pre-programmed Packages — The package `java.lang` is not the only pre-written package that contains code for the byte-code interpreter in the Java Runtime Environment (JRE). The Java Runtime System furnishes other collections of standard classes for relieving the programmer from rewriting standard code: The so-called Java Platform Application Programming Interface (API) is given by a set of packages.

Basically, a package offers a set of class-files, stored persistently somewhere, and loadable into the Java Runtime Environment. In practical programming, packages collect solutions to related programming problems. Packages contain classes that may extend the Java language: The class `java.lang.String` instantiates sequences of characters. Readily implemented sorting algorithms can be found in the package `java.util.Arrays`.

Pre-programmed Packages

Pre-programmed Packages — Those packages of classes which are a documented part of the Java Development Kit (JDK) have package-names beginning with “java”, like `java.lang`, `java.util`, `java.math`, `java.io`, `java.net`, `java.applet`, `java.security`, `java.awt`, `java.awt.event`, `java.awt.image`¹⁰⁷. — Whereas other packages furnished with the Java Runtime System (JRS), which may not be implemented on virtually all platforms and which often are given undocumented, have names like `sun.net.www`

Therefore their use has been strongly disadvised, if the Java-code should preserve its portability! All these packages and more are shipped as a (compressed) ZIP-file or JAR-file. The names of the collection of these files are `classes.zip` (with Java versions below Number 1.2) or `rt.jar` (beginning with Java Version 1.2, Mac OS X seems to prefer the name `classes.jar`).

Core-Classes and the Java Runtime System: Location/Content

Core-Classes and the Java Runtime System: Location/Content — On the level of the operating-system, consult the Java Platform API Specification [JPAPIS], which informs about the up-to-date directory-structure of the Java Runtime System (JRS). Alternatively, to see where such a directory-tree has been installed, search for files named “java” (In Unix-like terminal-windows for example enter the command `find / -name java -print`). Or more direct, by printing out the value of the PATH-variable (`echo`

¹⁰⁷Remember the three last names are flat, the hierarchy hinted at can be seen only as a mnemonic, a mnemonic on the language-level. But on the level of the file-system, the class-files are stored in a relative path built from the point-separated name!

\$PATH should produce an output with some java-directory-name in it). If multiple versions of Java Runtime Systems have been installed, how can the active one be found out? Consider the PATH-variable. With a working Java Runtime Environment (JRE) enter `java -version` into the terminal-window; or use the program on page 258 to read out the directory-path of the active Java Runtime System (JRS). Using the Java compiler (`javac`) or starting the Java Runtime Environment (`java`) with the option `-verbose` shows the paths from where the core-classes are loaded.

The path to that file may vary from platform to platform, as indicated below:

Linux/Unix The core classes may be found at places like `/usr/lib/jdk1.3/jre/lib/rt.jar` (below Java Version 1.2 see `/usr/lib/jdk/lib/classes.zip`). The same directory may contain the `ext`-folder.

Mac OS In Version 1.1 and lower within System 9.x and lower, the core-classes may be found in `/Hard-Disk/SystemFolder/System Extensions/JDKClasses.zip`. Within the Mac OS X the core classes most probably can be found in `/System/Library/Frameworks/JavaVM.framework/Versions/1.3/Classes/classes.jar`. The `ext`-folder may be found at `.../1.3/Home/lib/ext`.

Windows The Java Runtime System may be found for example in a path like `C:\jdk1.3.1`.

If the core-packages have been located, how can their containing classes, their variables and methods be seen? Use a ZIP-archive viewer/manager or the `jar`-tool that comes with the Java Development Kit (JDK):

Linux/Unix Enter the following into the command-line:

```
jar tfv classes.zip or
```

```
jar tfv /usr/lib/jdk/lib/classes.zip.
```

The K-Desktop Environment (KDE) with a preconfigured ZIP-archive viewer/manager needs only a double-click onto the icon of the file `classes.zip` or `newer rt.jar`.

The variables and methods of a class inside a package can be seen by entering into the command-line:

```
javap periodSeparatedPackageName.classNameWithoutTheClassPostfix
```

```
like javap sun.net.www.protocol.http.HttpConnection.
```

Mac OS On Mac OS X use a terminal-window with the current working directory `/System/Library/Frameworks/JavaVM.framework/Versions/1.3/Classes/classes.jar` and enter `jar -tvf classes.jar` to see a list of the core-classes. Use the `javap`-tool for getting further information of a single class. (In Mac OS versions below X go inside the Java tool-folder. Double-click the JAR-icon, enter the Mac OS-specific “/”-separated path and file-name, check the verbose checkbox and select “Listing”, then the Java Console will show the content of the file.)

Windows Use the DOS-window, choose the appropriate directory, and enter the `jar`- and `javap`-commands as presented above.

Below is given a short description of the contents of the various packages:

java.lang contains classes for wrapping variables of basic data-type (`Boolean`, `Byte`, ... , `Float`, `Double`, `String`, `StringBuffer`), associated functionalities (`Math`) and classes that furnish basic information and control about the Runtime Environment (`Runtime`, `SecurityManager`, `ClassLoader`, `Thread`, `ThreadGroup`, `Throwable`) and the native platform (`System`, `Process`, `Compiler`). And there are the two interfaces `Cloneable` and `Runnable`. For more on this specific package consider the next section.

java.lang.reflect contains classes for analyzing the class-structure of given objects (`Constructor`, `Field`, `Method`, `Modifier`, `Array`). These classes extend the function of `java.lang.Class`.

java.util contains classes like `Vector` (array-type with dynamic length) and, for example, the class `EventObject` that furnishes more a template, a convention for handling the notion “event”. And there are lots of other useful classes.

java.util.zip furnishes for example a class called `ZipFile` with a constructor that opens ZIP-files for reading and shows the entries of the ZIP-file. (see also page 326)

java.text furnishes classes that help in analyzing strings.

- java.math** offers arbitrary precision decimals (class `BigDecimal`) and integers (class `BigInteger`) for exact mathematical computations. (That *package* is not to be confused with the *class* `java.lang.Math`, introduced on page 257!)
- java.awt** contains classes for managing and coordinating keyboard- and mouse-input and screen-output.
- java.awt.event** contains classes specific to the event handling of the Abstract Window Toolkit (AWT).
- java.awt.peer** contains Java-interfaces for mouse-, keyboard- and screen-functions that are used by corresponding pre-written AWT-classes and thus almost never encountered when doing application-programming. Their concept is described beginning with page 548.
- java.awt.image** these classes handle basics of pixelwise image-processing.
- java.awt.datatransfer** gives classes which support basics of cut/copy/paste using a clipboard, to enable an end-user to transfer data between applications. The text around page 510 describes the concept.
- java.awt.dnd** gives classes which support basics of drag and drop functionalities, to enable an end-user to transfer data between applications. (See also text and graphic on page 510.)
- java.applet** this package contains applet-specific interfaces and the single class `Applet`. The class `Applet` represents the last sub-class of the `Applet` → `Panel` → `Container` → `Component` → `Object` hierarchy¹⁰⁸. Code of class-type `java.applet.Applet` represents (the core of) a program written only for running in a secured applet runtime environment.
- java.net** gives classes for managing the sockets of Internet connections and more network-related services.
- java.io** denotes the package that manages file-access and other higher level input- and output-routines. Provides access to low level I/O functions, like sockets or `System.in`/`System.out` streams. Some classes are given to directly access the Hierarchical File System (HFS) of the native operating-system, thus producing less portable code.
- java.rmi** furnishes classes that enable a Java Runtime Environment to invoke methods of objects which have been instantiated within another Java Runtime Environment. (RMI gives the acronym for Remote Method Invocation.) A prerequisite for such a functionality is an existent network-connection between the two Runtime Environments or a native operating-system with communicating, concurrent processes. The more complex non-Java counterpart is called CORBA (Common Object Request Broker Adapter).
- java.rmi.dgc** contains an interface that concerns itself with removing unused objects from a Runtime System (dgc = distributed garbage-collection)
- java.rmi.registry** gives RMI-related supplements.
- java.rmi.server** gives RMI-related supplements.
- java.security** contains classes and interfaces for managing the access to the platform's resources. This also includes implementing public/private key cryptographic routines. The `KeyPairGenerator`-class produces a pair of such keys. Much of this package is introduced beginning with page 447.
- (java.security.acl)** contains deprecated classes that organize access-rights on the basis of mechanism of the security class. Rather use the services of the `java.security`-package.
- java.security.interfaces** furnishes interfaces to the DSA (Digital Signature Algorithm) and the algorithm RSA (Rivest Shamir Adleman).
- java.beans** contains interfaces and classes that support the Bean-oriented design of Java source-code. Java classes that instantiate Java-Bean objects have to be given by source-code with at least three properties: The Java constructs adhere to standard rules of method naming, thereby implementing so-called **Bean design-patterns** (which are conventions on class-design, for example using method-names like `issomething()`, `getsomething()`, `setsomething()`). Different Java objects can be automatically analyzed for their function by one standard tool at run-time. This is called **reflection capability** and finally these Java objects can be stored on a disk and reinstantiated at a later time often even on another platform. This is called **serialization capability**. Java objects implementing these three properties are called **Java-Beans**. The non-Java counterpart to this functionality is given by the Active-X model of *Microsoft*. (see also page 311)

¹⁰⁸ The arrow points in the direction of the super-class and of inheritance. The arrow's counter-direction marks an extension-process.

java.sql furnishes interfaces and classes to communicate with relational databases by using the Structured Query Language (SQL). (see also page 321)

Some of the classes and ideas contained in the package `java.lang` are introduced below:

java.lang: Some Standard Classes

java.lang: Some Standard Classes — The package `java.lang` is always loaded when a Java Runtime Environment starts to work on a computer-platform. The following gives only a cursory overview of that package. The actual ➡☐ ➡☐ Java Platform API Specification [JPAPIS] or documentations on disks or special books are certainly more up-to-date and more extensive.

This package introduces some standard class- and interface-specifications of Java. Among the `java.lang`-classes can be found the super-class of all classes in Java: The `Object`-class. The following paragraph lists a selection of classes found within the `java.lang`-package:

`Object Class System ClassLoader SecurityManager Compiler Runtime Process Thread ThreadGroup Throwable Boolean Character String StringBuffer Number Byte Short Integer Long Float Double Math Void Runnable Cloneable`

Precise descriptions of these classes and interfaces can be found in the ➡☐ Java Language Specification [JLS] and the ➡☐ ➡☐ Java Platform API Specification [JPAPIS] from *Sun Microsystems Inc.*.

The `Object`-class The `Object`-class, introduced on page 164, has no super-class. Every object in Java is an automatic and implicit extension of the `Object`-class and therefore inherits the methods given by the `Object`-class. The `equals()`-method, where by default `x.equals(y)` returns the same as `x == y`, tests whether the references point to the same object. This method can be overridden to implement a different comparison method (page 162). The argumentless `hashCode()`-method returns a number that uniquely characterizes the object (see page 161); another unique characterization is given by the default-version of the argumentless `toString()`-method which returns the reference to a string-representation of the object; but that method often is overridden. The argumentless `getClass()`-method returns a reference to the `Class`-object that describes the object's type. The methods `notify()`, `notifyAll()`, several overloaded versions of `wait()`, concern themselves with managing exclusive access to the object in a multithreaded environment (page 220). The protected method `clone()` can be successfully invoked only if the object implements the `Cloneable`-interface. By default, the factory-method `clone()` instantiates a shallow copy of the qualifying object (page 192). The protected, non-blocking `finalize()`-method is called exactly once before its object is removed from the Java Runtime Environment (page 152). This method has been implemented empty, if deemed necessary, a Java programmer may override that method.

The `Class`-class References of this class-type make it possible to access information about a loaded class or a primitive-type. The process of analyzing the features of a loaded class or a primitive-type is also called **reflection**: The first step towards getting information is done by returning a reference to an object of type `Class`:

```
Class c = Integer.class;
Class d = int.class;      // called class-literal, alternatively:
Class c = Class.forName( "Integer" );
// generally:
Class c = TypeName.class; // a part of the language!
```

The expression that produces a reference to the `Class`-object is called class-literal and is further discussed on page 437. The `Class`-object of a given object also can be referenced using a given *identifier* which points to that object; this involves calling the `Object`'s `getClass()`-method:

```
Class c = objectsIdentifier.getClass(); // an instance-method!
```

The instance-method above is an example for a factory-like-method which returns a reference to an already existent Java-object! Another method to get a reference of a `Class`-object uses the class-name given as a `String`-object:

```
Class c = Class.forName( ClassNameAsString ); // a static class-method!
```

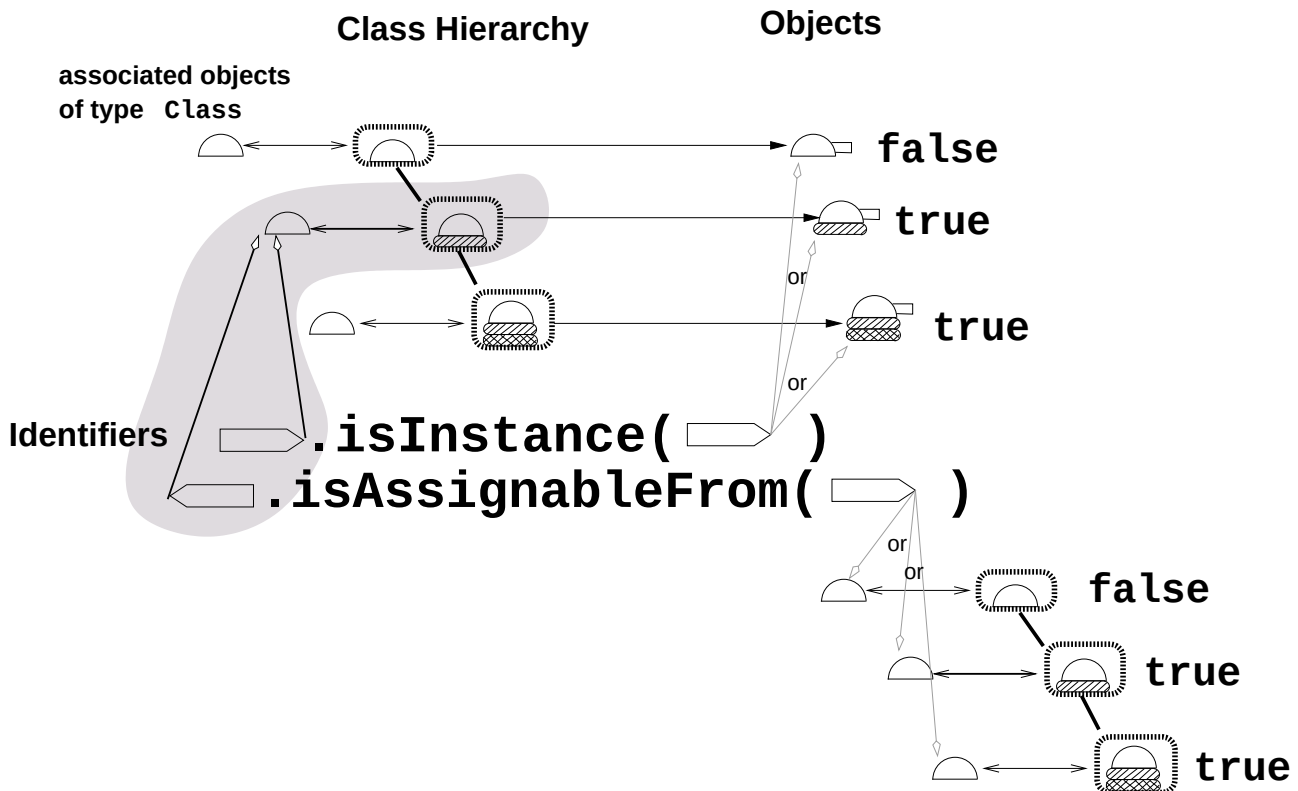

Once a reference to a `Class`-object is given, the actual information can be retrieved in a second step; the `Class`-object furnishes methods to retrieve information about the class to be investigated: Methods with an boolean return-value test for type: `isPrimitive()`, `isArray()`, `isInterface()`. The possibility of assigning (or casting) of types can be tested by using their `Class`-objects: If the expression

```
( reference1.getClass() ).isAssignableFrom( reference2.getClass() )
```

returns true, then the cast-expression `reference1 = reference2` is valid and throws no `ClassCastException`. Especially, if the type of `reference1` is of class-type, then its class has to be a super-class of `reference2` to make the cast above possible. If the expression

```
( referenceA.getClass() ).isInstance( referenceB )
```

returns true, then the cast-expression `referenceA = referenceB` is valid and throws no `ClassCastException` (implicit- or auto-cast).



A basic description of casting identifiers of reference-type has been given beginning with page 169. The instance-method `toString()` of `Class`-objects is overridden and **returns the name of the type**, if deemed appropriate with the space-separated prefix “class” or “interface”.

There are many other methods given by the `Class`-class, all of them of the form `get...()`, they all return further information about the type that is represented by the `Class`-object. Much of the returned data is given by objects of class-types found in the `java.lang.reflect` package: `Constructor`, `Field`, `Method`.

The `new`-keyword together with a constructor allocates RAM-space, methods that do the same without using the `new`-keyword explicitly are called factory-methods (page 145 and page 151): The expression `referenceOfTypeClass.newInstance()` returns a reference to a newly created instance of the class-type given by its name `referenceOfTypeClass.toString()`.

The System-Class The `System`-class has no public constructors and therefore cannot be instantiated. This class has only static methods and references static final objects (like `in`, `out` or `err`, elementary input/output-streams to be discussed later, beginning with page 297). So the `System`-class gives access to a collection of static methods and static final objects, which themselves give access to the native platform of the current Java Runtime Environment. The `System`-class also provides a list of platform-specific properties (This information can be supplemented later by platform-specific information from the `java.awt.Toolkit`-class as introduced on page 551). The three static final objects are referred to by the identifiers `out` and `err` of class `PrintStream`

and by the identifier `in` of class `java.io.InputStream`. These references enable elementary (low-level) communication with screen and keyboard. The associated methods `println()` and `print()` of the `PrintStream` were often used in the demonstration programs:

```
System.out.println(variableName); // or System.out.print(variableName);
```

Some other interesting (static) methods include `exit(int status)`, which terminates the work of the current Java Runtime Environment and therefore never returns. Or consider the method `System.gc()`, which allows the Java programmer to explicitly activate the erasing of unused (unreferenced) objects; the so-called garbage-collection. The method `setSecurityManager(SecurityManager)` installs a security-manager exactly once; the method `getSecurityManager()` gives access to that security-manager. The methods `load(String libraryName)` and `loadLibrary(String libraryName)` load dynamically-linked-libraries. The `getProperties()`-method returns a reference to platform-specific properties in the `java.util.Properties`-format (an aggregation of `String-String`-pairs which is an extension of the `Hashtable`-format):

```
os.arch=ppc          hardware-architecture below the operating-system
                      (ppc=PowerPC)
os.name=Mac OS X     name of the operating-system (Mac OS X)
os.version=10.0      version of the operating-system (10.0)
file.separator=/     the character that separates directory- or
                      folder-names in path-names of the
                      platform-specific Hierarchical File System.
                      ("/" UNIX; "\" Windows; "/" Mac OS X)
path.separator=:     the character that separates different path-names in
                      aggregations of path-names (see also page 47)
                      (":" UNIX (includes Mac OS X); "," Windows)
line.separator=      ("\n" UNIX; "\r\n" in Windows)
java.vendor=Apple Computer, Inc.  java vendor string (Apple)
java.vm.version=1.3.0  version of java (1.0, 1.1.8, 1.2)
java.class.path=.     value of Java-specific CLASSPATH
sun.cpu.endian=big    see page 533
```

and some other properties to be found by calling the argument-less `getProperties()`-method of the class `java.lang.System`. (A single-argumented method `getProperty()` returns single, key-specified items). Within a Java Runtime Environment, the supported properties and their values can be displayed by running the following little application:

```
public class TheSystemProperties
{ public static void main( String unused[] )
  { System.out.println( "System Properties" );
    System.getProperties().list(System.out);
    // java.util.Properties-class extends Hashtable and thus can be listed.
  } }
```

An example of the output of an extended version of this program is given on page 258. Furthermore, the `System`-class contains the method `currentTimeMillis()` which returns a long-integer of milliseconds elapsed since 0:00:00 UTC, January 1st, 1970. And there exists a method for copying entire arrays:

```
arraycopy( Object source, int sourceOffset, Object destination,
           int destinationOffset, int length ).
```

Another source of specific data on the current Java Runtime Environment is given by an identifier of class-type `java.awt.Toolkit` presented later beginning with page 551.

The Runtime-Class This class gives means to access the data of the current Java Runtime Environment (JRE). The Java Runtime Environment creates an instance of class `Runtime`, which can be referenced through the class-method `Runtime r = Runtime.getRuntime()`. The instance-methods include `totalMemory()` and `freeMemory()` which return the current number of bytes available to the Java Runtime Environment. Here are also found the methods `exit(int status)`, `gc()`, `load(String libraryName)` and `loadLibrary(String libraryName)` which are used by the `System`-class. Instances of the `Runtime`-class can initiate external processes on the platform with the overloaded factory-methods `exec()` that instantiate objects of the `Process`-class.

The Process-Class Inside the Java Runtime Environment, the Thread-class instantiates a control-object of type Thread which can start a thread. The code of the thread is given within a run() -method inside an associated class that implements the Runnable-interface. — Objects of class Process control a system-specific external (sub-)process. (Runtime-environments and the native operating-systems use threads and processes for running code (quasi-) parallel. Threads implicitly share memory, processes implicitly never share resources. Threads can communicate by their shared-memory or by sending messages. Processes communicate over rigidly pre-defined interfaces, like streams, ports or sockets, by sending and receiving data. Refer also to the introduction on page 230.)

Process-controlling Java objects of class Process are created by the (factory-)method exec() of the java.lang.Runtime-class. Even if the Process-object is removed from the Java Runtime Environment, (for example by setting all references to it null and letting a call of the garbage-collector (System.gc();) erase that Process-object), then the process may continue its run on the native platform. The destruction of the control-object just denies the Java Runtime Environment any further access to the process.

As long as the Process-object exists, the process remains under control from the Java Runtime Environment, from where the process can be stopped or destroyed by calling the method destroy(). If the process returns an integer-exit-value, then that value can be obtained by calling exitValue() or by calling the method waitFor(). In the last case the current Java-thread can wait until the external process has finished and then that Java-thread can use the process's exit-integer. (The waitFor() -method gives the Process-equivalent of the Thread's join() -method.) The Java-thread can listen to the external process by reading data from Input- and OutputStream-objects¹⁰⁹ of the process: getInputStream(), getOutputStream(), getErrorStream(). The last one of the three streams usually prints its messages into a terminal-window of the native operating-system or fills a log-file.

The Compiler-Class The Compiler-class cannot be instantiated because it has no public constructors and has been declared final. This class usually acts completely independent from standard Java compilers! The java.lang.Compiler-class furnishes a non-trivial service only if (1) the system-property java.compiler is defined (for example name=jit (just-in-time)) and, (2) the method System.loadLibrary(name) loads the library with the name java.compiler. Then, the referenced compiler can transform classes in BYTE-CODE FORM INTO NATIVE SYSTEM-CODE. There are three methods that control the compilation: The method compileClass(Class c) compiles the class indicated by the Class-object. The method compileClasses(String s) compiles the classes indicated by the String-object. Finally, there can be found the method command(Object obj), where the argument and functionality depends on the compiler-library.

The ClassLoader-Class is an abstract class with a loadClass() -method. A ClassLoader-object can use the loadClass() -method to 1. take files or byte-streams from the file-system or a network and 2. if these bytes are of proper format, reinterpret them as classes and put these classes into their runtime-representation within the current Java Runtime Environment. This ClassLoader-class is described later beginning with page 433.

The SecurityManager-Class is an abstract class and thus has to be extended, if a Java-object of this type is to be created. And once an extension of this class has been instantiated and installed in the current Java Runtime Environment by calling System.setSecurityManager(), the installation cannot be changed during the life of the current Java Runtime Environment. The instance of the security-manager can be accessed by calling System.getSecurityManager() and calling its numerous instance-methods. The default implementations of the SecurityManager's instance-methods are maximally restrictive. A trivial extension produces a severe case of an autistic Java Runtime Environment after instantiation and installation:

```
class AutisticSystem extends java.lang.SecurityManager{
... System.setSecurityManager( new AutisticSystem() ) ...
```

No information will enter or leave that Java Runtime Environment. An extension that trivializes each method (setting the block empty {}) of the default security-manager produces a "FreeSystem":

```
class FreeSystem extends java.lang.SecurityManager
{ ...
  public void checkAccept( String host, int port ){ }; ...
}
... System.setSecurityManager( new FreeSystem() ) ...
```

¹⁰⁹For more on streams see the next chapter about communication.

free system =
no security-
manager

Every information can enter and leave that system freely as if there existed no security-manager! A list of all the `check...()`-methods of a security-manager can be found in the   Java Platform API Specification [JPAPIS]. For more on security-managers in Java see the notes beginning with page 462.

The Thread-Class and its extensions instantiate objects that can start and control threads. Thread-objects behave like software-dashboards of their associated threads. The Java source-code of the thread is given by `run()`-methods that are members of classes which implement the `Runnable`-interface. Since the `Thread`-class implements the `Runnable`-interface by default, an extension of the `Thread`-class may contain a definition of the `run()`-method, otherwise a `Thread`-object may delegate the code-definition to another object of type `Runnable`. Most of the functionality of `Thread`-objects and their methods has been described in the section about programming with threads.

The ThreadGroup-Class The `ThreadGroup`-class is used to instantiate objects that control sets of threads with a single command. Most details involve the handling of pure technicalities and are therefore omitted here.

The Throwable-Class This is the super-class of all those objects which can appear as arguments in a throw- or try-catch-clause in Java. These objects can be freely formatted to contain any messages that serve their cause. There exists a standard hierarchy of sub-classes rooted within the `Throwable`-class in Java. That hierarchy classifies errors and exceptions, but the classification does not furnish any additional functionality. The hierarchy is introduced on page 281.

The Boolean-Class, the Number-Class and its Extensions The `Boolean`-class and the extensions of the abstract `Number`-class `Byte`, `Short`, `Integer`, `Long`, `Float` and `Double` have instances which contain the same information as their counterpart of primitive-type. Since no information is lost, these objects are called wrappers of the variables of primitive-type. (Refer also to the wrapper description beginning on page 169.) What are these wrapper-classes good for? Remember that variables of primitive-type are used in Java because of their higher speed of access and processing, compared to variables of class-type; but variables of class-type furnish a greater flexibility and a wider range of possible operations. And remember that method-calls with arguments of primitive-type transfer these arguments by-value (as described around page 120). Methods with parameters having the type of a wrapper-class can process properly wrapped variables of primitive-type by-reference. — Furthermore, many (pre-written) methods and data-structures require arguments and variables of (sub-)class `Object` and at this point the wrapper-classes also become highly useful. For example an array of numbers can be defined as `int[] {1, 2, 3}`; but now only numbers may be inserted in the array, whereas an array of class `Object` can hold a wider variety of instances:

```
Object[] {new Integer(1), new Integer(2), new Integer(3), new String("maverick")};
```

The classes `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float` and `Double` cannot be extended because they are declared `final`. All these classes have a constant — a final static variable — named `TYPE` that returns a reference to the `Class`-object of that class:

```
Class classByte = Byte.TYPE; // the same as
Class classByte = byte.class; // !!!
```

Then each of these classes has a static method `valueOf(String s)`, which is sometimes overloaded. This method transforms string-information into data of the related primitive-type (a coding-transformation — a possibly destructive cast):

```
Boolean bool = new Boolean( Boolean.valueOf("true") );
```

Furthermore, the information contained in each of these variable-pairs of class-type and primitive-type (`Short` - `short`) can be shuffled back and forth — wrapped and unwrapped — by using the following methods (constructor and instance-method):

```
short s1 = 3; Short S1 = new Short(s1); short s2 = S1.shortValue();
```

The `hashCode()`-, `toString()`- and `equals()`-methods of the `Object`-class have been overridden and depend only on the data of the `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float` and `Double` objects. The `Number`-class is an abstract super-class of the `Byte`-, `Short`-, `Integer`-, `Long`-, `Float`- and `Double`-class. This class furnishes six abstract methods: `byte`-, `short`-, `int`-, `long`-, `float`-

and `doubleValue()`, which have been implemented in each class-extension. SO ANY OBJECT OF SUPER-CLASS-TYPE `Number` CAN BE TYPE-CAST TO FIVE ALTERNATIVE FORMATS. The `Byte`, `Short`, `Integer`, `Long`, `Float` and `Double` classes all have two public static final variables of the corresponding primitive-type, constants that are named `MIN_VALUE` and `MAX_VALUE` which contain the data that fit their name. (byte `MAX_VALUE` = 127 and byte `MIN_VALUE` = -128). The `Float`- and `Double`-classes additionally have been given three constants (final variables): `NaN`(=0.0/0.0), `NEGATIVE_INFINITY`(=-1.0/0.0) and `POSITIVE_INFINITY`(=1.0/0.0).

The Math-Class The class `java.lang.Math` is an aggregation of two constants and some mathematical functions. The constants are `E`(=2.7182818284590452354)¹¹⁰ and `PI`(=3.14159265358979323846)¹¹¹. Mathematical functions are those like `Math.random()` that returns a random number between 0.0 and 1.0 (uses the more versatile `java.util.Random`-class). Since the final-declared `Math`-class has no public constructors, it cannot be instantiated. An incomplete list of its (static) methods can be found on page 94.

The Character-Class The `Character`-class is an object-wrapper for values of primitive-type `char`. The `Character`-objects follow the Unicode-standard (page 34). The `Character`-class has many constants (final variables) and many class-methods, most of which are of the *issomething()*-pattern.

The String-Class Since the `String`-class is declared final, this class cannot be extended. Furthermore, a freshly allocated `String`-object can be assigned a value only once. Then that `String`-object remains a constant (An object of the `StringBuffer`-class makes it possible to freely manipulate its contents.). So, a `String`-object contains a constant sequence of characters, nevertheless its references still can be changed by assignment, as it is possible with every other object of class-type. The Java compiler (`javac`) has been designed to automatically compile string-literals to `String`-objects, this makes the following usage possible:

```
s1 = new String("abc"); s2 = "abc";
// s1==s2 or s1.equals(s2) are true since HERE contents are compared!
```

The string manipulating class-methods (`trim()`, `toLowerCase()`) of the class `String` produce *new* `String`-objects from old ones! Thus, these methods are factory-methods and the original `String`-object remains constant. The class-method `valueOf(any)` turns every value of primitive-type and every object's informational content into a newly created `String`-object. The instance-method `length()` returns the numbers of characters in the `String`-object. The factory-method `trim(String s)` returns the reference to another newly created `String`-object; that new object has been created to contain the string of the old object minus leading and trailing white-space and control-characters. There are many more methods that take a given `String`-object and return a different newly instantiated `String`-object to be stored.

The StringBuffer-Class The `StringBuffer`-class is the more flexible counterpart of the `String`-class. The `StringBuffer`-class has highly overloaded `append()`- and `insert()`-instance-methods. These methods manipulate the `StringBuffer`-object itself, unlike the factory-method of a `String`-object, they do not produce any new objects. The methods `charAt(int pos)`, `getChars(...)`, `length()` are also found in `StringBuffer`-objects (as well as in `String`-objects). The method `charAt()` returns the character at the position `pos` in the `String`-(Buffer), the first character has the position zero (chime-like counting). Use the instance-method `toString()` for converting a `StringBuffer`-object into a `String`-object. Otherwise use an appropriately chosen constructor (`String(...)` or `StringBuffer(...)`)

The Void-Class is an uninstantiable wrapper-class for the primitive-type `void`. It is just a reference to the Class-object `Void` in the context of the package `java.lang.reflect`.

¹¹⁰As could be seen on page 86, Java allows 17 digits past the decimal-point. The mathematical constant is called Eulerian number, a constant number, also written using the Latin letter "e". This number appears naturally when calculating a compound interest (or dynamical growth problem) problem with arbitrary small growth-intervals. Unfortunately this mathematical constant cannot be represented easily in the decimal number-system: Its exact description within the decimal number-system would have to use infinitely many digits: $e = 2.7182818284 \dots$. So when working explicitly with numbers in the decimal number-system only approximations are realistically usable, like $e \approx 2.72$ or the one in the parentheses above.

¹¹¹As could be seen on page 86, Java allows 17 digits past the decimal-point. The mathematical constant pi, a constant number, also written using the Greek letter "π", can be used to calculate the circumference l of a circle with the diameter d by using the formula $l = \pi \cdot d$. The area of a disk, the surface and volume of a sphere are also calculated using the constant π . Unfortunately this mathematical constant cannot be represented easily in the decimal number-system: Its exact description within the decimal number-system would have to use infinitely many digits: $\pi = 3.141592653589793238 \dots$. So when working explicitly with numbers in the decimal number-system only approximations are realistically usable, like $\pi \approx 3.14$ or the one in the parentheses above.

The Runnable-Interface requires an implemented `run()`-method and tags classes and objects telling the Java compiler and a fellow programmer: Inside this class can be found a method with the modifier, return-type and signature `public void run()`.

The Cloneable-Interface acts just as a tag to prevent the exception `CloneNotSupportedException` to be thrown, when the `clone()`-method of an instance of the `Object`-class is invoked.

For an in-detail coverage of these classes, interfaces and methods in this section consider the   Java Platform API Specification [JPAPIS] or the   Java Language Specification [JLS]. Some elementary aspects are presented below:

Platform and Runtime Environment

Platform and Runtime Environment — The classes `java.lang.System` and `java.lang.Runtime` predominantly furnish information about the current platform and the Java Runtime Environment (JRE). In the previous section, within the item “The System Class”, has been presented a piece of source-code for retrieving and storing system- and runtime-property information. This program has been extended below to show how such property-information might be changed:

```
public class MoreSystemProperties
{ public static void main( String unused[] )
  { Runtime r = Runtime.getRuntime();
    System.out.println
      ( "total Random Access Memory: " + r.totalMemory() +
        "\navailable bytes: " + r.freeMemory()
      );
    System.getProperties().list( System.out );
    // Above, first point evaluated first, rather parenthesize this!
    // java.util.Properties class extends java.util.Hashtable
    // and therefore can be listed to a PrintStream like System.out.
    java.util.Properties theProps = System.getProperties();
    theProps.put( "user.language", "ja" );
    System.setProperties( theProps );
    System.getProperties().list( System.out );
  } }
/*
```

```
[localhost:~/JavaCodeFiles/Basic1/SCPackaging] marycosway% javac MoreSystemProperties.java
[localhost:~/JavaCodeFiles/Basic1/SCPackaging] marycosway% java MoreSystemProperties
total Random Access Memory: 2031616
available bytes: 1839424
-- listing properties --
java.runtime.name=Java(TM) 2 Runtime Environment, Stand...
sun.boot.library.path=/System/Library/Frameworks/JavaVM.fra...
java.vm.version=1.3.0
gopherProxySet=false
java.vm.vendor=Apple Computer, Inc.
java.vendor.url=http://apple.com/
path.separator=:
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
java.vm.specification.name=Java Virtual Machine Specification
user.dir=/Users/marycosway/JavaCodeFiles/Basic1/Ag...
java.runtime.version=1.3
java.awt.graphicsenv=sun.awt.MacGraphicsEnvironment
os.arch=ppc
java.io.tmpdir=/tmp
line.separator=

java.vm.specification.vendor=Sun Microsystems Inc.
java.awt.fonts=
os.name=Mac OS X
java.library.path=./usr/lib:/usr/lib/java:/lib:/System...
java.specification.name=Java Platform API Specification
java.class.version=47.0
os.version=10.0
user.home=/Users/marycosway
user.timezone=
java.awt.printerjob=com.apple.mrj.internal.awt.MacPrinterJob
file.encoding=MacRoman
java.specification.version=1.3
user.name=marycosway
java.class.path=.
java.vm.specification.version=1.0
java.home=/System/Library/Frameworks/JavaVM.fra...
user.language=en
java.specification.vendor=Sun Microsystems Inc.
awt.toolkit=com.apple.mrj.internal.awt.VToolkit
java.vm.info=mixed mode
java.version=1.3.0
java.ext.dirs=/System/Library/Frameworks/JavaVM.fra...
sun.boot.class.path=/System/Library/Frameworks/JavaVM.fra...
java.vendor=Apple Computer, Inc.
file.separator=/
java.vendor.url.bug=http://developer.apple.com/java/
sun.cpu.endian=big
sun.io.unicode.encoding=UnicodeBig
mrj.version=3.0
user.region=
sun.cpu.isalist=
-- listing properties --
java.runtime.name=Java(TM) 2 Runtime Environment, Stand...
...
```

```

java.home=/System/Library/Frameworks/JavaVM.framework
user.language=ja
java.specification.vendor=Sun Microsystems Inc.
...
sun.cpu.isalist=
[localhost:~/JavaCodeFiles/Basics1/Aggregation] marycosway%
*/

```

System-Property: String-variables containing information about the platform and the Java Runtime Environment which is currently running. System-properties become especially important in security considerations on page 455.

The `user.dir`, `java.home` and `java.class.path` properties can be used later to get navigational bases for traversing the Hierarchical File System (HFS). The `user.dir.classpath` contains the standard base-directory which is used to extend relative directory-paths to absolute directory-paths. (see around page 47) — The class `java.io.File` organizes the access of files in the Hierarchical File System of the computer-platform.

```

class JavaAndHFS // (Hierarchical File System)
{ public static void main( String unused[] )
  { String path = new String
    ( System.getProperty("user.dir")
      + System.getProperty("file.separator") + "JavaAndHFS.java"
      // reconsider the output if the line above is omitted
    );
    java.io.File HFSObject = new java.io.File( path );
    System.out.println
    ( "path = " + path +
      "\nThe HFS-parent of the HFSObject is " +      HFSObject.getParent() +
      "\nThe name of the HFSObject is " +            HFSObject.getName() +
      "\nThe HFSObject a file? " +                   HFSObject.isFile() +
      "\nThe HFSObject is a directory/folder? " +     HFSObject.isDirectory() +
      "\nThe HFSObject is a path-collection? " +
        (!HFSObject.isFile() & !HFSObject.isDirectory()) +
      "\nIs the HFSObject as path absolute? " +      HFSObject.isAbsolute() +
      "\nThe HFSObject's length in bytes " +         HFSObject.length() +
      "\nLast modified in milliseconds since 1970: " + HFSObject.lastModified() +
      "\ncan be read " +                             HFSObject.canRead() +
      "\ncan be written " +                          HFSObject.canWrite()
    );
    try
    { String [] contentHFSObject = HFSObject.list();
      for( int i = 0; i < contentHFSObject.length; i++ )
      { System.out.println( contentHFSObject[i] ); }
    }
    catch( java.lang.NullPointerException e )
    { System.out.println("A file has no string contents"); }
    // exception catching could be avoided by using a simple if-clause
  } }

/*
[localhost:~/JavaCodeFiles/Basics1/SCPackaging] marycosway% javac JavaAndHFS.java
[localhost:~/JavaCodeFiles/Basics1/SCPackaging] marycosway% java JavaAndHFS
path= /Users/marycosway/JavaCodeFiles/Basics1/Aggregation/JavaAndHFS.java
The HFS-parent of the HFSObject is /Users/marycosway/JavaCodeFiles/Basics1/Aggregation
The name of the HFSObject is JavaAndHFS.java
The HFSObject a file? true
The HFSObject is a directory/folder? false
The HFSObject is a path-collection? false
Is the HFSObject as path absolute? true
The HFSObject's length in bytes 1921
Last modified in milliseconds since 1970: 992354783000
can be read true
can be written true
A file has no string contents
[localhost:~/JavaCodeFiles/Basics1/SCPackaging] marycosway%
*/

```

For more features of the class `java.io.File` refer to the   Java Platform API Specification [JPAPIS]. Objects of class `java.io.File` as well as the class itself have to be targeted in formulating access restrictions when enforcing security-measures (page 458). How to retrieve and store files in that file-system? Retrieving files is shown on page 298, storing files is shown next to page 298.

Internationalization - Localization

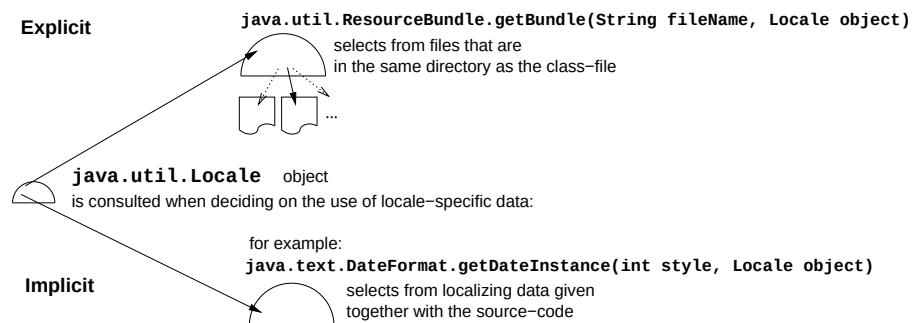
In Japan the sun in all its symbolic variations has a special meaning. In Japan and China lines of texts may be read from top to bottom of the page. Probably not only in China, voluminous manuals are often

discarded and personal advice is asked for, so the use of pictograms might attract more attention. In China the color red and the dragon have a special meaning. The Arabic world is used to read, write and conceive images from right to left (←). In Arabic countries, the color green generally makes a positive impression on people, the same perspective is found towards shadow. Therefore internationalization of user-interfaces may already enter the programming conception at an early stage. (more programming-related information starts on page 507 and can be found especially on page 513)

For a better reception of programs in different cultural and geographic settings, as well as in other localizable surroundings, programmers could try to adapt their code to different social contexts. Though, in many cases, the adaptation may often be far from thorough, even the preparation for these cases may be of significant help to the prospective users: Labels, messages that later can be easily adapted to the local language simplify the configuration and eventually the use of the program. Africans, Americans, Asians, Australians and Europeans will be happy to be able to write texts that incorporate their text-font peculiarities. Items which are affected by localization are for example:

- text-labels
- number-, date- and time-formats
- time-zones and time
- currencies

The basic operation reduces to that of a simple branching, much like in an `if-` or `switch`-statement: An object of class-type `java.util.ResourceBundle` supplements these branching statements. The equivalent to the boolean conditional expression in the `if-` or `switch`-statement is given by an object of class-type `java.util.Locale`. Many core-classes, so-called locale-sensitive classes, already are written to take `Locale`-objects and to do the branching *implicitly* (for example `java.text.DateFormat`).



Localization, as far as the Java language is concerned, introduces a branching mechanism with three differing entities to be identified:

(deciding instance) A `java.util.Locale`-object encapsulates labels for language, country, optional additional specifiers like tribal identity, network or operating-system. As an argument in locale-sensitive classes (for example some kind of "Formatting" classes) this `Locale`-object is referenced for triggering the switching between formats. A `Locale`-object gives the instance according to which switch-mechanisms decide upon the use of locale-specific data.

(the localized data) What about the locale-specific data, where is it stored? Locale-specific data can be either stored in objects of type `java.util.ResourceBundle` and its sub-classes. Or locale-specific data is given by text-files or class-files which are referenced by an object of class-type `ResourceBundle`. SIMPLE STRINGS MAY BE STORED IN A TEXT-FILE, OTHER LOCALE-SPECIFIC OBJECTS MAY BE STORED IN A CLASS-FILE.

(selector of the localized data) How is the localized data chosen? In the simplest case the locale-specific data may be selected with a simple `if-` or `switch`-statement. This may still cause the source-code to be scattered with locale-specific-data. Ideally, the source-code of an application should make it easy to introduce new locales even after its completion. Therefore, when possible, there the `Locale`-object is used by a `java.util.ResourceBundle`-object to select the appropriate file that contains localized data, usually in a hashtable-like format.

Identifying Locality with `java.util.Locale`-Objects

The `java.util.Locale`-class is an extension of the `java.lang.Object`-class. `Locale`-objects encapsulate up to three pieces of data:

Language code ISO-639, a string, coding a language, for example "en" for English (<http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>).

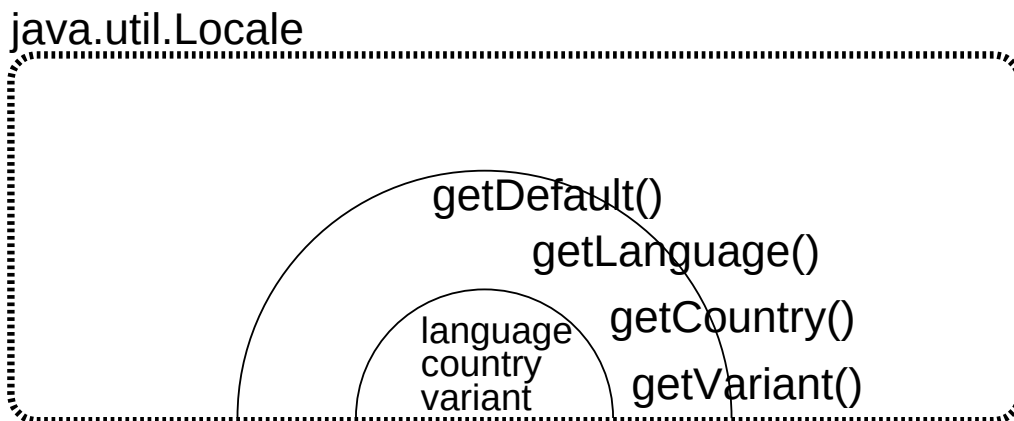
Country code ISO-3166, a string, coding a country, for example "GB" for Great Britain (<http://www.chemie.fu-berlin.de/diverse/doc/ISO-3166.html>).

Variant, a string, taking an option for specifying tribal identity, network or operating-system.

A selection of possible combinations and the Java-tag-format, excluding variants, can be seen in the following table:

ISO-639 language code	ISO-3166country code	tag	"locale"
en	US	en_US	United States of America
en	GB	en_GB	England and Scotland and Wales
de	CH	de_CH	Switzerland (German language)
fr	CH	fr_CH	Switzerland (French language)
it	CH	it_CH	Switzerland (Italian language)

The methods of a `Locale`-object can be referenced together with more information in the   Java Platform API Specification [JPAPIS].



Java source-code that introduces objects of the type `java.util.Locale` for deciding what localizing-data is used, appears in the next section(s). Objects of class-type `java.util.Locale`, for example, may determine the format of numbers (`java.text.NumberFormat`) or the format of a date (`java.text.DateFormat`).

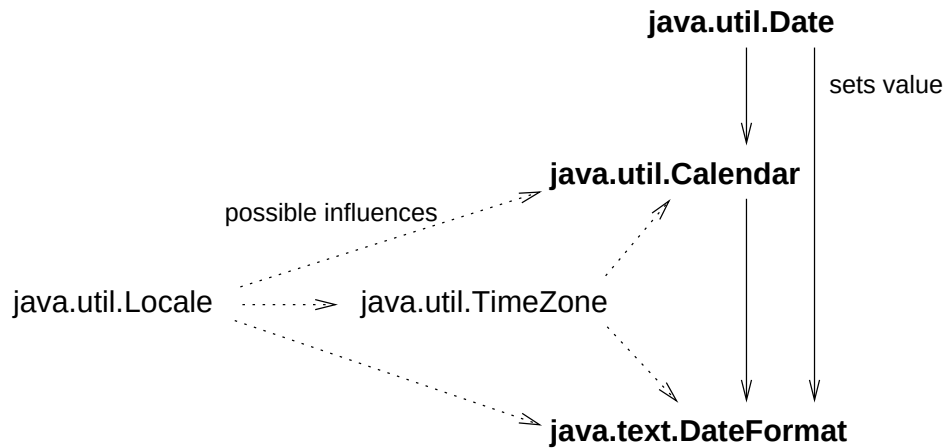
Example: Calendar and TimeZone

Measuring time accurately has been a challenge over centuries. Proper timekeeping was a necessity for such diverse activities as being able to spot the right time to harvest crop, to being able to accurately determining the longitude of a ship's position. Simple devices of timekeeping were a pendulum, the heart-beat, a sun's cycle (a day), a lunar cycle of appearance (a month) or the change of the year's seasons (a year). Today, atomic clocks give the world a time-frame independent from celestial or other cruder observations. Nevertheless, with greater precision and more conventionalities, like time-zones and culture-specific notations, the description of a point in time requires much more consideration. The documentation of the Java Application Programming Interface (API) gives further information about modern timekeeping, especially in the context of the class `java.util.Date`.

The current time on a computer is given by a public static long method

```
java.lang.System.currentTimeMillis()
```

that returns the milliseconds between the call of the method and midnight of January first 1970 UTC (Coordinated Universal Time).



The class `java.util.Date` encapsulates such a long integer for further interpretation (by classes like the following ones). The classes

`java.util.TimeZone` or `java.util.SimpleTimeZone` encapsulate a time-zone specification and associated manipulations like adjustments for daylight-savings-time. Time-zone identifiers can be found at <http://www.worldtimezone.com> or <http://tycho.usno.navy.mil>.

If needed, sub-classes of the abstract class `java.util.Calendar`, especially the fully-implemented class `java.util.GregorianCalendar`, transform the long integer of a `Date`-object into YEAR, MONTH, DAY, HOUR integer-fields. Although the class `java.util.Calendar` has been declared abstract, the class furnishes static `getInstance()`-methods, which allow to produce an instance for using its basic functionalities, as is demonstrated in the source-code below.

To get a human readable form use the class `java.text.DateFormat` which formats the previous information to a conventionally readable date according to `Date`, `Calendar`, `TimeZone`, and `Locale` informations.

```

public class Time
{ public static void main( String unused[] )
  {
    java.util.Date theTimePoint = new java.util.Date();
    System.out.println( theTimePoint.getTime() );
                                // milliseconds since 00:00:00 Jan 1 1970 UTC

    java.text.DateFormat formatter = java.text.DateFormat.getDateTimeInstance
    ( java.text.DateFormat.FULL, // date-style integer
      java.text.DateFormat.FULL, // time-style integer
      java.util.Locale.FRENCH    // this makes DateFormat a locale sensitive class
    );
    String s3, s2, s1 = formatter.format( theTimePoint );
    System.out.println( s1 );
    java.util.TimeZone tz = java.util.TimeZone.getTimeZone( "PST" );
    formatter.setTimeZone( tz );
    s2 = formatter.format( theTimePoint );
    System.out.println( s2 );
    java.text.DateFormat formatter2 = java.text.DateFormat.getDateTimeInstance();
    java.util.Calendar yearMonthDay = java.util.Calendar.getInstance
    ( java.util.TimeZone.getTimeZone( "EST" ),
      new java.util.Locale( "en", "US" )
    );
    formatter2.setCalendar( yearMonthDay );
    s3 = formatter2.format( theTimePoint );
    System.out.println(s3);
  } }
// java.util.Calendar is abstract but has factory-methods. WHY?
// java.util.TimeZone is abstract but also has factory-methods. WHY?
// java.text.DateFormat is abstract but has factory-methods WHY?
// The use of the unextended class is thus signaled to be disadvised.
//
// The original DateFormat settings of Locale override Calendar Locale settings,
// this can be tested by using formatter in producing string s3

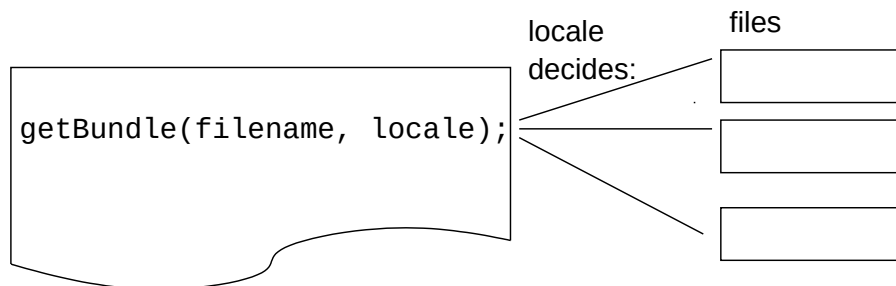
/*
[localhost:~/JavaCodeFiles/Basics1/SCPackaging] johnharrison% javac Time.java
[localhost:~/JavaCodeFiles/Basics1/SCPackaging] johnharrison% java Time
992361056704
mardi 12 juin 2001 17 h 50 CEST
mardi 12 juin 2001 08 h 50 PDT
  
```

```
Jun 12, 2001 11:50:56 AM
[localhost:~/JavaCodeFiles/Basics1/SCPackaging] johnharrison%
*/
```

Locale-sensitive classes, like the `Calendar`-class, generally have a `getAvailableLocales()`-method returning an identifier of array-type. That array refers to all `Locale`-objects present for that class (available with Java Versions 1.2 and larger). In Java versions before 1.2 the classes `Date` and `GregorianCalendar` sometimes did produce unreliable results.

Example: Resource-Bundle

A simple case of a resource-bundle occurs with window-objects of a Graphical User Interface (GUI) like buttons (for an elementary example see page 544); depending on a chosen locale (given by an object of class-type `java.util.Locale`), such buttons may be labeled “Save” or “Sauver” in French or “Sichern” in German. Instead of re-writing the program with the appropriate strings for each language, or instead of devising a `switch`-statement with each window-object, the window-object refers, locale-dependent, to an external file which contains the translated labels. A `java.util.Locale`-object can be used to decide like a (non-cascading: “breaks” inserted) `switch`-statement between files that contain the relevant information. The content of these files is organized like a hashtable (key and entry pairs).



The class `java.util.ResourceBundle` and objects of this class-type manage the selection-process for a given `java.util.Locale`-object as can be seen in the source-codes below. This turns the entire class `java.util.ResourceBundle` into a kind of a database with locale-specific objects for multiple locales, either loaded from individual files or from a class. Here the file-version:

```
public class LocalHelloWorld
{ public static void main( String unused[] )
{ // the language locale specification
  java.util.Locale english, french;
  french = java.util.Locale.FRANCE; // more constants see java.util.Locale documentation
  english = new java.util.Locale("en",""); // "" or the country code "US" or "UK"
  // loading the data
  java.util.ResourceBundle dataEnglish, dataFrench;
  try
  { dataFrench = java.util.ResourceBundle.getBundle( "theData", french );
    dataEnglish = java.util.ResourceBundle.getBundle( "theData", english );
  } // using the data
  { System.out.println( dataFrench.getString( "Hi" ) + "\n" +
                        dataFrench.getString( "id" ) );
    System.out.println( dataEnglish.getString( "Hi" ) + "\n" +
                        dataEnglish.getString( "id" ) );
  } catch( java.util.MissingResourceException e )
  { System.out.println
    ( "Resources for localization not available, default: Hello World!" ); }
} }
/*
--- The data for localization has to be given in text-files in an assignment-like format.
--- (PREREQUISITE) Part of the names of the text-files has been given above 'theData';
    underline-suffixes enable the Java Runtime Environment to choose the proper file.
    ('theData.properties' and 'theData_fr.properties')
--- The text-files have to be found within the same directory as the byte-code
    of this program.
--- The try-catch clause may be omitted, thereby sacrificing the exception-handler.
*/

/*
[localhost:~/JavaCodeFiles/Basics1/SCPackaging] marycosway% javac LocalHelloWorld.java
[localhost:~/JavaCodeFiles/Basics1/SCPackaging] marycosway% java LocalHelloWorld
Bonjour Monde,
```

```

je suis un programme de Java.
HelloWorld,
this is a Java program.
[localhost:~/JavaCodeFiles/Basics1/SCPackaging] marycosway%
*/

```

Here the content of the file named “theData.properties”:

```

# comment
# a file named "theData.properties"
Hi = HelloWorld,
id = this is a Java program.

```

Here the content of the file named “theData_fr.properties”:

```

# comment
# a file named "theData_fr.properties"
Hi = Bonjour Monde,
id = je suis un programme de Java.

```

Above the files have been pure text-files following a simple convention of identifying strings by names and assignment symbols. Comments could be inserted when following a sharp (#) symbol. In the following case the FILES THAT CONTAIN THE DATA HAVE TO BE COMPILED; therefore they may contain entire Java objects.

```

public class LocalHelloWorldList
{ public static void main( String unused[] )
  { // the language specification
    java.util.Locale english, french;
    french = java.util.Locale.FRANCE; // more constants see java.util.Locale docs.
    english = new java.util.Locale("en","US"); // the country code "US" or "UK" or ""
    // loading the data
    java.util.ResourceBundle dataE, dataF, dataG;
    dataE = java.util.ResourceBundle.getBundle( "Data", english );
    dataF = java.util.ResourceBundle.getBundle( "Data", french );
    dataG = java.util.ResourceBundle.getBundle( "Data", java.util.Locale.GERMAN );
    // using the data
    System.out.println
      ( dataF.getString( "Hello" ) + "\n" +
        dataF.getString( "id" ) + dataF.getObject( "number" ) );
    System.out.println( dataE.getString( "Hello" ) + "\n" + dataE.getString( "id" ) );
    System.out.println( dataG.getString( "Hello" ) + "\n" + dataG.getString( "id" ) );
  } }
/*
--- Without the resource-files, the method getBundle() throws a
    java.util.MissingResourceException.
--- (PREREQUISITE) Before running this program compile the code contained in
    the files named 'Data_en_US.java', 'Data_fr_FR.java' and 'Data_de.java'.
--- Let the byte-code (in class-files) be found in the same folder.
*/

```

Here the content of the file named “Data_fr_FR.java”:

```

// content of the file named 'Data_fr_FR.java'
// to be compiled before usable!
public class Data_fr_FR extends java.util.ListResourceBundle
{
  private Object [][] contents =
  {
    { "Hello", "Bonjour Monde"},
    { "id", "ici parle Java, avec un object: " },
    { "number", new Integer(1) }
  };
  public Object [][] getContents(){ return contents; }
}

```

Here the content of the file named “Data_en_US.java”:

```

// content of the file named 'Data_en_US.java'
// to be compiled before usable!
public class Data_en_US extends java.util.ListResourceBundle
{
  private Object [][] contents =

```

```

    {
        {"Hello", "Hello World"},
        {"id", "this is Java speaking!"}
    };
    public Object [][] getContents(){ return contents; }
}

```

Here the content of the file named “Data_de.java”:

```

// content of the file named 'Data_de.java'
// to be compiled before usable!
public class Data_de extends java.util.ListResourceBundle
{
    private Object [][] contents =
    {
        {"Hello", "Hallo Welt"},
        {"id", "hier spricht Java!"}
    };
    public Object [][] getContents(){ return contents; }
}

```

These were only simple cases, the online-documentation has a lot of additional information in store.

Character Codes and Fonts

Character Codes and Fonts — Although images, audio- and video-files may be produced, stored, and viewed on a computer (see page 527), much of the information that is stored on a computer or exchanged between computers is coded as letters (like this book, like any (Java-)program). Remember that characters and other symbols within a computer-system can be stored and retrieved in binary form by mapping the set of characters and symbols into a set of bit-sequences. Such a mapping, called a (coding-)format, has been introduced beginning with page 33. There also has been mentioned that the graphical representation of such a symbol, used for output purposes usually has to be stored separately as an image-file. Such image-information, to depict symbols and individual characters, has been called font or glyph.

The mapping of bytes to letters can be done by using the ASCII-format, which codes one symbol per byte (8bits); or the mapping may be given by the Unicode-format, which codes one symbol using two bytes. The ASCII-format can be seen as a subset of the Unicode-format, but there are other variants; for example the widely used ISO-Latin-1-format. See also <ftp://ftp.isi.edu/in-notes/iana/assignments/mediatypes/media-types> or search in the Internet for “data-format” if <http://www.kostis.net> doesn't work.

Encoding-schemes used in different parts of the world are:

Western Europe ISO-8859-1 or MacRoman or Latin-1

Central Europe ISO-8859-2 or MacCE or Windows-1250

Japanese Shift_Jis or EUC-JP

Traditional Chinese: Big5 or EUC-TW

Simplified Chinese: GB2312

Cyrillic ISO-8859-5 or MacCyrillic or Windows-1251 or KOI8-R

Greek ISO-8859-7 or MacGreek or Windows-1253

Turkish ISO-8859-9 or MacTurkish

Unicode A two-byte character-encoding format. Also defined by ISO-10646. Unicode Transformation Formats (UTF), especially UTF-8, give variants of the Unicode-format which allow to store Unicode-coded files more efficiently. For more, consult the book *The Unicode Standard, Version 3.0*, (ISBN 0-201-61633-5) Section 3.8; this book is also downloadable in the PDF-format from <http://www.unicode.org>.

UCS (Unified Coding Standard) ISO-10646. A four-byte character-encoding format, a super-set of the Unicode-format.

The difference between encodings/formats (possibly platform-specific) and content-types (portable and application-oriented) is described further on page 368.

Java: Encodings and Fonts

Java: Encodings and Fonts — Java uses platform-dependent fonts. Java 1.0 can only display a subset (ISO-Latin-1) of the Unicode-character-set. Java 1.1 can display the full Unicode-character-set, *provided* the necessary fonts are installed on the platform, which was usually not the case. Why considering text encoding-formats anyway? Later, with text-streams, on page 303, the names of the encodings are used.

Answers to the question “How to install font/glyph-files into a computer-platform?” are not given at this point. For this consider the instructions of the   Java Standard Edition Documentation [JSDK] which can be found in the Internationalization-Fonts section.

The question “How to list the Java-available fonts of the current Runtime Environment?” is answered by the following piece of source-code. This code anticipates the use of a class from the `java.awt` package: `java.awt.GraphicsEnvironment`. The object, that gives information about the graphics-capabilities of the current Java Runtime Environment, is given by using a factory-like-method¹¹²:

```
import java.awt.GraphicsEnvironment;
class PlatformFonts
{ public static void main( String unused[] )
  {
    GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
    String [] fontList = ge.getAvailableFontFamilyNames();
    for( int i = 0; i < fontList.length; i++ )
    { System.out.print( fontList[i] + " -" );
    } } }

/*
C:\WINDOWS\JavaCodeFiles\Basics1\SCPackaging>javac PlatformFonts.java

C:\WINDOWS\JavaCodeFiles\Basics1\SCPackaging>java PlatformFonts
Arial -Arial Black -Comic Sans MS -Courier New -Default -Dialog -DialogInput -Im
pact -Lucida Console -Lucida Sans -Marlett -Monospaced -SansSerif -Serif -Symbol
-Tahoma -Times New Roman -Verdana -Webdings -Wingdings -
C:\WINDOWS\JavaCodeFiles\Basics1\SCPackaging>
*/
```

How these fonts can be actually put to use is demonstrated on page 569, where font-bearing texts are produced on-screen. The program above only informs about the availability of fonts within the Java Runtime Environment! This concludes the brief introduction to a selection of core-classes of the Java Runtime System (JRS).

¹¹²Remember that *factory-methods* create an object and return a reference to it. The initial type of the object is given by a non-abstract class. *Factory-like-methods* return references of mostly abstract class-type, these references mostly point to native-code. (also refer to page 185)

Part IV.

Basic Programming II Internal Communication

Basic Programming II, Internal Communication — Proceduralization, object-orientation, multithreading, aggregation and packaging have been introduced so far as means of structuring code. These features of a programming-language are offered to programmers and, if properly used, these features can ease programming significantly. (Of course the wealth of these features increases the amount of initial learning for the programmer.) The next part will introduce other methodologies of structuring code, especially with respect to communication between parts of source-code. Those communication-concepts rely on the concept of a method-call, but they surpass the idea of a simple method-call at the same time: Method-calls are used for registering listener-objects, for dispatching event-objects, for reading out bytes of streams, in that sense method-calls become part of an entire standardized communication-context, thus hopefully increasing the flexibility of programming. The three core-concepts, to be introduced, are messaging, streaming and component-based software.

Rather elementary examples of communication-aspects already have been given by some pieces of source-code introduced so-far: Consider the callback example on page 127, where the communication-process is given by two methods mutually calling each other. Or consider the producer-buffer-consumer threaded transfer of information presented on page 228. In these examples method-calls are used to implement rather unspecific, interaction-oriented functions (registering, message sending, organizing the handling of messages). This idea produces the surprising wealth of possibilities connected to concepts like messaging, streaming and component-based software.

Java: Communication: Introducing Classes across Files of Source-Code

Java: Communication: Introducing Classes across Files of Source-Code — Consider two classes defined in different files of source-code. Assume that both classes interact only on a restricted level of knowledge of each other, then their interaction can be made a loose one by encapsulating the gist of interactive functionality into a set of Java-interfaces which have to be available and implemented by both classes. One example of such a communication-practice can be seen in the demonstration of remote method calling on page 414; the first of the two graphical representations explicitly displays the two occurrences of the same interface; each piece of source-code implements the common interface.

The case of communication between classes or objects by implementing the same Java interface can be reformulated: Without using a network- or process-infrastructure, imagine the case where all the classes and objects are within the same Java Runtime Environment (JRE): The classes and objects may interact only using the common knowledge given by their mutually implemented interface. (In this context the latter Java-construct (interface) visibly acts according to its name!) An example, for the previously described case, can be seen on page 404, there the “Receiver”-interfaces allow an effective communication of otherwise independent pieces of source-code.

This approach to communication goes along the direction of making information-exchange independent from knowing individually defined method-signatures of the targeted objects. This uncoupling of the communicative instances may lead to concepts like messaging, streaming, protocols and more.

Communication: Messaging and Streaming

Communication: Messaging and Streaming — Computing also means sending data in the form of bits and bytes from one place to another: On the level of the processor, this means to transfer bytes from the Random Access Memory to the processor's stack for further manipulation, or to exchange bytes between the hard-drive and the computer's Random Access Memory (RAM), or to fetch bytes from address-space (port) associated with keyboard, mouse, telephone- or data-lines, or to put bytes into address-space (port) associated with screen-output, printer-output, telephone- or data-lines.

On the user-level, these actions could translate to initiating a spreadsheet calculation by a mouse-click, to load a program from the hard-disk. These processes can be seen as more or less complex acts of a (technical) communication-procedure: In the first case the processor and the RAM communicate over the CPU's bus. In the second case the RAM communicates (more or less processor-driven) with the hard-disk over the external bus extension.

Here has been tried to describe different forms of computer-related communication-processes by a relatively simple model. And then to use this model as a guiding-light into related, Java-specific programming features.

In programming-languages, especially object-oriented ones, one of the simpler means of communication between pieces of sources-code can be realized by a method-call. The finishing of a calculation can be communicated by a method-call (`System.out.println()`, `java.util.Vector.add()`). This requires the caller to know the called method's signature; this means the caller and the called pieces of source-code have to share a minimum of common knowledge.

This minimum of common knowledge can be further reduced by using Java-interfaces, as indicated in the previous section.

Another approach strives to externalize the minimum of common knowledge into MUTUALLY INDEPENDENT SPECIFICATIONS OF EXCHANGING INFORMATION, namely: Messaging

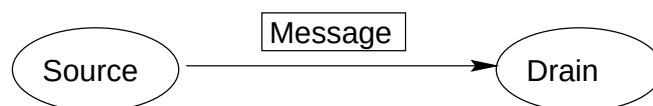
Concept: Basic Communication-Models

Concept: Basic Communication-Models — This section compiles three basic models of communication that are partly interrelated. The basic **message-model** can be varied into a **stream-model** for considering data-transfer within fixed channels. The same basic message-model can be extended into the **protocol-model** by introducing the time-parameter and bidirectional message-exchange.

Message-Model

Message-Model — Consider clearly separated entities which can communicate only over an external mechanism that condenses their common knowledge on communicating with each other. To simplify the concept, consider a one-way-communication because the two-way communication can be implemented easily by considering two counter-directed one-way-communication settings.

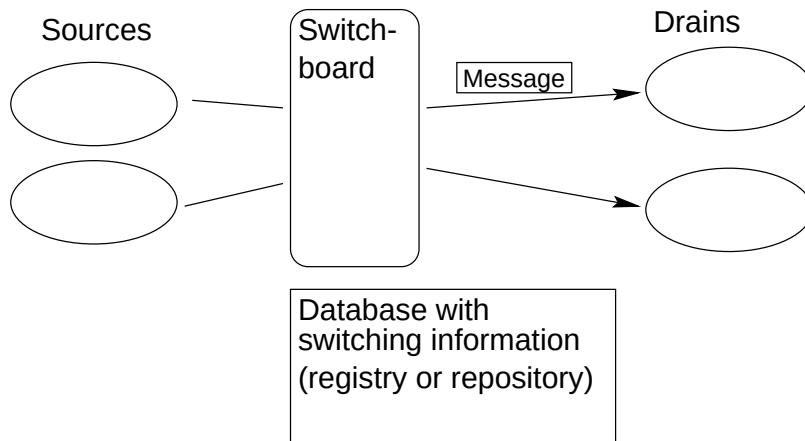
Let their common knowledge about one-way-communication allow one entity, the **source** to send information to another entity, the **drain** (=sink). Information that can be coded by one side (source), so that the other side (drain) can decode and interpret it properly, is called **message**. Thus the notion message indicates the existence of the minimal common knowledge that allows communication.



In short: A source sends a message to the drain. In computing, the raw message may just be a sequence of bytes moved from one place to another.

If there are multiple sources or drains, then there may be a **switchboard** managing the routing of the message.

switchboard



The basic questions to ask with the basic communication-model are "Who sends?", "What is sent?", "Who catches?" and "Who switches?"

Source sends message (alternatively named **observable**, **dispatcher**)

Message contains information (alternatively named **event-message**, **stream**)

Switchboard routes message to its destination (alternatively named "adapter" objects/classes, broker objects/classes and much more)

Drain receives the message and handles or processes its content (alternatively named **observers**, **destinations**, **targets**, **listeners** or **handlers**)

A **trigger** causes the message to be sent (introduced later with protocols)

Messages **Message** can be classified:

event-
messages

Messages that are used for *purposes of control* are called **event-messages**. Mouse-clicks, key-presses, also called events, produce event-messages. (Often event-messages are just called events.) Usually event-messages are very short sequences of bytes, a few hundred or less.

streams

Messages that mainly *transfer content or data* are also called **(data-) streams**. Streams generally code texts, pictures and other user-related data. Most streams are (much) longer than a few hundred bytes and usually cannot tell their size in advance (for example video- or voice-streams).

packets or
frames

Messages that contain bytes of *message-specific control-information* (mostly the address(es) of their drain(s) or other addresses and checksums) together with bytes of data are also called **packets** or **frames**.

Often event-messages are a specific piece of high-priority information from outside the CPU. The registering of event-messages usually has a high priority, therefore many event-messages preempt the CPU. Beside that, event-messages can be seen as streams of highly predefined content. Examples for events are a keyboard-input or a mouse-click. The basic concept of an event-model later is refined, especially on page 278 and following the page 571.

A differing approach to routing (event-)objects may be chosen by separating the switching-process from its information-store. Then this information-service, used by the switchboard, to possibly retrieve more information about availability of sources or drains, is called **repository** or **registry**. (For a hint on a concrete RMI-example, see page 414.) In special cases, the switching-process may even be controlled from the sources or the drains (For an example, see the graphics concerning Java Beans beginning with page 314.).

Stream-Model

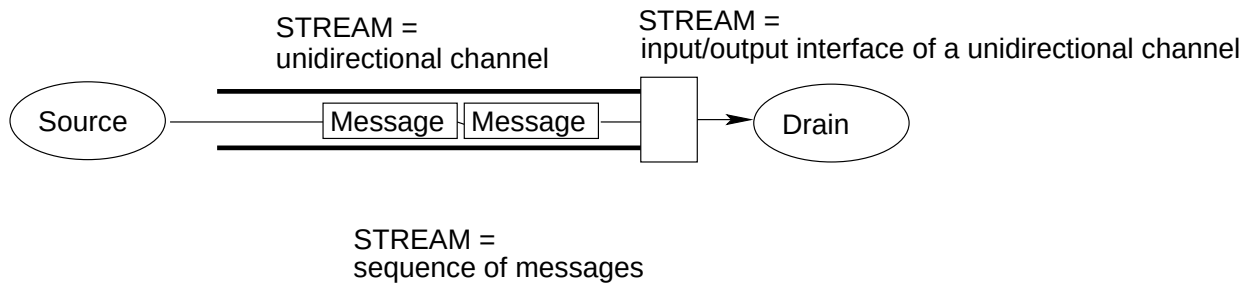
stream

Stream-Model — A reserved channel for sending or receiving data-streams is called stream or sometimes pipe. A programming interface connected to such a channel, so that this interface acts as the source for input and output, is also called **stream**. Even data received or sent without knowing the end of the data

in advance may be called a stream (of data).

Stream: Observe that the notion "stream" can stand for a

1. **flow** of data
2. **channel** of communication or
3. **source** of input **or drain** of output that does buffering and formatting



Operating-System/Runtime System: Sockets for Input and Output

Operating-System/Runtime System: Sockets for Input and Output — When a computer is connected physically to a network, it still needs a piece of basic software (driver-software, part of the operating-system) that controls and manages the connection to the network and thus establishes a bidirectional connection to another computer; such a piece of software is called a **socket**. After a socket has established the network-connection, it usually implements an input-stream and an output-stream for the data (Stream-definition Number 3 above).

socket for communication with another computer

Input/Output-Streams

Socket

Socket 1. A piece of software, furnished by the operating-system on hardware-level, enabling the programmer to establish a bidirectional data-link to an otherwise remote resource (network, disk-drive). Sockets usually help to establish a remote connection between different CPUs, especially within a network. Sockets transform data of the computer's unidirectional → hardware-ports into a programmer-manageable form and vice-versa. Data exchanged between remote CPUs is received or sent as streams over sockets. Sockets give an application-programmer access to **other** computers in a network by furnishing two streams (= source of input or drain for output). Whereas (unidirectionally working) ports (page 231) usually give an application-programmer access to other processes that run on the **same** computer.

Sockets are pieces of software on system-level that handle the Input/Output of external, usually networked, communication. Sockets support networking-protocols, this is exemplified by the class `java.net.Socket` on page 384 which represents a TCP-socket in the Java Runtime Environment. (TCP abbreviates Transmission Control Protocol, see page 363) — In Unix-based operating-systems so-called Remote Procedure Calls (RPC) (Java Runtime Environment: Remote Method Invocation (RMI), introduced on page 414) also transfer information using sockets.

2. A socket, as a functional entity of the native operating-system, connects to and disconnects from a remote CPU, sends and receives data over an established connection to a remote CPU. The Java implementation of the TCP-protocol is given by the class `java.net.Socket` and presented on page 384; there can be seen that sockets cannot be automatically classified into client- or server-sockets, sockets may also service peer-to-peer connections!

3. A server-socket always accepts connections from clients and usually bind them to a TCP-port (page 363). A server-socket listens for incoming communication requests. Sending and receiving data over an established connection to a remote CPU is done by using a regular socket! Thus in a Java Runtime Environment, objects of class-type `java.net.ServerSocket` accept client-requests, but delegate the process of information-exchange to an object derived from the class `java.net.Socket`, see page 385 and the graphic on page 421.

4. TCP-sockets denote something completely different: The aggregation of an Internet-address and a TCP-port is also called TCP-socket, this notion has been introduced on page 363.

Computer-communication in programming-languages is usually done using so-called "socket"-constructs: Picturesquely speaking: In programming-languages, sockets are the telephone-dial into the computer-network, whereas streams are the microphone (output-stream) and loudspeaker (input-stream) of the telephone's receiver. The Socket-methods take the signaling part, whereas stream-instantiating factory-methods of the socket furnish the means for data-exchange. Therefore often, sockets can be classified as

Sockets signal, Streams send data

Client-, Server- and Peer-to-Peer Sockets and each of those sockets furnishes an input- and output-stream. (“SOCKETS SIGNAL, STREAMS SEND DATA”)¹¹³

For a collection of the elementary different means of communicating with the Java Runtime Environment, consider streams and triggerable-methods, introduced on page 297 (the `System.out`-stream-object for printing a `String`-object onto the screen in the “HelloWorld”-program on page 71) and on page 127 (the “java”-command for starting a Java (application)-program and its Java Runtime Environment).

Protocol-Model

Protocol-Model — Essentially, protocols gather all that information on how the sender can communicate with the receiver and vice-versa, so that the communication process can be described independently from individual senders and receivers. As one of the more crucial aspects consider the time-dependency of interaction, described by protocols. An elementary example of a protocol is given when considering two simple cases of triggering the dispatch of a message: When the source triggers the sending of the message this is called **PUSH-case**. But there are cases where the drain triggers the sending of the message, called the **PULL-case**. Any Internet-browser, with which information can be pulled from a server by entering the Internet-address, constitutes an example of a PULL-case: Entering an Internet domain-name `java.sun.com` into the address-field of an Internet-browser triggers information-retrieval. Or for example, getting mail by an email-client triggers a mail-server to send the emails that have been received so far, a PULL-case again. On page 228 the PULL-model has been implemented; a call of a `getMessage()`-method fetched an `Integer`-object. The PULL-concept is used also in Java image-management; the Java Abstract Window Toolkit (AWT) Java2D uses a pull-construct, a “drain-triggers-loading”-process, reoccurring regularly when receiving data over a network. PULL-mechanisms are often associated with asynchrony and waiting,

whereas the PUSH-concept can be associated more with synchrony and real-time interruption. Compare this also to thread-communication.

Basic Possibilities for Triggering the Message Dispatch

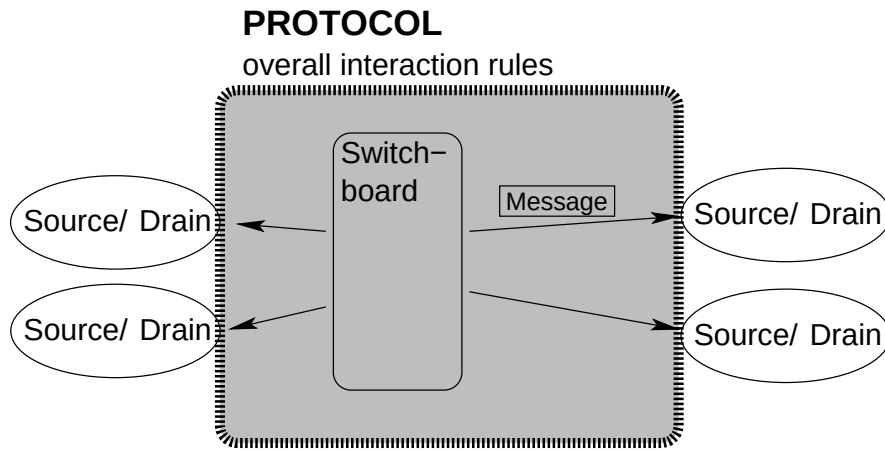


The same in a diagram with time-dependence:



The PULL-case of communication represents a simple example which comes close to the characteristics of a protocol: Bidirectional communication involving a scheme of recurring question- and answer-exchange to produce a result. Bidirectional communication alone already automatically involves a time-dependent relation of information-exchange:

¹¹³The equivalent in multithreading can be formulated as follows: “Thread-objects control, objects of interface-type `Runnable` run.”



A bidirectional connection often just switches the roles of drain and source (like speaking over a standard telephone line). Sometimes the messages can be exchanged over two unidirectional channels (like the ports in multiprocessing). There are some overall aspects of bi- or unidirectional connections:

The format or type of messages AND additional related information, for example how to initiate a connection, how to extract data from multiple packets or how to answer a request, is called **protocol** of the communication-process¹¹⁴. The protocol represents the entire knowledge of source and drain that enables them to communicate sensibly.

If the protocol lets the drain reconstruct data- and control structures from synthesizing multiple messages, then the protocol is called a **packet-oriented protocol**. If the protocol requires that the communication be initiated and terminated by some control messages, then the protocol is called a **connection-oriented protocol**¹¹⁵.

Protocol: A protocol collects all the rules that are necessary to establish a functioning communication. A protocol is a part of an → interface that describes a communication-process: The protocol-part of an interface describes the time-dependent exchange of messages during the process of communication.

For more on protocols consider the next paragraphs and/or follow the text on page 347.

Initial Triggering

Initial Triggering — Consider two instances of a communication-setting: If both sides can initiate the communication to mutually exchange control-messages (event) and data (stream), then the communication is called a **peer-to-peer communication**.

If only one side can **initiate** the communication-process, then the other side is only waiting for incoming communication-requests. When the initiator, also called the **client**, has established a connection, then both sides can freely exchange control-messages and data. A communication-partner, which waits for one or more clients or initiators to establish a communication, is also called **server**. Much of the communication between computers has this type of unbalanced **initiation-structure**. Such a kind of communication is also called **client/server-communication**.



A server waits for events that trigger actions, often these actions include sending streams to the client. Thus communicating computers can be distinguished: **CLIENTS INITIATE THE COMMUNICATION. SERVERS**

¹¹⁴Communicating with another person by mail may illustrate the notions event, stream and packet: A letter within an envelope contains *data* written on the folded and enclosed letter-paper. The address and stamp on the envelope act as letter-specific *control-structures* without which the letter would not reach its destination. Such a letter can be seen as a "real-life *packet*". If you live near the recipient of your letter, you may drop the letter in his/hers mailbox maybe announcing other letters to follow, sending a "real-life *stream*" of data. If you ring the recipient's phone to tell him to look for the letter in the mailbox, you control the other's actions and have a "real-life *event*". The knowledge that the recipient checks his/hers mailbox from time to time, and that you write the letter in a language the recipient understands, is part of the "real-life *protocol*" that you follow.

¹¹⁵Some people follow a packet-oriented protocol by sending you each year a letter with pictures of their families and a letter that summarizes the past year. Other follow a more connection-oriented protocol by asking you if you like to hear about their family-history and if you accept, they may sent you a little book or they send you chapters of it (packet-oriented), after you have received it, you may exchange comments on it.

ARE LISTENING CONSTANTLY FOR COMMUNICATION-REQUESTS, a typical PULL setting. (For an introductory example remember the source-code on page 228.) Peer-to-Peer and Client-Server concepts can be found also in computer-networks, as indicated on page 352.

Server: A **communication-partner** unable to initiate a communication-process, but waits for the initiation by some other communication-partner. After the initiation a server may not be inhibited in any way to communicate with its initiating-partner, basically it can send and receive data freely.

1. A **software** that waits for requests and answers those requests. The software that issues requests is called → client-(software). Server- and client-software may reside on the same computer or may reside on different computers that are part of a network.

2. A **computer** on which server-software is running.

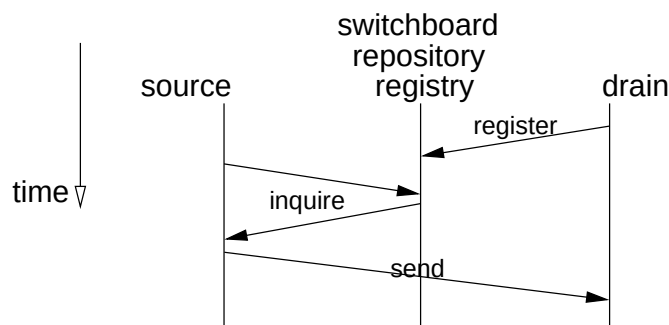
Clients initiate a communication process, servers wait for a communication process to be initiated.

Client: A **communication-partner** that initiates a communication-process, but cannot respond to other initiations. Those communication-partners with which the client can initiate communication are the servers. After the contact has been established by the client, both sides, ideally, can exchange data bidirectionally, free from any restrictions.

1. A networking **software** initiating contact with another software, which itself waits for requests and that answers those requests (called → server-(software)). The name “client” only describes how the contact to another software is made, this implies no information about how or in what direction the bulk of data is transferred. Client- and server-software may reside on the same computer or may reside on different computers that are part of a network.

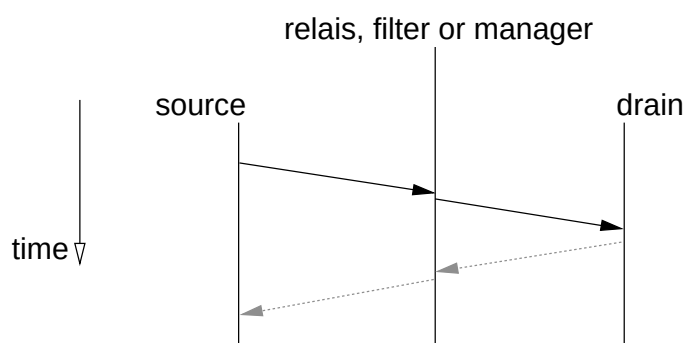
2. A **computer** on which client-software is running.

Source-event-drain implementations can use different ways to establish and characterize switchboarding-structures. Another example of the idea of a protocol represents a switchboarding-infrastructure which allows entities to register themselves as drains and then to receive messages:



The above can be reformulated as follows: Event-source and event-drain have to be acquainted by a procedure in which the drain registers itself as a listener for potential sources. This initiation protocol-structure is also known under the name **Delegation Event-Model** or **subscriber event-model** or **callback-model**. Sometimes source and switchboard are identical. This model actually is used in Java: Beginning with Java 1.1, the Abstract Window Toolkit (AWT) organizes event-handling with the delegation event-model (page 573). And the framework around the class `java.util.EventObject` uses methods like `addListener()` and `removeListener()` to establish connections to drains (page 286). The Java-Beans convention uses this model too, see page 314.

Another simpler conceptual aspect of a protocol between three entities is given below, it describes the workings of a kind of relay or filter:

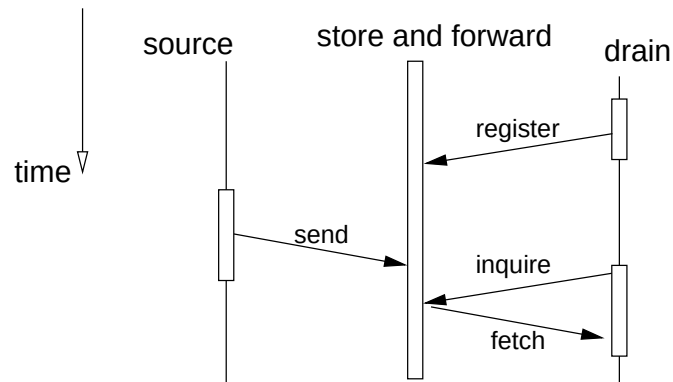


If considering the *one-way part* of the message-transfer, then this may be called **Chain-Model of switching**. In this case no explicit registration-process is necessary, usually there occurs a cascading over several

forwarding entities to a drain. In Java: Exception throwing uses a chain of method-calls to jump back to a catch-clause used as a drain (page 282)¹¹⁶.

Some extended bidirectional relay-functions are implemented by entities called management- or resource-distribution-middleware, broker, firewall, gateway, bridge, server-servlet, three-tier database-access.

As another variation, email-functions and enews-infrastructures have the following model of a protocol (see also page 371):



The message-switching-process can be seen further refined in several ways:

A single source sending a message to a single drain is also called a **point-to-point connection**.

point-to-point
connection

If there are multiple sources and drains, then messages have to be switched to reach a single drain or messages can be forwarded to multiple drains. Message forwarding to multiple, usually chosen, drains is called **multicasting**, message forwarding to every available drain is also called **broadcasting**.

multicasting and
broadcasting

The switching-process can be done by temporarily reserving a channel between the source and the drain (temporary point-to-point connection) which is called **channel-switching** or by using control-information furnished by the messages to route each message, which is called **packet-routing**¹¹⁷. (For more on switching and routing, see page 345.)

OUTLOOK: The notion of a protocol and some of its related notions dominate the description of computer-networks, beginning around page 337. Especially the introduction of open protocols and their layering, beginning with page 347, structures ideas associated with computer-networks.

- Session: 1. **Time-interval** during which a communication-process is started, proceeds and ends. During that time-interval, a series of requests and responses are exchanged freely between two communication-partners (human or non-human, client/server or peer-to-peer communication) without needing to re-establish the communication-line. Sessions are possible only when using connection-oriented protocols.
2. The **communication-process** itself, starting at a point in time and ending at a point in time. The standard example of a session is given by a user who turns the computer's power on, beginning a computing session and when turning the power off, ending that session. The protocol used in that case is given by the computer-hardware usage (Power-switch, keyboard, mouse), the rules according to which the native operating-system works and the handbooks of the applications which are run.
3. (Java) A **Java object** used by servlets to gather information about various related HTTP-accesses of the same client. This is not further considered in this text, such objects are used with the Java 2 Enterprise Edition (J2EE).

Practical Event-driven Communication

Practical Event-driven Communication — If a programming-environment uses method-calls to communicate, then the caller has to know the signature and functionality of the called method. Changes in the drain, that concern the signature of the method, immediately affect all sources, that means the callers of the method. This problem is eased by introducing messages. Now the KNOWLEDGE ABOUT HOW TO COMMUNICATE IS EXTERNALIZED; the communication cannot be affected by internal changes of source and drain as long as the message-format is kept the same. In this sense, messages are the precursor

¹¹⁶The deprecated Java 1.0 AWT event-handling jumps back within the containment chain of screen-components to a handler-method, the drain (page 574).

¹¹⁷In the past, telephone lines were connected by successive switching of copper-wire connections until the users were connected by one continuous line. This exemplifies channel-switching. Telephoning via the Internet means that the spoken word is digitized into a stream of bytes and the stream is broken into packets that reach their destinations over possibly different data-lines. Here the spoken word is transferred by packet-switching.

for defining protocols. Thus, sending messages may require some formalized overhead, but this evades working, with possibly rather specialized, method-definitions!

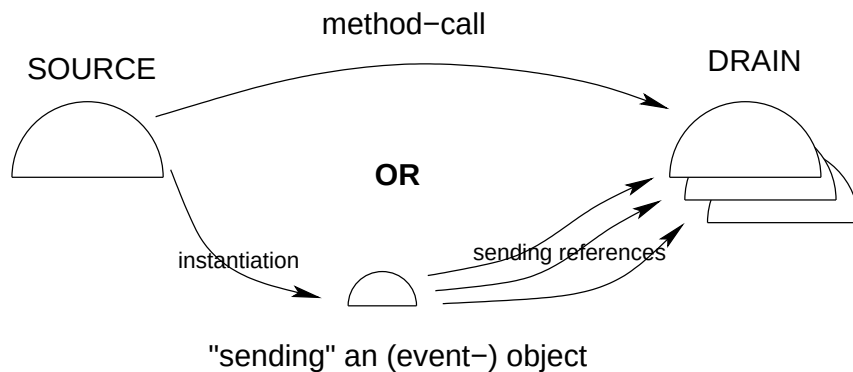
Using the event-model in programming SEPARATES THE MANAGEMENT OF KNOWN COMMUNICATION-STRUCTURES (protocols, data-formats) FROM THE COMMUNICATING OBJECTS (Input/Output-handlers). If these principles are applied properly, then the advantages include simplified application-programming and more efficient use of source-code, especially with larger pieces of source-code. Actually, messaging gives another way to structure code conceptionally!

For example, the Abstract Window Toolkit (AWT), designed to manage single user interfacing, relies on the event delegation-model for communication. Likewise, the Java-Beans-component Application Programming Interface (API) allows the (Bean-)objects to communicate heavily by exchanging predefined messages.

But below all the organizational overhead, THE MESSAGE CONCEPT WITHIN OBJECT-ORIENTED PROGRAMMING REDUCES ITSELF TO A SUCCESSION OF SIMPLE, STANDARDIZED, MOSTLY IMPLICIT, METHOD-CALLS IN PROGRAMMING PRACTICE.

Another aspect dominates Java programming: The event-delegation model is realized by collecting all the event-information into an event-object. Informing various drains, then just amounts to sending various references of that standardized event-object! "Dispatching, firing or throwing an event-object" in Java circumscribes the mere instantiation of an object and the distribution of its references!

Communication by method-call or by dispatching an Event-Object



So, the high-level-approach ("sending an event-object") within source-coding is REDUCED BY THE JAVA COMPILER TO A SUCCESSION OF SIMPLE METHOD-CALLS, though often highly standardized, as the essence of the communication-process.

For another listing of the advantages and disadvantages of the event-managed communication, also consider the comments at the beginning of the source-code on page 633.

Code Structuring: Messaging

Code Structuring: Messaging — Calling a method requires to know that the receiver has implemented that particular method. This knowledge has to be present independently of whether the method is called to give information or if the method is called to get information. SENDING MESSAGES PERMITS TO HAVE LESS INFORMATION ABOUT THE RECEIVER OF THE OBJECT. THE ONLY COMMON KNOWLEDGE HAS TO BE THE INFORMATION CONTAINED INSIDE AN EXTERNAL PROTOCOL SPECIFICATION.

In messaging, an event triggers a source-object to throw or dispatch an event-object — rather a reference to the event-object — that is then caught by a drain-object. The event-object is more or less explicitly directed to the drain-object by some switchboard-mechanism. If there exists no platform-specific mechanism of distributing (references of) event-objects, then the programmer has to explicitly write a switchboard (class or object) for the thrown objects.

Java: Implemented: Communication by “Throwing Objects”

Java: Implemented: Communication by “Throwing Objects” — Or rather throwing references of objects! — Recall the introductory example on restricted jumping (page 112). And remember the examples of Java-code using this functionality for communicating errors (see pages 192 and 259).

This section considers those exceptional conditions or problems that occur irregularly during the run-time of Java-code. Examples for such conditions or problems are an integer division by zero, a non-existent file to be opened or missing RAM-space. The Java Runtime System (JRS) furnishes a fully-implemented infrastructure to manage such situations:

From the block of code, in the active thread, at which the problem occurs, the Java Runtime Environment (JRE) or the application-programmer can throw a (reference of a) message-object that represents a report about the kind of problem that occurred. To catch and handle this message-object, that block of code has to be (called) inside a try-statement. The code written into the catch-blocks which follow the try-statement should handle the problem. If the message is not caught, which means that no handling- or no try-clause is found, then the throwing thread is terminated¹¹⁸ by the Java Runtime Environment issuing an information-string to the standard output, mostly a terminal-window. This describes in short the problem-report and related flow-control by “throwing objects”.

So, in Java, “throwing messages” means changing the flow of control by transferring this flow to another part of the code and by delivering a reference of class `java.lang.Throwable`. All in all, a throw-statement identifies the innermost try-statement and branches to the respective catch-clause. If necessary, an entire reiterated sequence of try-catch-throw-clauses is run through.

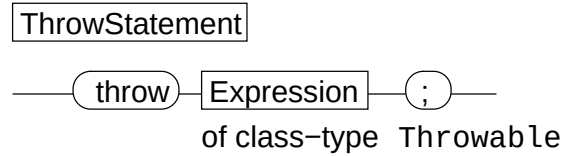
Source: Who throws How from Where?

Source: Who throws How from Where? — The application-programmer can throw a message from any expression in some block of code:

```
if( somethingIsWrong )
    throw    new java.lang.RuntimeException("Something is wrong!");
```

Of course, the message-object could have been instantiated with a name of its own: `RuntimeException message = new RuntimeException("Something is wrong!");` but often it suffices if the instantiation is done anonymously. The throw-statement allows the Java programmer to issue own error-messages. The throwing of the message-object stops the regular processing of the code and activates a message-handling infrastructure (either the try-catch-routines for exceptions or, in the case of message-objects of Error-type, totally stopping the entire Java Runtime Environment).

¹¹⁸ Generally, only the throwing thread is terminated, not the entire Java Runtime Environment!



ThrowStatement := throw *Expression* ; (Where *Expression* produces a reference pointing to an object of type `java.lang.Throwable` or a sub-class.) — Expression 90

— Or the Java Runtime Environment, even the Java Compiler, can throw messages:

```

int i = 1/0;
// produces an Arithmetic Exception at compile-time
int n = 0, j = 1/n;
// produces a java.lang.ArithmeticException: / by zero at runtime

```

The object, the piece of Java-code (source-code or byte-code) from which the message-throwing takes place, represents the source of that message. (In some cases the message is caught within the object, then the object becomes its own drain.)

What is thrown?

What is thrown? — (Throwable) Only instances derived from (sub-classes of) the `java.lang.Throwable`-class can be thrown. (`java.lang.Throwable` is an immediate sub-class of `java.lang.Object`) The `Throwable`-class essentially encapsulates a message-string and a stack-trace information string. So, thrown message-objects (or simply “messages”), need to be of class-type `java.lang.Throwable`! Of course, as every Java object, the `java.lang.Throwable`-objects have to be instantiated before thrown. The instance-method `getMessage()` retrieves the string used as an argument of the class’s constructor.

Exceptions and Errors

(Exceptions and Errors) **Exceptions** thus are objects of type `Throwable`, that inform about problems running (a thread in) the Java-code; contrary to **Errors** which inform about more basic malfunctions within the Java Runtime Environment. Errors describe problems that occur with the supporting code: The Java Runtime Environment, a class-loader. Contrary to exceptions which, if unhandled, at worst may terminate their thread, most errors are unrecoverable and terminate the entire Java Runtime Environment!

checked exceptions

(Checked- and Unchecked Exceptions) Programming-commands that possibly produce exceptions which the Java compiler requires to be treated with an enveloping `try-catch`-construct are also called **checked exceptions**. Thus, the throwing of a checked exception has to be anticipated by enclosing the sensitive area of the source-code with a `try-catch`-clause.

Checked Exception All sub-classes of the class `java.lang.Exception` can be handled with the `try-catch`-mechanism¹¹⁹. The throwing of references of class-type `Exception` has to be announced in method-definitions with the `throws`-clause, enabling these objects to cross method-calls on their way back up the method-call chain. This `throws`-announcement assures a proper switching of the identifiers of type `Exception`. In this way the channels are opened for objects of class-type `Exception` (or their sub-classes) to be propagated along a method-call chain. (Otherwise the throwing *thread* terminates prematurely.)

```

Type methodName() throws ExceptionName
{
... if(somethingWrong()){ throw new ExceptionName() }...
}

```

This allows the Java compiler (triggered with `javac`) to check whether there exists an exception-handling infrastructure within the program. — When extending classes, observe that overriding methods can only throw what the overridden method already throws.

RuntimeException All sub-classes of `java.lang.RuntimeException` produce message-objects that are **routed back** to the nearest `catch`-clause or to the `run()`- or the `main()`-method **WITHOUT need-ing throws-clauses in methods or classes**. The Java Runtime Environment can encounter such

¹¹⁹As already presented, a statement `int i = 1/0;` generates a Java compiler error-message “`ArithmeticException`”. Whereas, at runtime, the statement `int n=0, j =1/n;` can be detected and handled by a `try-catch`-clause without severely interrupting the Java-application.

type of deficiencies throughout the source-code. For example, a `java.lang.NullPointerException` is a (sub-class of the class) `java.lang.RuntimeException`. But, if these exceptions remain uncaught, they still terminate the *thread* from which they have been thrown.

Error If thrown by the Java Runtime Environment, all sub-classes of the class `java.lang.Error` abort the thread, the application and the *entire* Java Runtime Environment, they ignore try-catch-clauses. The Java Runtime Environment uses a message-object of such type to inform about own limits, like an `java.lang.OutOfMemoryError`.

The objects that are derived from the exceptions `java.lang.RuntimeException` and from `java.lang.Error` return from methods without needing to inform the method with a throws-declaration. Therefore messages of type `Error` and `RuntimeException` need not be listed in a throws-clause. `Error`, `RuntimeException` or a sub-class of them can occur in virtually every piece of code, therefore they need no throws-clause in method-declarations. Their counterpart is given by a:

Checked Exception: The type of exception that *has to* occur inside a try-block of a try-catch-clause. AND the kind of exception that has to announce itself in a throws-clause of a method-declaration.

The Java Runtime System has been given a hierarchy of predefined `java.lang.Throwable`-classes with names that indicate what went wrong. So, the messages, used by the Java Runtime Environment, are structured into a hierarchy of sub-classes to give a better view on possible problems. That hierarchy does not extend any functionality! Many messages in that programmer-extensible hierarchy are self-explaining.

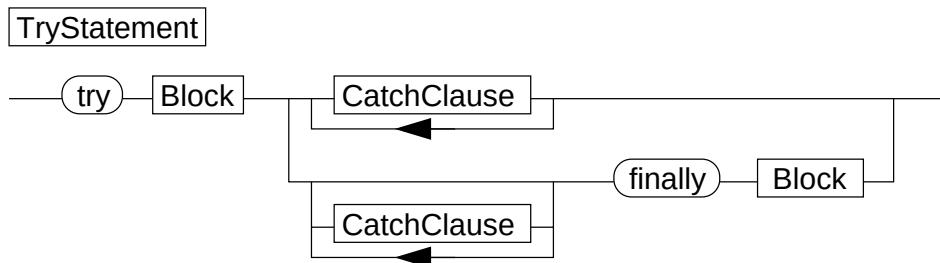
Object	
Throwable	
Exception	exceptions, which on runtime always can be caught with a try-catch-clause
ClassNotFoundException	class-loader does not find class-file (byte-code)
CloneNotSupportedException	Cloneable-interface not implemented
IllegalAccessException	class-loader is denied access to class-file
InstantiationException	instantiating an interface or an abstract class
InterruptedException	response of a sleeping thread to interrupt()
NoSuchFieldException	variable cannot be found
NoSuchMethodException	method cannot be found
RuntimeException	exceptions thrown at runtime, not needing a try-catch-clause, but can be caught by such a clause
ArithmeticException	integer division by zero
ArrayStoreException	wrong type of value attempted to assign as array-element
ClassCastException	misapplied type-cast, for rules see page 169
IllegalArgumentException	argument not matching parameter-list of method
IllegalThreadStateException	example: start an unfunctional thread
NumberFormatException	String parsed, but gives no number
IllegalMonitorStateException	wait(), notify(), notifyAll() misapplied
IllegalStateException	unsuitable method-call
IndexOutOfBoundsException	super-class of the two below:
ArrayIndexOutOfBoundsException	index smaller than zero or larger or equal than length of array.
StringIndexOutOfBoundsException	index smaller than zero or larger or equal than length of string (in String- or StringBuffer-object)
NegativeArraySizeException	length of array smaller than zero in declaration
NullPointerException	class-type access attempted without existent object
SecurityException	when SecurityManager settings are violated, see page 462
Error	once thrown, most Errors terminate the Java Runtime Environment
LinkageError	problems in resolving references between classes
ClassCircularityError	circular reference among classes
ClassFormatError	class-file-format differs from Java class-format
ExceptionInInitializerError	in static-initializers
IncompatibleClassChangeError	incoherent changes of class-definitions
AbstractMethodError	invoking an abstract method
IllegalAccessError	usually a compile-time error, may occur after unupdated compilation on runtime
InstantiationError	instantiating interfaces or abstract classes, after unupdated compilation
NoSuchFieldError	variable cannot be found, if not found on compile-time, occurs after unupdated compilation
NoSuchMethodError	method cannot be found, if not found on compile-time, occurs after unupdated compilation
NoClassDefFoundError	class-definition cannot be found
UnsatisfiedLinkError	implementation of a native method is missing
VerifyError	byte-code-verifier irritated (page 451)
VirtualMachineError (abstract)	something is wrong with the Java byte-code interpreter
OutOfMemoryError	No RAM available for the byte-code interpreter's data
StackOverflowError	No RAM available for the byte-code interpreter's commands
InternalError	something else is wrong within the current byte-code interpreter (the part of JRE that implements the Java Virtual Machine (JVM)).
UnknownError	something unspecifiable is wrong within the current Java Runtime Environment (JRE).

These are only throwable objects that have been defined in the package `java.lang`, other packages may furnish further extensions of the `Throwable`-class. The package `java.rmi` (RMI acronym for Remote Method Invocation) declares the `java.rmi.RemoteException` which extends the class `java.io.IOException`,

Since there exists no explicitly addressable switchboard-instance and the only places to influence the Java switching-mechanism are the catch-clauses scattered across the source-code, this constitutes an example of a **non-localized switching-mechanism** and cascading-mechanism — also used by Runtime Environment for signaling errors to the application-programmer or to the user.

non-localized
switching-
mechanism

A try-statement encloses a block from where to catch possibly various types of exceptions. The branching leads to cleanup code in exception-specific catch-clauses and to the single finally-clause per try-statement.



TryStatement := try *Block* < *CatchClause* > | try *Block* {*CatchClause*} finally *Block* — Block 105, Catch-Clause see below

Observe that the first matching catch-clause becomes the handler for the thrown exception; catch-clauses that come later are ignored. Thus catch-clauses are order-sensitive. So the handlers of more specific exceptions should appear first in the list of catch-clauses.



CatchClause := catch ("FormalParameter") *Block* — FormalParameter 118, Block 105.

The try-catch-mechanism is also used by the Java Runtime Environment (JRE) for telling the application-programmer and user about something that went wrong.

Especially the catch-clause can be filled with code that tries to recover what went wrong (division by zero, a network-connection that went down). The blocks in the catch-clauses are also called the **exception-handlers** of the try-block. Minimalistic exception-handlers, which just inform about an exception having been thrown, may take one of the following two forms:

catch-clause =
exception-
handler

```
catch( Exception e ){ System.out.println( e ); } // OR
catch( Exception e ){ e.printStackTrace(); }
```

What aspect gives the advantage over a simple pre-checking with an if-statement? The if-statements usually have to be more specific than a try-code-block. Multiple if-statements, for seldom arising cases, muddle the source-code (see also page 112).

At the beginning of this section have been mentioned examples for the use of the Java infrastructure for channeling objects of class-type Throwable (integer division by zero, a non-existent file to be opened or missing RAM-space). Here is presented another one which demonstrates some of the more specific aspects of throwing objects:

```
/* different ways to throw or let throw exceptions */
class Warning extends java.lang.Throwable
{ Warning(String s){ super( s + ", RELAYed by an object of type Throwable!" ); }
}

public class ExceptionThrowing
{ public static void main( String unused[] ) throws Exception
{
    try{ int x, y=0; x = 1/y; }
    catch( Exception e ){ System.out.println( e ); } // or e.printStackTrace()

    try
    { method();
    }catch( Exception e )
    { System.out.println( "received " + e.getMessage() + " as a regular Exception" );
      throw new Exception();
    }
}
```

```

        }catch( Throwable e )
        { System.out.println( "received " + e.getMessage() + " as a Throwable-object" );
        }
    }
    static void method() throws Throwable
    { try{                // outcommented try-catch-statement AUTOMATES the RELAY
        anotherMethod();
    }catch( Exception e ){ throw new Warning( e.getMessage() );} // RELAY
    }
    static void anotherMethod() throws Exception
    { if( true ) throw new Exception( "initial message" );
    }
}
/*
(outcommented try-catch-statement in 'method()' AUTOMATES the RELAY:)
(this case also rethrows an exception, caught by the Java Runtime Environment)
[localhost:~/JavaCodeFiles/Basics2/CSMessaging] andreadipietro% javac ExceptionThrowing.java
[localhost:~/JavaCodeFiles/Basics2/CSMessaging] andreadipietro% java ExceptionThrowing
java.lang.ArithmeticException: / by zero
received initial message as a regular Exception
Exception in thread "main" java.lang.Exception
        at ExceptionThrowing.main(ExceptionThrowing.java:24)
[localhost:~/JavaCodeFiles/Basics2/CSMessaging] andreadipietro%
*/

/*
(try-catch-statement in 'method()' RELAYS by rethrowing an object of type Warning:)
andreadipietro@linux:~/JavaCodeFiles/Basics2/CSMessaging > javac ExceptionThrowing.java
andreadipietro@linux:~/JavaCodeFiles/Basics2/CSMessaging > java ExceptionThrowing
java.lang.ArithmeticException: / by zero
received initial message, RELAYed by an object of type Throwable! as a Throwable-object
andreadipietro@linux:~/JavaCodeFiles/Basics2/CSMessaging >
*/

```

See also file-opening on page 298 and on economizing of catch-clauses see the program on file-reading and -writing on page 298; there only exceptions are caught and handled indiscriminately.

Summary: “Throwing Objects”

Summary: “Throwing Objects” — The `java.lang.Throwable`-class has instances that are meant to serve as special event-objects to be used with a standard switching-mechanism. As required by the Java compiler, the “checked” messages of the class-type `java.lang.Exception` are instantiated and thrown within a try-block of source-code and thus interrupt the running program from within that try-block. Alternatively the throwing may take place from a method that implements an appropriate throws-clause. Then, the central element of the switching-mechanism becomes some enclosing try-catch-finally-clause: This clause receives a message that has been thrown from within the then-abandoned-try-block and matches it with the appropriate catch-block. From there, the message can be rethrown or be handled, even by a method-call.

An “unchecked” message of class-type `java.lang.RuntimeException` can be thrown from anywhere; but if that message is thrown from within some try-block, then it is caught and processed as described above. Without such a try-block, throwing any kind of exception means stopping the throwing thread, preserving the rest of the working Java Runtime Environment (JRE).

Instances of the `Throwable`-class permit branching across method- and class boundaries, limited by the restrictions imposed by the try-catch-finally-clause. In that context they can be used effectively for sending messages. Any catch-clause at least should print out the message of the caught exception!

After having sent messages of class-type `java.lang.Error`, the Java Runtime Environment (JRE) shuts itself down.

Java: To be Implemented: Observable-Class and Observer-Interface

Java: To be Implemented: Observable-Class and Observer-Interface — The observable/observer-mechanism makes an `Observable`-object broadcast an object’s references to registered `Observer`-objects. First consider a sketch of the basics:

Source: An object derived from the `java.util.Observable`-CLASS.

Message: An object, an instance of the `java.lang.Object`-class.

Switchboard: Is furnished by the Java Runtime Environment (JRE).

Drain: An object that implements the `java.util.Observer`-INTERFACE by defining an `update()`-method that contains the handler-code.

Trigger: PUSH: The method `notifyObservers()` of the `java.util.Observable`-object.

The source of the message is given by an instance of the class-type `java.util.Observable` (or rather an extension thereof). This class furnishes the fully-functional methods:

`public Observable()` When called with the keyword `new`, constructs an object of class-type `Observable`, with zero observers registered, and returns a reference to that object.

`public void addObserver(Observer)` Adds, registers an observer to the set of observers for this object.

`public int countObservers()` Returns the number of observers of this object.

`public void deleteObserver(Observer)` Deletes an observer from the set of observers of this object. A call of the method `deleteObservers()` removes each observer from the set of observers so that this object no longer has any observers (note the “s” at the end of the method’s name).

`protected void setChanged()` Marks the `Observable`-object as changed, so that a call of the `notifyObservers()`-method will create a (message-)object and dispatch its references.

`public boolean hasChanged()` Returns `true` if this object has changed, otherwise returns `false`.

`protected void clearChanged()` Marks the `Observable`-object as unchanged, so that a call of `notifyObservers()` won’t cause any event-dispatching.

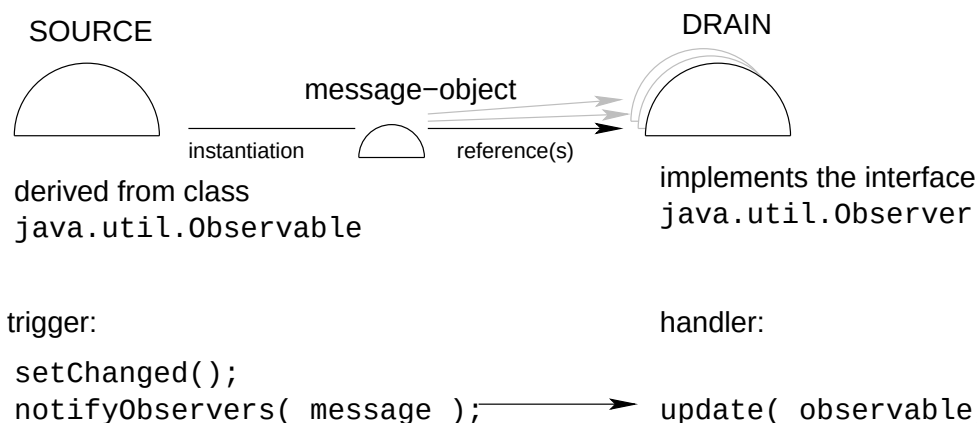
`public void notifyObservers(Object message)` If the `Observable`-object, from which this call is issued, has changed — as can be seen by the `hasChanged()`-method which should return `true` — then all of its observers are notified and the `clearChanged()`-method is automatically called to indicate that this object has no longer changed. This method implements a PUSH-trigger. — Then, on the `Observer`-side, this method’s call automatically triggers a call of the method `update( observable, message )` which hopefully has been implemented non-trivially by the Java programmer.

`public void notifyObservers()` Same as above, only with a message-reference set to `null`, giving only a sort of a signaling call, with very little information: The equivalent would be a call `notifyObservers(null)`. On the `Observer`-side this translates to a call `update( observable, null )`.

The interface `java.util.Observer` gives the method-signature (plus modifiers and return-type)

`public abstract void update(Observable, Object)`

to be implemented. This method is called whenever (*first*) the observed object has been set changed and then (*second*) one of the `notifyObservers()`-method has been called. These functionalities have been visualized within the diagram below and they have been implemented in the following piece of source-code:



```

/* A Producer sends strings to a Consumer by calling notifyObservers() */

class Out{ Out( String s ){ System.out.println( s ); } }

class Producer extends java.util.Observable
{ public void setChanged(){ super.setChanged(); }
  // this makes protected setChanged()-method accessible publicly
}
class Consumer implements java.util.Observer // requires update()-method
{
  public void update( java.util.Observable observed, Object messageObject )
  { new Out( this.toString() + " got " + messageObject + " from " + observed ); }
}
class ObservableObserver
{ public static void main( String unused[] )
  { Producer source = new Producer();
    Consumer firstDrain = new Consumer();
    Consumer secondDrain = new Consumer();
    new Out( "number of observers " + source.countObservers() );
    source.addObserver( firstDrain );
    source.addObserver( secondDrain );
    new Out( "number of observers " + source.countObservers() );
    source.notifyObservers( "a first String object as message" ); // lost
    new Out( "hasChanged? " + source.hasChanged() );
    source.setChanged(); new Out( "hasChanged? " + source.hasChanged() );
    source.notifyObservers( "a second String object as message" );
    source.deleteObservers();
    new Out( "number of observers " + source.countObservers() );
  } }
/* Why is setChanged() redeclared 'public' in Producer-class?
The inherited method produces a 'method has protected access' compile-time error
in the main()-method when calling source.setChanged().
The class ObservableObserver is not a sub-class of Observable
and therefore the protected method setChanged() cannot be invoked!
*/

/*
andreadipietro@linux:~/JavaCodeFiles/Basics2/CSMessaging > javac ObservableObserver.java
andreadipietro@linux:~/JavaCodeFiles/Basics2/CSMessaging > java ObservableObserver
number of observers 0
number of observers 2
hasChanged? false
hasChanged? true
Consumer@7a78d3 got a second String object as message from Producer@129206
Consumer@30f13d got a second String object as message from Producer@129206
number of observers 0
andreadipietro@linux:~/JavaCodeFiles/Basics2/CSMessaging >
*/

```

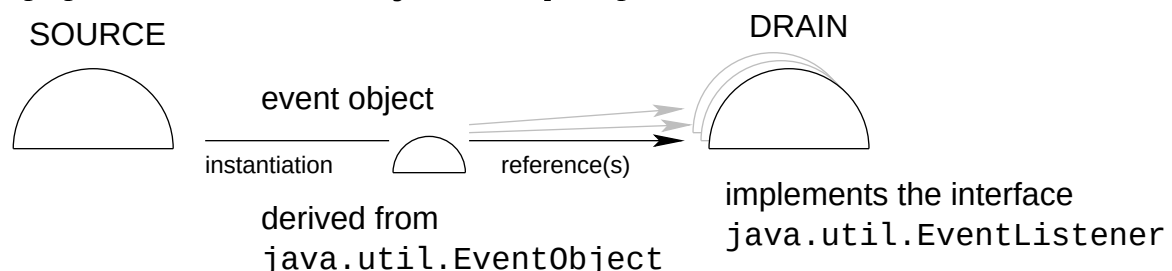
The first String-object does not reach its observers because the source's `setChanged()`-method has not been called. Observe that the methods `setChanged()` and `clearChange()` of the predefined class `java.util.Observable` are declared protected (page 175). Therefore, here, these methods only can be invoked by code *within* sub-classes of the class `Observable`. Or, the access-area of the method can be widened by overriding the method with a public version, which itself calls the more restricted one (`super.setChanged()`).

access a method
by moving code
into its
access-area or
by overriding

For what is this observable/observer-mechanism meant and used? This means to be an offer for use with the Model View Controller concept (page 515), presented in the part about the single user interface. But nevertheless the mechanism can be used for any problem it suits!

Java: Event Model — A Mechanism fully to be Implemented

Java: Event Model — A Mechanism fully to be Implemented — Java gives a more general approach to messaging within the context of the `java.util` package.



The `java.util.EventObject`-class is intended as a basic template for user-generated event-messages. Java furnishes no switching-mechanism and no registering mechanisms, so almost ALL of the functionalities have to be implemented by the Java programmer. This supports generality but makes the customizing a bigger programming task. A first sketch of the basics reveals how little is furnished and how much has to be fully implemented by the Java programmer:

Source: An object for which the Java programmer has to implement drain/listener registering methods. And possibly also some event-dispatch-methods. Or truly general: An object or class that has instantiated a `java.util.EventObject` in some of its method-calls.

Message: An object, an instance of the `java.util.Event`-class or a sub-class, that means an object of type `java.util.Event`.

Switchboard: Fully to be implemented by the Java programmer!

Drain: An object that implements the empty¹²¹ `java.util.EventListener`-interface which just tags the class and its derived objects. The programmer has to declare an event-handler-method and has to make sure that it is called when the event is received.

Trigger: Fully to be implemented by the Java programmer!

Thus, the furnished class `java.util.EventObject` and the empty interface `java.util.EventListener` merely give a framework into which the Java programmer has to implement an entire infrastructure of event-dispatching, drain(listener)-registering, event-switchboarding and event-handling.

Source

Source — Source-code of event-sources may take any form if only an event reaches the drain! Nevertheless there are some standard guidelines that can be followed: Event-sources, that send Java event-objects, usually follow design-patterns for registering drains or targets for their event-objects:

```
public void addListenerType( ListenerType listener );
public void removeListenerType( ListenerType listener );
```

The number of registered listeners can be restricted by the Java programmer, then the following pattern may be used (to be used only with exactly one permitted listener, that means a case of unicast event-messaging):

```
public void addListenerType( ListenerType listener )
    throws java.util.TooManyListenersException;
public void removeListenerType( ListenerType listener );
```

Another convention recommends message- or event-classes to end with the word `Event`, listener-classes end with the word `Listener`.

Finally, the design-pattern to sent events may take one of the following form:

```
protected void notify...();
protected void fireEventType( EventType e );      (Java bean typical)
protected void dispatchEvent( EventType e );
```

In those cases, where the source and drain are given by the same object, the pattern described above still works. A simplified pattern that tries to shortcut the routine described above, which processes event-objects directly, may be given by the signature, modifier and return-type:

```
protected void processEventType( EventType e );
```

All the patterns above are realized within the Abstract Window Toolkit (AWT), as introduced on page 572. There, the last line above has to be activated by calling a method `java.awt.Component.enableEvents(long BIT_MASK)` and inactivated by calling `disableEvents()` (page 593, for more on masks see page 533 and for more on their use as arguments see page 591).

¹²¹The interface `java.util.EventListener` prescribes no method-signature and no static variables; the implemented interface just tags objects or classes that register somewhere as event-listeners. A sort of really minimalistic type identification.

Event

Event — The class `java.util.EventObject` is a class directly extending `java.lang.Object`. The `public Object getSource()`-method returns a reference to the object that generated the event-object. The overridden method `public String toString()` reproduces the message-string that is sent with every object of type `java.util.EventObject`.

Drain

Drain — The interface `java.util.EventListener` prescribes no method-signature and no static variables; this interface just tags objects or classes which can be registered somewhere as event listeners. That's all the Java Runtime Environment (JRE) provides, everything else, especially the organization of the event-handling, has to be done by the programmer.

Trigger

Trigger — A possible convention may require to define a method `dispatchEvent( EventType e )` or a method `fireEventType( EventType e )` which, as called with a reference to a Java event-object, activates the switchboarding and ultimately the handler-code.

Much of the structures that have been described above, can be seen in future applications. Therefore the following example merely illustrates the idea. The example may be skipped in reading, but it makes explicit what parts have to be programmed even only to furnish a minimally implemented infrastructure.

Trivial Example

Trivial Example — The following program illustrates the usage of event-objects. But the whole program could be substituted by the one simple method-call. In that sense the program shows only the overhead which is necessary to use the flexibility of the event-model:

```
/* the pure event-throwing infrastructure (could be substituted by a simple method-
call) */
class Out{ Out( String s ){ System.out.println( s ); } }

class Event extends java.util.EventObject
{ Event(Object source)
  { super(source);
    new Out( "4. Source: Event-object instantiated" );
  } }

interface EventListener extends java.util.EventListener
{ void handleEvent( Event evt );
}

class Source
{ Source(){ new Out( "1. Source instantiated" ); }
  EventListener listener;
  void addEventListener( EventListener l ) // called by drain -- links source to drain
  { listener = l;
    new Out( "3. Drain calls: source.addEventListener establishes link (The Call)" );
  }
  void removeEventListener( EventListener l ){ listener = null; }
  void activateTheSourceObject()
  { Event THEEVENTOBJECT = new Event(this); // event-object instantiated
    new Out(
      "5. Source dispatches MESSAGE: calls now (registered) listener (The Callback)" );
    listener.handleEvent(THEEVENTOBJECT); // central message transfer by method-call <===
    new Out( "7. Source finished" );
  } }

class Drain implements EventListener
{ Drain( Source source)
  { new Out(
    "2. Drain instantiated, registers as event-listener and furnishes handlers" );
    source.addEventListener( this );
  }
  public void handleEvent( Event evt ) // event-handler is required to be public
  { new Out( "6. Drain: Message received (and handled)" );
  } }
}
```

```

class EventMechanism
{ public static void main( String unused[] )
  { Source theSource = new Source();
    Drain theDrain = new Drain( theSource );
    theSource.activateTheSourceObject();
    theSource.removeEventListener( theDrain );
    theSource.activateTheSourceObject(); // null-pointer exception
  } }

/*
marycosway@linux:~/JavaCodeFiles/Basics2/CSMessaging > javac EventMechanism.java
marycosway@linux:~/JavaCodeFiles/Basics2/CSMessaging > java EventMechanism
1. Source instantiated
2. Drain instantiated, registers as event-listener and furnishes handlers
3. Drain calls: source.addEventListener establishes link (The Call)
4. Source: Event-object instantiated
5. Source dispatches MESSAGE: calls now (registered) listener (The Callback)
6. Drain: Message received (and handled)
7. Source finished
4. Source: Event-object instantiated
5. Source dispatches MESSAGE: calls now (registered) listener (The Callback)
Exception in thread "main" java.lang.NullPointerException
    at Source.activateTheSourceObject(EventMechanism.java:24)
    at EventMechanism.main(EventMechanism.java:43)
marycosway@linux:~/JavaCodeFiles/Basics2/CSMessaging >
*/

```

If the `EventListener`-object is deregistered, then a `NullPointerException` is thrown by the system because the pointer to the drain is `null`.

Unicasting events can be made mandatory by letting the `addListener()`-method throw a reference of class-type `java.util.TooManyListenersException`, in the case there are more than one listeners registered. The `java.util.TooManyListenersException` also could be extended to a “`OneListenerOnlyException`”, if that is found more appropriate.

As the case of a unisource-unicast example, the source-code above demonstrates the pure mechanism of communication with objects of type `java.util.EventObject`.

Remarks about Non-Trivial Extensions

Let the unisource-unicast case become complemented by the multisource-multicast case (as envisioned in the graphics on pages 272 and 274). In the multisource-multicast case, where multiple event types have to be channeled to multiple drains, the advantages of the source-message-drain model become clearer.

The main advantage¹²² of implementing that model is given by a clear separation of notions, allowing to specify a “protocol” to be followed independently from already existing sources and drains. Then the two points of contact, remaining to be clarified, are the type of message and the method of registering:

Type of Message The only thing source and drain need to know of each other, is the format of the event-objects; this maximizes independence.

Method of Registering Changes in design of messages or addition and removal of sources and drains become possible with much more ease. The switchboarding-process can be automated to such an extent that the programmer needs only to register the drain-objects, nothing more (consult the graphic on page 276). Thus searching for the handler-methods and, to a certain extent, even registering them to the switchboard could be done automatically by the switchboard. Here the `java.lang.reflect`-package proves to be very useful.

Registering and deregistering of events should be easy and localized at one point. Registering of drains should be also possible by one simple universal method-call, for example, just by giving the switchboard’s constructor an array of potential drain-objects. Registering sources means doing the same with source-objects.

The next thing to know could be: To which entity to register the listeners: In cases of one single source, the switchboarding often is integrated into the source. So, if there exists only one single drain-object, then the switchboard-object often is attached to the drain by defining it as an inner class. Those switchboards, that forward event-objects to handling methods within one single object, are often called **adapters**.

adapter

¹²²What advantage has this over a simple method-call? Basically a greater flexibility in more complex situations like multicasting of event-objects, in cases of event-exchange where the source is a GUI-component, in stream-handling where input-streams are sources and output-streams are drains, furthermore events can be sorted hierarchically like the exceptions and errors. (GUI is the acronym for Graphical User Interface)

An elementary switching-mechanism takes an event-object and uses that object to determine the event-handler-method to call. Usually the reference of the event-object is forwarded to the event-handler.

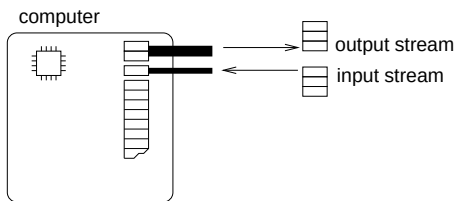
As it is imaginable, switchboards can easily increase in complexity. On network-level such switchboard-software-objects can evolve to object-broker-software. (The hardware equivalent to software-switchboards on the packet- or frame-level of networks are routers.). Especially when considering the broad area of communication-hardware — for example networks — further routing-information also may be stored outside the messages, in some database (graphic page 272).

When using custom-written event-objects, this includes to-be-programmed switchboard-classes or -objects that take over the registering of listeners and routing of events. If the switchboard-class is programmed general enough, then re-routing (new) events becomes possible without any change of code. There has been deployed a fully-functional switchboard-class, together with sources and drains, on page 633, to demonstrate the flexibility of such an implementation.

Event-handling using event-objects with a fully-implemented infrastructure is offered in Java by the Java Abstract Window Toolkit (AWT, page 575) and with the Java-Beans Application Programming Interface (API), on page 314.

Concept: Streams in Java

Concept: Streams in Java — Remember the computer-model on page 28 and how the communication with the outside world had been visualized.



A large part of the processor's address-space is occupied by the Random Access Memory. Other parts of the address-space furnish the contact to the outside world (also called input/output-ports, see page 231): In many cases the operating-system encapsulates the organization of the byte-transfer by using stream-constructs.

Remember also the differing meanings of the word “stream” when introducing the stream-model on page 272. The word “stream” is used to denote three things: — A message that contains (mainly)

data. — A channel of communication the data-messages take, such a channel is also called pipe. — Or, a “stream” can be one end of a communication-channel or a filter on such an end of a communication-channel, a “half-stream”. Such a filter may buffer received data, post received data for reading and even reformat the received data. IN JAVA, STREAMS REALIZE ENDS OF COMMUNICATION-CHANNELS, FILTERS AND INNER-PROCESS-PIPES.

Java stream: end of a communication-channel (“half-stream”)

This Java stream-concept, used for internal purposes (like filtering data) or used for communicating with the outside world, is presented in the text below.

A simple example of a Java-stream is given by the `StringReader` stream-class of the `java.io` package. This stream can be fed with a `String`-object. The stream-object stores the string characterwise. And, character for character, the stream can be read out by invoking the stream's `read()`-method. The program below prints out each single character of a given `String`-object, but does this successively with the delay of one tenth of a second per character:

```
/* A String-object is transformed into a Stream-object.
   This stream is read out integerwise, in intervals of 1/10th second;
   then transforming the integer into a character and printing it.
*/
class HelloStringReaderStream
{ public static void main( String unused[] )
  { String aString = new String( "Hello StringReader stream" );
    java.io.StringReader aStream = new java.io.StringReader( aString );
    try
    { do
      { int anInteger = aStream.read();
        if( anInteger == -1 ) break;           // -1 = end of stream
        System.out.print( (char)anInteger );  // Unicode-characters
        Thread.sleep(100);                   // 100 milliseconds = 1/10 of a second
      }while( true );
    }catch(java.io.IOException e){ e.printStackTrace(); } // or
    catch(java.lang.InterruptedException e){ System.out.println(e); }
  } }
/* Each character appears on screen with about 1/10 second delay. */
/*
C:\WINDOWS\JavaCodeFiles\Basics2\StreamsPipes>javac HelloStringReaderStream.java

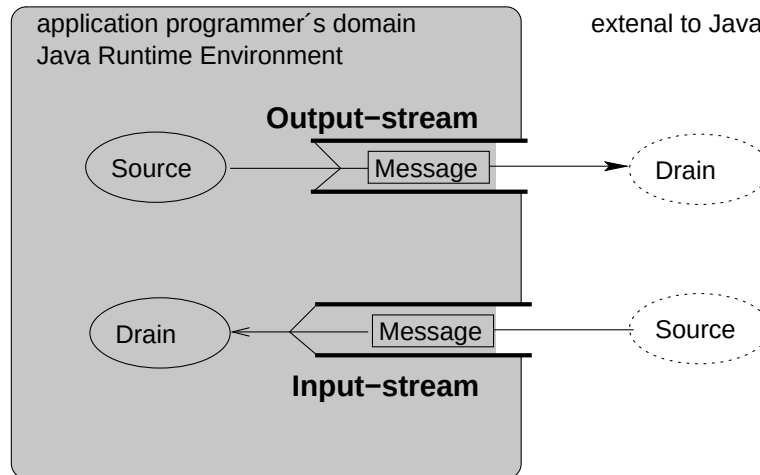
C:\WINDOWS\JavaCodeFiles\Basics2\StreamsPipes>java HelloStringReaderStream
Hello StringReader stream
C:\WINDOWS\JavaCodeFiles\Basics2\StreamsPipes>
*/
```

The stream lets the `read()`-method return the string, character by character, in fifo-manner (first-in first-out, page 233). Here can be seen the fifo-buffer aspect of the stream. And observe the stream's output, given as an integer-value, which has to be reformatted to a suitable character-representation. Using a Java-stream, means getting information out only sequentially by using the `read()`-method. (Breaking down a string to its characters also can be accomplished by using the `java.lang.String.toCharArray()`-method; the resulting array gives random-access to each character-position of the string.)

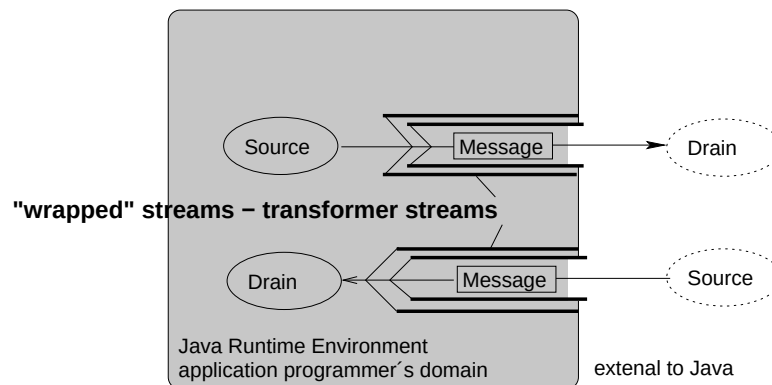
Next to being classified into input- and output-(“half”)-streams, Java-streams can be recognized as representing two basic kinds: Streams connected directly to an external channel of communication of the given

I/O-streams

computer, so-called **I/O-streams (Input/Output-streams)**, these streams mostly furnish or take simple sequences of bytes or characters.



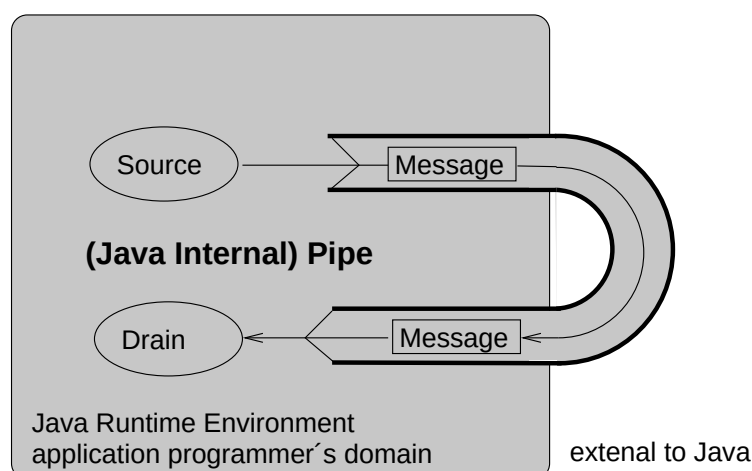
And then there are those streams that can encapsulate or wrap I/O-streams, so that the programmer receives or sends data of specific types or classes or the data is monitored, decompressed, decoded or reformatted to suit a special purpose. Those streams are called **format-** or **filter-** or **wrapper-streams**.



Streams can be multiply wrapped, in the graphic above input- and output-streams have been wrapped.

pipe

Streams where input and output are within the same process, and can be fully configured by the Java programmer, are so-called **pipable streams** or **pipes**. Such streams have to be constructed from two Java half-streams, one input- and one output-stream, which have to be connected:

stream ≠
variable

So, a stream having its input and its output within the same environment (process, runtime-environment) is called pipe.

Streams are something different than variables! Both hold data, but data-input or -output with variables is done by assignment (=) whereas streams usually act like fifo-aggregations, their data-elements can be only read and written successively using `read()`- and `write()`-methods. (fifo = first-in-first-out, page 233). The state of a variable usually is fully defined, whereas especially input-streams may furnish their data time-delayed and often cannot foresee the end of the data they deliver.

In Java, streams are given as class-members by references to (stream-)objects instantiated from (stream-)classes. A stream-object in Java exemplifies a **fifo-container** that **either can be written to or can be read from, byte for byte** (or two-byte-token for two-byte-token). Communication by Java stream-objects means **one-way-communication**. Two-way-communication can be realized by working with two streams of opposing direction.

Mostly streams, many of them may be user-defined, act as **filters or transformers for data-streams**. For example, they may transform an incoming stream of bytes into an object of a certain class. So, the message aspect of streams in Java is supplemented by various additional functionalities. Streams in Java still are centered around a data-content, but the stream-object often furnishes additional methods to reassess that data (transform, filter).

input-stream

A stream in Java is an object that has its **input- and output-channels generally within different threads** or processes. For the Java programmer, streams usually present themselves as **an object of input or output**. A (wrapped) **input-stream** often receives data-messages from some I/O device. A (wrapped) **output-stream** in Java is an object that sends data-messages to some drain, usually outside the reach of the programmer.

output-stream

The so-called **buffered streams** in Java are fifo-containers (first-in-first-out) whose data-content can be read or written blockwise, meaning in entire blocks of bytes, not byte-wise. This usually eases programming and saves resources, because calling a `read()`-method explicitly to get one- or two bytes, byte by byte, takes more time than calling its block-counterpart once and getting a buffer-array filled with bytes.

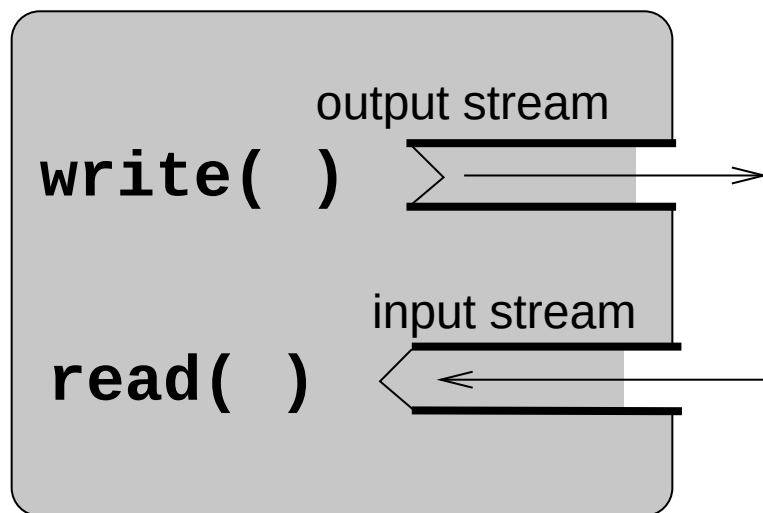
buffered
stream

Java: Streams

Java: Streams — The terminal-command “java *fileName*” activates a Java application-program¹²³, as introduced on page 127. The java-tool has the option of postfixing the terminal-command with space-separated strings for further processing that **array** of strings within the block of the triggerable `main()`-method (pages 72 and 236). This option demonstrates a one-way-communication between the end-user’s keyboard and the Java software, written by the Java application-programmer.

The `System.out.println()`-method opens a one-way channel for sending information onto the computer’s screen. In that command the identifier `out` references a **stream**-object. — So, arrays and streams have been always used implicitly for input- and output purposes.

Reading from input-streams and writing to output-streams is done by `read()`- and `write()`-instance-methods of the stream-objects:

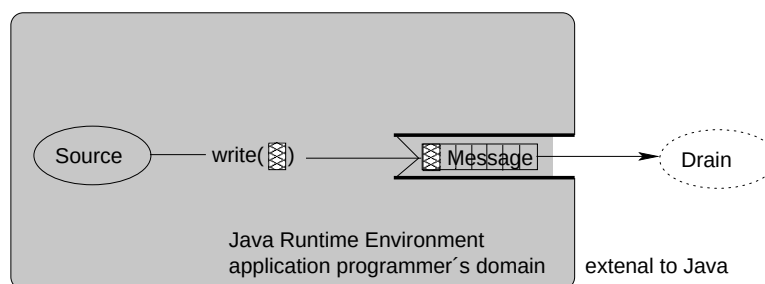


The following presents just a coarse classification of the stream-classes furnished by the `java.io` package. Implemented and functional stream-objects are presented in the next section. Due to their unidirectionality, the streams in Java are divided into input- and output-streams. **Input-streams** import data into the current programming-context. **Output-streams** devour data coming from the current programming-context, the Java Runtime Environment (JRE).

(output-stream) Usually a byte or a character (two bytes) are sent as single units by calling the method `write()`. Buffered streams wait until a block of data is filled before sending it; the method `flush()` sends also a partially filled block of data.

output-stream -
`write()`

Writing to Output-Streams



¹²³This command activates the Java byte-code interpreter to load the file `fileName.class` and to run the byte-code contained therein. The Java byte-code interpreter implements the Java Virtual Machine specification. The Java byte-code interpreter represents the core of each Java Runtime Environment (JRE).

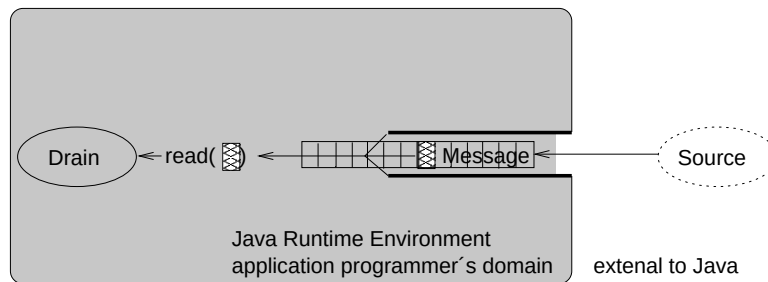
As can be seen above, the denotation “output” has been chosen according to the current environment, the analogous “Input”-case is presented below:

input-stream -
read()

(input-stream) To assess whether the input-stream is empty, there are either the **available()**-method (for byte-units), or the **ready()**-method (for two-byte characters). The **skip(long n)**-method discards *n* bytes. In those implementations of Java which let the **markSupported()**-method return the boolean value **true**, reading may not automatically discard the data in the stream-object; in these cases, a unit of data may be marked with a **mark(int readLimit)**-method and, later, the input-stream can be **reset()** to that marked position. (The integer value **int** tells after how many bytes the mark has to be discarded — a buffer-size in byte-units.)

If the standard **read()**-method returns the integer-value **-1**, then the end of the data-stream has been reached.

Reading from Input-Streams



All **InputStreams** in Java have a **read()**-method to get the data furnished by the stream. Data is written into the **OutputStreams** by calling a predefined **write()**-method. This convention is useful when memorized, maybe with the help of the graphic on page 295.

one- and
two-byte units

Streams in Java are also classified according to the informational units they transfer: **JAVA-STREAMS WITH NAMES SUFFIXED WITH Stream TRANSFER ONE-BYTE UNITS**, whereas **JAVA-STREAMS WITH NAMES SUFFIXED WITH Reader OR Writer TRANSFER TWO-BYTE UNITS**. The basic stream-conventions in Java are given as part of the **java.io**-package:

data -unit	ABSTRACT INput-stream-class (read)	ABSTRACT OUTput-stream-class (write)
byte/ 2byte	InputStream reads bytes Reader reads char	OutputStream writes bytes Writer writes char
	int read() returns the single byte read from stream as int	void write(int i)
	int read(byte[] b) fills “b” with streamed bytes	void write(byte[] b) writes “b” fifo-like into stream
	int read(byte[] b, int offset, int length) read into “b”	void write(byte[] b, int offset, int length) write from “b”
	available() or ready() (with Reader)	
	skip(long n)	
	markSupported()	
	mark(int readLimit)	flush()
	reset()	
	close()	close()

As can be seen above, scrutinizing the **read()**- and **write()**-methods in the table, single bytes sometimes are coded as **int**-values. Retrieving or encoding single bytes is done simply by casting to or from a byte-value (**b = (byte)i**; or **i = (int)b**). The same procedure becomes necessary when considering **Reader** or **Writer**-streams that transfer two-byte entities, being read and written as variables of primitive-type **char**.

All stream-classes are sub-classes of one of these four basic streams; all those extended classes have properly overridden **read()**- and **write()**-methods. The four kinds of streams above are given by abstract classes; sub-classes with stream-characteristics have to implement especially the **int read()** or **write(int i)**-methods; the overloaded methods use these basic method-versions. In **Reader**- and **Writer**-classes, the arrays, denoted “b” above, naturally have array-elements of type **char**. The **Writer**-class furnishes also a **write(String s)**-method. For the full specification of these classes see the Java Platform API Specification [JPAPIS]. (The above constitutes a highly non-trivial example of the use of abstract classes, as referencable on page 183.)

Java: Runtime Input/Output-Streams

Java: Runtime Input/Output-Streams — Java-streams managing actual input/output-procedures with respect to the Java Runtime Environment (JRE) are:

java.lang.System.in stream-OBJECT that stores and relays keyboard input

java.lang.System.out stream-OBJECT that takes textual output onto some screen-window

The basic I/O-stream-CLASSES of Java are:

java.io.FileOutputStream instantiated instances write data into a file accessible by a path of the Hierarchical File System (HFS)

java.io.FileInputStream instantiated instances read data from a file accessible by a path of the Hierarchical File System (HFS)

Later there are presented streams which allow inter-process- and inter-computer-communication:

socketName.getOutputStream()

socketName.getInputStream() A `java.net.Socket`'s `get...Stream()`-factory-method¹²⁴ furnishes a stream-object for unidirectionally communicating with some other process or some other computer (the first example of that kind is given on page 384).

So, in Java there are very few basic means to communicate with the outside world! For more on computer-to-computer (network-)communication or computer with end-user-communication see the similarly named, remaining parts of this text.

The following program demonstrates the use of the Java Runtime Environment's in and out-streams:

```
class SystemIn
{ public static void main( String unused[] )
  { System.out.println(
    "enter typed key-sequence with 'return'\nend program by typing \"#\";");
    int inKey = 0;
    try
    { do
      { System.in.skip( System.in.available() );
        // the line above discards any remaining element in the stream.
        inKey = System.in.read();
        System.out.println(
          "read: " + (char)inKey + " having ASCII/Unicode no. " + inKey );
      }while( inKey != 35);
      // above: A "#" at the beginning of the input ends the pre-condition loop.
    }catch(java.io.IOException e){ e.printStackTrace(); }
    System.out.print( "Remaining elements in the stream: " );
    try
    { while( System.in.available() != 0 ){ System.out.print( (char)System.in.read() ); }
    }catch(java.io.IOException e){ e.printStackTrace(); }
  } }

/*
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > javac SystemIn.java
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > java SystemIn
enter typed key-sequence with 'return'
end program by typing "#"
a different keyboard input
read: a having ASCII/Unicode no. 97
zzz...
read: z having ASCII/Unicode no. 122
#
read: # having ASCII/Unicode no. 35
Remaining elements in the stream:
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes >
*/
```

Especially this source-code gives an example of the usefulness of pre- and post-condition loops, which were mentioned on pages 108 and 110.

¹²⁴Methods like the `get...Stream()`-method instantiate an object and therefore are also called factory-methods (see also page 151).

Remember that the Hierarchical File System (HFS, visualized on page 47) makes it possible to address data/sequences of bytes/files without addressing explicitly the hard-drive, the CD-ROM or any other storage device. Remember also that access to information about the file-system is given by the package with the fully-qualified identifier `java.io.File` (page 259). When being able to locate files, then reading those files becomes possible with methods furnished by the class `java.io.FileInputStream` below: Here the addressing of files is supplemented by the method of how to retrieve files from devices within the reach of the Hierarchical File System. Thus, reading and writing files is done by stream-objects that are instances of (sub-)classes like:

data-unit	INput-stream-class (read)	OUTput-stream-class (write)
byte/ 2byte	FileInputStream FileReader	FileOutputStream FileWriter
	File ... (File f)	File ... (File f)
	File ... (FileDescriptor fd)	File ... (FileDescriptor fd)
	File ... (String s)	File ... (String s)
		File ... (String s, boolean)

These classes already have been implemented and furnish fully-functional `read()`- and `write()`-methods. The following program demonstrates the use of Java file-streams:

```
public class FileInputStream
{ public static void main( String commandLineInputs[] )
  { /* File-name relative to what directory?
    This class's byte-code's directory! (=current directory)
    */
    String fileName = null;
    // null because otherwise a compile-time error "variable fileName may not be initialized"
    try
    { fileName = commandLineInputs[0];
      java.io.FileInputStream aStream = new java.io.FileInputStream( fileName );
      System.out.println( "file " + fileName + " found and ready to read." );
      try
      { int data;
        do
        { data = aStream.read();
          System.out.print( (char)data );
        }while( data != -1 );
      }catch(java.io.IOException e){ System.out.println(e); }
      finally{ System.out.println( "\nFile has been loaded or not" ); }
    }
    catch( java.io.FileNotFoundException e )
    { System.out.println( "file " + fileName + " NOT found or unreadable." ); }
    catch(java.lang.ArrayIndexOutOfBoundsException e)
    { System.out.println( "insert file-name when starting the application." ); }
    }
    finally{ System.out.println( "Filecheck ended " ); }
  } }

/*
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > javac FileInputStream.java
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > java FileInputStream FileInputStream.java
file FileInputStream.java found and ready to read.
... this listing ...
File has been loaded or not
Filecheck ended
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes >
*/
```

The source-code above also gives a concrete example of the use of try-catch-clauses (past page 283). More economically, the reading of entire lines, instead of single bytes, is demonstrated within the source-code on page 456.

The following program saves a string onto a persistent storage-device and can retrieve it, even when re-run within another Java Runtime Environment during another computer-session. Here can be seen how an object-output-stream wraps a file-output-stream resulting in a stream that stores a Java-object persistently!

```
class FileInputOutput // create a file and save it
{ public static void main( String unused[] )
  { String stringObject = new String( "data to be saved" );
    try
    { java.io.FileOutputStream fos = new java.io.FileOutputStream( "fileName" );
```

```

        java.io.ObjectOutputStream oos = new java.io.ObjectOutputStream( fos );
        oos.writeObject( stringObject );
        // append some numbers:
        for( int i = 0; i < 10; i++ ){ oos.writeInt( i ); };
        oos.flush();
    }
    catch(java.io.IOException e){ e.printStackTrace(); }
    // (... or declared in throws-clause of this method)
    // here the writing has been completed; now read:
    try
    { java.io.ObjectInputStream ois =
      new java.io.ObjectInputStream( new java.io.FileInputStream( "fileName" ) );
      String readString = (String)ois.readObject();
      System.out.println( readString );
      for(int i = 0; i < 10; i++ )
      { double readNumber = ois.readInt();
        // above: written as 'int', READ as 'double' by autocasting!
        System.out.print( " " + readNumber );
      } }
    /*catch(java.io.FileNotFoundException e){System.out.println(e);}
    catch(java.io.StreamCorruptedException e){System.out.println(e);}
    catch(java.io.OptionalDataException e){System.out.println(e);}
    catch(java.io.IOException e){System.out.println(e);}
    catch(java.lang.ClassNotFoundException e){System.out.println(e);}*/
    catch(java.lang.Exception e){System.out.println(e);} // indiscriminate handling
  } }
  /* in the current directory appears a file with the name "fileName" */

  /*
  andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > javac FileInputOutput.java
  andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > java FileInputOutput
  data to be saved
  0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes >
  andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes >
  */

  // test with reading 20 integers, getting a java.io.EOFException (End of File)
  // test with reading "fileNameI", getting a java.io.FileNotFoundException

```

Observe the list of discernable exceptions and the possibility to handle each one of them differently. All these catch-clauses can be omitted, as in the case above has been shown: Outcommenting them and adding only one single catch-clause for exceptions having the class-type of the mutual super-class `java.lang.Exception`. — The classes `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` are introduced later on page 308.

Another way of storing data persistently in files are the so-called random-access-files (in Java referencable with identifiers of type `java.io.RandomAccessFile`). They can be used to aggregate data and access it relatively easy by positioning a pointer (counting bytes chime-like, this means beginning to count with zero). Random-access-filing already has been indicated on page 238. Random-access-filing works without explicit streaming:

```

class RandomAccessFile
{ public static void main( String unused[] )
  { try
    { java.io.RandomAccessFile raf =
      new java.io.RandomAccessFile( "fileName.raf", "rw" );
      // fileName either String-object or java.io.File-object
      // "r" for read only, but then file already should exist
      // "rw" for read and write, generate new file if non-existent.
      // standard DataInputStream / DataOutputStream methods plus:
      raf.writeChars
      ( "A string, stored as a sequence of two-byte-units, the high byte first!"
      );
      System.out.println( "Length of file in bytes: " + raf.length() );
      raf.close();
    }catch(java.lang.Exception e){ e.printStackTrace(); };

    try
    { java.io.RandomAccessFile rafRead =
      new java.io.RandomAccessFile( "fileName.raf", "r" );
      System.out.print
      ( rafRead.readChar() + "\t 2 byte Character read, new pointer-position: " +
        rafRead.getFilePointer() + "\n" +
        rafRead.readInt() + "\t 4 byte integer read, new pointer-position: " +
        rafRead.getFilePointer() + "\n"
      );
      rafRead.seek(10);
    }
  }
}

```

```
// above means position pointer for reading byte number 10, means 6th character ("i")
System.out.print
( rafRead.readChar() + "\t 2 byte Character read, new pointer-position: " +
  rafRead.getFilePointer() + "\n"
);
rafRead.close();
} catch (java.lang.Exception e) { e.printStackTrace(); }
} }

/*
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > javac RandomAccessFile.java
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > java RandomAccessFile
Length of file in bytes: 140
A      2 byte Character read, new pointer-position: 2
2097267 4 byte integer read, new pointer-position: 6
i      2 byte Character read, new pointer-position: 12
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes >
*/
```

Deleting such a random-access-file is possible with the method `java.io.File.delete()` (page 259). Furthermore, any Java-class documentation shows the class `java.io.RandomAccessFile` to be implementing the interfaces `java.io.DataInput` and `java.io.DataOutput` (more on these interfaces is said around page 308)¹²⁵.

Java: Piping with Streams

Java: Piping with Streams — In Java, piped streams are connected pre-written halves of pipes: A piped input-stream must feed a piped output-stream as demonstrated in the graphic on page 292.

data-unit	INput-stream-class (read)	OUTput-stream-class (write)
byte/ 2byte	PipedInputStream PipedReader	PipedOutputStream PipedWriter
	Piped...()	Piped...()
	Piped...(PipedOutputStream or PipedWriter)	Piped...(PipedInputStream or PipedReader)
		flush()
	connect(PipedOutputStream or PipedWriter)	connect(PipedInputStream or PipedReader)
	close()	close()

For what are piped streams useful? – For implementing a fifo-buffer (fifo = first-in-first-out). – For sending data between threads within the same Java Runtime Environment. – Furthermore, changing a pipe to an I/O-stream is easier than rewriting a program to support I/O-streams. Especially when developing software, piped streams return information to the programmer; for example instead of sending the data out into the network.

The `InputStream`, `Reader`, `OutputStream` and `Writer` are the basic abstract stream-types that are extended to the classes above. All these classes instantiate I/O-streams, or in the case of a piped Java stream a complete pipe:

```
class Pipe
{ public static void main( String unused[] )
{ java.io.PipedWriter pipeOut;
  java.io.PipedReader pipeIn;
  java.io.BufferedReader pipeBuffer;
  try
  { System.out.println( "Installing the pipes ..." );
    pipeOut = new java.io.PipedWriter(); //>>>>----\ pipe-out
    pipeIn = new java.io.PipedReader( pipeOut ); //<<<<----/ pipe-in
    // wraps pipe-out
    pipeBuffer = new java.io.BufferedReader( pipeIn ); //<<<<----<<<< wrap
    System.out.println( "Sending and converting strings of characters ..." );
    pipeOut.write( "First text to be piped.\n" ); // newline character separates text-data
    //pipeOut.flush(); // unnecessary
    pipeOut.write( "Second text to be piped.\n" );
    System.out.println
    ( "Read single character (coded as int): " + pipeIn.read() +
      "\nRead the rest of the line: " + pipeBuffer.readLine() +
      "\nRead the second line: " + pipeBuffer.readLine()
    );
  }
  catch (Exception e) { e.printStackTrace(); }
}
```

¹²⁵For more on random-access-files see also the Java TechTip of May 5th, 2000.


```

    });
    }catch(java.io.IOException e){ e.printStackTrace(); }
} }

/*
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > javac Pipe.java
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > java Pipe
Installing the pipes ...
Sending and converting strings of characters ...
Read single character (coded as int): 70
Read the rest of the line: first text to be piped.
Read the second line: Second text to be piped.
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes >
*/

```

Pipes are often used as data-buffers between threads¹²⁶. The class `java.io.BufferedReader` has been anticipated here, it will be introduced in the section past the next.

Java: More on Java Streams

Java: More on Java Streams — When using streams for exchanging textual data use (extensions of) `Reader/Writer`-streams (universal Unicode-encoding). Only when transferring non-textual data (compressed text-files, video- or audio-data) then use objects derived from (extensions of) classes `InputStream/OutputStream`.

Reading can done token-wise or array-wise, whereby the length of the array is given by the source-code that implements the input-stream. If there are less tokens¹²⁷ available than the array needs to be filled, then the array is only partially filled and the end of the data is signaled by an entry of value minus one (−1).

As already mentioned, Java stream-classes can be classified as belonging to different groups:

- Output-streams that wrap output-streams (example on page 298).
- Output-streams which do not wrap other output-streams. These are output-streams that write to a non-stream resource, which may be located outside or inside the Java Runtime Environment (JRE). Such stream-objects are given by objects like `System.out` or by classes like `java.io.PrintStream`. (Each time a string is printed into the terminal-window the stream-object `System.out` has been used.)
- Input-streams that wrap input-streams (example on page 298).
- Input-streams that do not wrap other input-streams. Such input-streams read from a non-stream resource, which may be located outside or inside the Java Runtime Environment (JRE). Stream-objects of that kind are given by objects like `System.in` or by classes like `java.io.StringReader` (example on page 291).

There are other more or less stream-related classes like

- (PIPES) Output-streams that wrap input-streams and vice-versa. (The latest piece of source-code demonstrated the usage of pipes.)
- (STREAM TO NON-STREAM) Streams which take stream-data but transform it into non-stream formats, like the `java.io.StreamTokenizer` with its `nextToken()`-method¹²⁸, that returns an `int`-value coding the type of the token. According to the type information, the value of the current token can be found either in the `sval`-variable (class-type `String`) or in the `nval`-variable (primitive-type `double`) of the stream-object. (no example in this text)
- (HARDWARE-INTERFACES) Instances of the class-type `java.io.RandomAccess` (page 299) and of the class-type `java.net.Socket` (beginning with page 382) encapsulate input- and output-streams.

¹²⁶For more see the Java TechTip of February 8th, 2001.

¹²⁷A token may be a byte in the case of objects of type `java.io.InputStream` or `java.io.OutputStream`. But a token may also be a two-byte entity of primitive-type `char` in the case of streams of class-type `java.io.Reader` or `java.io.Writer`.

¹²⁸Not to be confused with the class `java.util.StringTokenizer`. Instances of this class take a `String`-object and break it - word by word - into a sequence of substrings; then each call of the `nextToken()`-method returns one substring. The substrings are returned in the order of their occurrence. (Not to be confused with the class `java.io.StringReader`, whose objects read single two-byte characters sequentially from a given `String`-object, as introduced on page 291.)

Java-streams reading or writing to non-stream resources. How can those streams be identified? The stream-objects may be given as members of a class (`System.out`, `System.in`) or the stream-objects are given by the factory-method (`java.net.Socket.getInputStream()`; or by the factory-method `java.net.Socket.getOutputStream()`). Or the constructor of the stream-class takes references of non-stream type: Consider for example, the constructor of the `java.io.StringReader`-class, `StringReader( String )`.

Piped stream-classes. How can they be identified in Java? By considering their constructors: Their constructors use stream-arguments of counterdirection: The constructor of a piped output-stream has an input-stream as its argument and vice-versa. Piped streams in Java usually have a name containing the substring “Pipe”. In Java, piping is done only with specially designated streams.

Within applications, working with streams usually means to wrap standard streams multiply, thereby chaining them until getting the resultant stream which actually produces the desired result. Writing own stream-classes from scratch gives another way of producing wrapable streams for custom-written streams: Usually this means to extend classes of type **java.io.FilterInputStream** (or **FilterOutputStream**), because this kind of class already implements more methods than their abstract super-class: For example, compare the class `java.io.FilterReader` with the abstract class `java.io.Reader`: The class `FilterReader` has already implemented the methods `read()` and `close()`, which have been declared abstract in `Reader`. Below, fragments of source-code indicate the process of properly overriding the `read()`- and `write()`-methods of filter-streams:

FilterOutputStream-Classes:

FilterOutputStream-Classes: — Anticipating a wrapping procedure, consider a given instance of type `OutputStream` referenced with the identifier `out`, then standard methods of overriding a `FilterOutputStream` are: (for a `FilterWriter` substitute the primitive-type “byte” by “char”)

```
public void write( int b ) throws java.io.IOException
    // b for byte (given as an int!)
{ preprocess( b ); // manipulating, for example filtering
  out.write( b );
  postprocess( b ); // analyzing or logging
}
public void write( byte[] ba ) throws java.io.IOException
    // ba for byte-array
{ preprocess( ba ); // manipulating, for example filtering
  out.write( ba, 0, ba.length );
  postprocess( b ); // analyzing or logging
}
public void write( byte[] ba, int off, int len ) throws java.io.IOException
    // ba for byte-array
{ preprocess( ba ); // manipulating, for example filtering
  out.write( ba, off, len );
  postprocess( ba ); // analyzing or logging
}
```

The functionality of the newly defined filter-stream is encapsulated in the various methods called `preprocess()` and `postprocess()`.

FilterInputStream-Classes:

FilterInputStream-Classes: — Take for example the stream-characterizing methods of an extended `FilterInputStream` (for a `FilterReader` substitute the primitive-type “byte” by “char”), the stream wraps an `InputStream`-object referenced by the identifier `in`:

```
public int read() throws java.io.IOException
{ int b = in.read();
  if( b != -1 ){ analyzeOrManipulate( b ); }
  return b;
}
```

```

/*
--- Usually the read()-methods below are much faster,
    than reading multiply single bytes.
--- The dimension of the array determines the chunks of data
    to be read out of the stream.
--- If the length of the array is larger than the (currently) available
    number of tokens, then the last token codes the -1.
*/
public int read( byte[] ba, int off, int len ) throws java.io.IOException
{ len = in.read( ba, 0, ba.length );
  if( len != -1 ){ analyzeOrManipulate( ba ); }
  return len;
}
public int read( byte[] ba, int off, int len ) throws java.io.IOException
{ len = in.read( ba, off, len );
  if( len != -1 ){ analyzeOrManipulate( ba ); }
  return len;
}

```

Some other stream-characterizing methods are:

skip(long n) as the name implies, this method allows to skip over tokens of the stream.

mark(int readLimit) and reset() These methods are implemented within the current Java Runtime System, ONLY when a call of the boolean-valued `markSupported()`-method returns `true`: So, only if implemented, the `mark()`-method marks the current-stream token and the current stream-state. Then, the following stream-tokens may be read as usual, for example for some inspection; a call of the `reset()`-method re-establishes the stream-state existent at the time the `mark()`-method had been called. The `int`-argument of this method tells how many bytes of the stream can be maximally memorized, it controls the size of the associated buffer.

InputStream.available() tells how many bytes are available to be read before the next call of the `read()`-method blocks.

Reader.ready() like the `available()`-method, except that it doesn't tell about the number of tokens available.

OutputStream./Writer.flush() writes out any buffered tokens to the destination, file or socket.

A call of a `write()`-method blocks when the stream's (internal) buffer is full. A call of a `read()`-method blocks when the associated stream's buffer is empty. As a general rule, avoid transformations like stream-nonstream-stream; whenever using streams, aspire to let as much as possible of the work be done in the stream context without changing stream and non-stream contexts.

Java: Filtering and Formatting with Streams

Java: Filtering and Formatting with Streams — The Java stream-classes which are given by the four identifiers `java.io.FilterInputStream`, `java.io.FilterReader`, `java.io.FilterOutputStream` and `java.io.FilterWriter` extend the standard streams minimally and mark them as filter-streams. Such filter-streams are not meant to be streams with an autonomous input/output functionality. Streams derived from so-called filtered-stream-classes are to be “wrapped around” other streams. Additionally, there are pre-programmed filter-like, wrapable streams that are not derived from any special filter-stream.

For example, there are two special stream-extensions of the `java.io.Reader`-stream-class and respectively the `java.io.Writer`-stream-class; streams instantiated from these classes transform the one-byte- and two-byte formats:

data-unit	INput-stream-class (read)	OUTput-stream-class (write)
byte to 2byte	InputStreamReader reads bytes, wraps characters extends Reader	OutputStreamWriter writes characters, wraps bytes extends Writer
	InputStreamReader(InputStream s)	OutputStreamWriter(OutputStream s)
	InputStreamReader(InputStream s, String encode)	OutputStreamWriter(OutputStream s, String encode)
	String getEncoding()	String getEncoding()
	close()	

Variants and names of some of the encodings have been listed on page 266 or just before that referenced page.

All the Java-streams, presented so far, read or write one- or two-bytes entities at a time. Blockwise reading from input-streams or blockwise writing to output-streams can be done with the so-called buffered¹²⁹ streams (BufferedReader extends Reader, whereas BufferedInputStream extends FilteredInputStream, which in turn extends InputStream):

data-unit	INput-stream-class (read)	OUTput-stream-class (write)
byte/ 2byte	BufferedInputStream BufferedReader	BufferedOutputStream BufferedWriter
	Buffered ... (... s)	Buffered ... (... s)
	Buffered ... (... s, int bufferSize)	Buffered ... (... s, int bufferSize)
		newLine() (for Writer writes a "line.separator")
		flush() premature block-fragment is dispatched
	close()	close()
	String readLine() (only with Reader: reads a line of text)	

("line.separator" denotes a system-property that codes the newline-command, in Java given by the string "\n") Buffering the data means transferring the data blockwise and thus more efficiently. An example of how to apply such streams already has been given in the latest piece of source-code which introduced a pipe. The source-code below shows a pipe, multiply wrapped with filter-streams:

```

/* main()-method in class FilterStream */
/* First introduce a custom-written stream:
   --- to change or to establish the stream's functionality override the read()-methods
       according to the convention specified in the documentation
       of the java.io.FilterReader-class, or on page 302.
       See for (*) in this piece of source-code.
*/
/*
This programs has five options for reading from an input-stream:
1. read directly from input-stream
2. read from a filter, a sieve, wrapped around the input-stream (finds character 'e')
3. delayed, read from buffered output, wrapped around the input-stream
4. delayed, read from sieve, wrapped around the input-stream (finds character 'e')
5. delayed, read from buffered sieve, wrapped around the input-
   stream (finds character 'e')
*/
class SieveReader extends java.io.FilterReader
{ char mesh;
  public SieveReader( char m, java.io.Reader r)
  { super(r); mesh = m;
    } // if super(r) is outcommented, 'cannot resolve symbol' error at compile-time
  public int read() throws java.io.IOException // (*)
  { int i = super.read();
    if( i == (int)mesh )
    { report( "\nfound " + mesh + " with read() " ); return 0; } // omit mesh-character
    else return i; // returns Unicode-number of character read (*)
  }
  public int read( char[] charray ) throws java.io.IOException // (*)
  { return read( charray, 0, charray.length );
  }
  public int read( char[] charray, int offset, int length )
  throws java.io.IOException // (*)

```

¹²⁹Java input-streams store the incoming data anyway like a fifo-container for bytes (fifo = first-in-first-out). Here a buffered stream means just that these stored bytes can be read or written blockwise.

```

    { report( "\n array[0.." + chararray.length + "]" );
      super.read( chararray, offset, length ); // reads in the data!
      return analyze( chararray ); // method returns number of characters read (*)
    }
    private int analyze( char chararray[] )
    { for( int i = 0; i < chararray.length; i++ )
      { System.out.print( chararray[ i ] );
        if( chararray[i] == 0 ) break;
        // above: end of data, substitutes the function InputStream.available()
        if( chararray[i] == mesh )
          report( "\nfound " + mesh + " from read(.) at position " + i );
        chararray[i] = 0; // omit
      }
      return chararray.length; // method returns number of characters read (*)
    }
    private void report( String s ){ System.out.print( s ); }
    /* public void close() throws java.io.IOException{ super.close(); }
    would be necessary if abstract java.io.Reader was extended;
    unnecessary with non-abstract java.io.FilterReader
    */
  }
  class FilterStream
  { private static int counter = 0;
    private static void listArray( char array[] )
    { for( int c = 0; c < array.length; c++) System.out.print( array[c] );
    }
    public static void main( String integer[] )
    { int source;
      char mesh = 'e';
      char array[] = new char[1024];
      /* default length of array, may be substituted by buffered streams,
       / conveniently chosen to be a power of the number '2' usually 1024.
       / The length of the array determines the size of the
       / data packets attempted to be read from the stream.
      */

      try
      { java.io.PipedWriter charOut = new java.io.PipedWriter();
        // connect two half pipes to a pipe:
        java.io.PipedReader charIn = new java.io.PipedReader( charOut );
        // and continue wrapping streams:
        SieveReader sieve = new SieveReader ( mesh, charIn );
        java.io.BufferedReader bufferedSieve = new java.io.BufferedReader( sieve );
        /*
        * The following branch does not imply that reading
        * can be done from both branches at the same time,
        * otherwise one branch loses information to the other!
        */
        java.io.BufferedReader bufferedCharIn = new java.io.BufferedReader( charIn );
        // Action:
        while( true ) // a continuous stream of characters
        { source = (int)( 33+93*Math.random() );
          counter++;
          // if( counter == 5000 ) charOut.close();
          // above: throws exception in the 5000th cycle
          charOut.write( source ); // range 33 .. 126
          switch( Integer.parseInt( integer[0] ) )
          { case 1: System.out.print("[ " + counter + " " + (char)source + "> " );
              System.out.print( " >" + (char)charIn.read() + "]" ); break;
            // tests the immediate echo thus does not block
            case 2: System.out.print("[ " + counter + " " + (char)source + "> " );
              System.out.print( " >" + (char)sieve.read() + "]" ); break;
            case 3: System.out.print("[ " + counter + " " + (char)source + "> " );
              if( counter%100 == 0 )
                { System.out.println( " read buf>> " +
                  bufferedCharIn.read(array) + ": " );
                  listArray( array ); System.out.print( " ] " );
                } break;
            case 4: if( counter%100 == 0 )
              System.out.print( "\n" + counter + " read sieve>>" +
                sieve.read(array) + "]" ); break;
            case 5: if( counter%100 == 0 )
              System.out.print
                ( "\n" + counter + " read sievebuf>> " +
                  bufferedSieve.read(array) + "]" ); break;
          } // case
        } // while
      } catch( java.io.IOException e){ e.printStackTrace(); }
      //catch( Exception e ){ e.printStackTrace(); }
    }
  }

```

```

} }

/*
andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > java FilterStream 2
[1 U> >U] ... [84 a> >a] [85 e>
found e with read() >] [86 >> >>] ....

andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > java FilterStream 3
[1 ,> [2 L> [3 &> [4 %> .... [97 X> [98 a> [99 '> [100 <> read buf>> 100:
,L&2NSm@EY'p^W-Z1PxG,T[r=kBu{z0R%-p3["-h!9A8"6_z63$XB>N2*Ai!^!<aOPt<$liTabZ.f'KY={oB5He^PH=0[<|Xa'< ] [101 ->
[102 {> [103 s> [104 {> ....

andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > java FilterStream 4

array[0..1024] (#==r68MHnMhnl9qm(/,9K)?f9t'0/0|{%viXRs|tQc@x;#)+w4Xx%oJu0|<^X%;kWlbKe
found e from read(.) at position 69z6/]a/$<0JQNh'I!jc;_>'r?6='KEs
100 read sieve>>1024]
array[0..1024]LB0)A?]{8NMDxswX'A)BPf>d.<d,UjJSPMgg+{[4N]xp/M+H)VoD; }6Grhi#F1kmU.t0nwdjdr>61o<x(3J&imK57?tR\"Bm2#jZ
200 read sieve>>1024]
array[0..1024],J8WUB,SjqLQ0MSd8n$Vv9uks-BW882fYk](z8jyEN=OX0>mz*Kh{-51nZ2SqY$Ywd5BA/+K'e
found e from read(.) at position 732VK@v.6c^z}WH&Rxn.bw/..!,A7
300 read sieve>>1024]
array[0..1024]sRwstsD[Fc:IwW=QVJ'!e
found e from read(.) at position 20@0aCa%mpXqg{t"*Q#MbTY")#P>TV$e
found e from read(.) at position 50%'F\p5x'0'Ee
found e from read(.) at position 62n);@K)c06cG190J^ghhA=!x?<;If^==v3mxES
400 ....

andreadipietro@linux:~/JavaCodeFiles/Basics2/StreamsPipes > java FilterStream 5

array[0..8192]B-ESdF9_e
found e from read(.) at position 8"3ke
found e from read(.) at position 12K/P5kIhY90Y^ZP,'80LUk_kzL4q;h9g,>)1DCiBY?b=x{e
found e from read(.) at position 58x4KRy/s?S^TOHCnU<e
found e from read(.) at position 76pe
found e from read(.) at position 78ZCPDn?e
found e from read(.) at position 85)d\Wmr>$Mmu_Ed
100 read sievebuf>> 1024]
200 read sievebuf>> 1024]
*/

```

The call of the `read()`-method gives an implicit call within the wrapped streams above, another triggerable method!

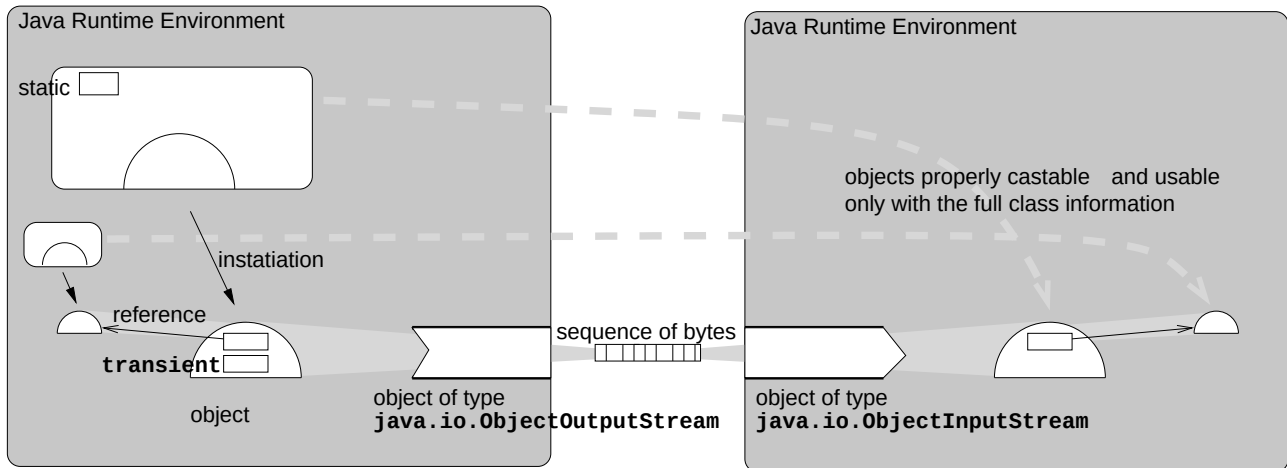
Java: Serialization of Objects, Persistence

Java: Serialization of Objects, Persistence — Simple data-types (among them primitive-types) and entire object-graphs (objects together with their referenced objects) can be sent through Java-streams as sequences of bytes.

Java classes and interfaces are described by writing source-code, and saved as text-files on persistent storage-devices like a hard-disk. The Java compiler reads that source-code, transforms it into byte-code and stores the result in so-called class-files. Class-loaders within the Java Runtime Environment (JRE) read the byte-code of class-files and transform the byte-code into a form (“loaded class”) accessible to the Java byte-code interpreter of the Java Virtual Machine. From those loaded-classes can be instantiated objects within the Java Runtime Environment (More on this can be found beginning with page 433). How about saving an object on a hard-disk or transferring it into a network’s computer?

In the programs on pages 298 and 299, objects of type `String` have been stored persistently and could be read anytime later. Is this also possible with arbitrary objects? Yes, by “serializing” objects, using the object-streams that are introduced below. Questions in that context are: (Class) Does a serialized object include its full class-information, which would be needed when reconstructing the object? (Graph) Are referenced objects also serialized, and if not, how to solve the problem of references to unserializable objects? (This may recall faintly the considerations in the context of deep or shallow copying, is there a deep and shallow serialization imaginable?)

Serialization



Only non-static, non-transient fields are serialized.
 Serialization does not bother about how to transfer the class-information!

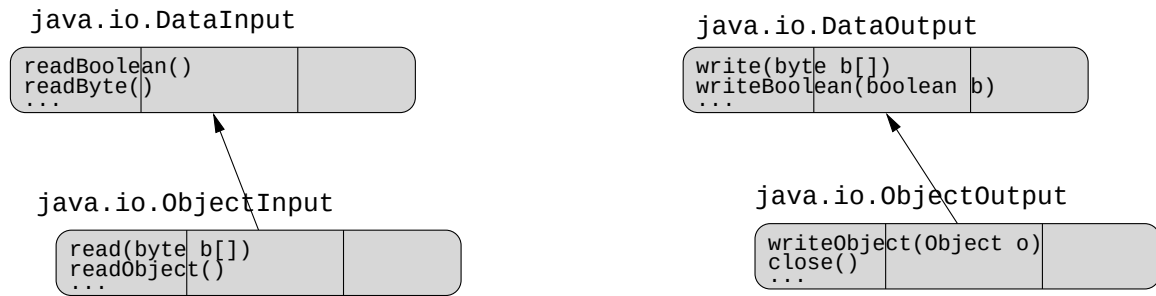
Java's object-output-streams transform entire Java objects, INCLUDING THE REFERENCED OBJECTS (Graph), into a sequence of bytes, but WITHOUT ADDING THEIR FULL CLASS-INFORMATION (Class). Furthermore, only specially marked Java objects qualify for serialization to a stream of bytes; the flag marking an object as serializable is given by the implemented interface `java.io.Serializable`. Thus, if the object is of (interface-)type `Serializable`, then THE VALUES OF THE NON-TRANSIENT AND NON-STATIC FIELDS of the class and all of its super-types are coded into a sequence of bytes for transfer over the stream.

serialization concerns only objects not classes! (only class-loading includes class-information)
 non-transient and non-static fields

(Graph) Objects, that reference each other, make a graph built by references. A graphs of objects, all objects being of type `Serializable`, may be turned into a sequence of bytes, just by reading the root-object of that graph into an `ObjectOutputStream`. Java-serialization includes all referenced objects found inside the initial object and thus serializes the entire graph-structure of related objects. Thus objects referenced by the serialized object are serialized until there exists no uncared reference, so that a complete graph of objects is serialized to a stream of bytes. This graph of *objects* can be reconstructed in another Java Runtime Environment (JRE), ONLY if that runtime-environment furnishes the entire *class*-information. The source-code for deserialization has to make sure **the class-files for the serialized objects are accessible** in the Java Runtime Environment and **the deserialization needs to be done with the programmer's knowledge what kind of object is to be deserialized**. Serialization with its relation to class-loading is revisited later on page 443.

Serialization: 1. The act of transforming any piece or structure of data into a sequence of bytes from which the original data or structure can be recovered. 2. In Java: The object and all its referenced objects have to implement the `java.io.Serializable`-interface. Then a reference to that object can be put into a `java.io.ObjectOutputStream` which takes the (non-transient and non-static) values or references out of that object and transforms them into a sequence of bytes. The reconstruction of this byte-stream requires the associated classes to be loaded into the Java Runtime Environment! This means, the serialization-process in Java only turns the data inside the Java objects into a sequence of bytes, the entire class-information to reconstruct the object-graph has to be furnished from elsewhere!

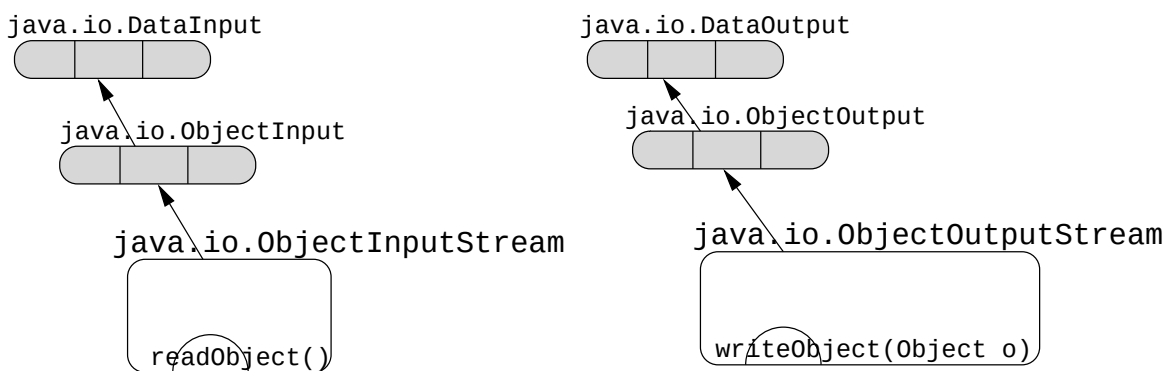
The related stream-classes implement extended interfaces: The Java interfaces `java.io.DataInput` and `java.io.DataOutput` supply a list of method-signatures. These interfaces furnish a suggestion for implementing methods that may be of use when importing data of primitive-type or of `String`-type into the current Java Runtime Environment or when exporting such kind of data. Some of the signatures are introduced in the table below, for a list of all signatures consult for example the Java Platform API Specification [JPAPIS]. (Both interfaces also have been implemented by the class `java.io.RandomAccessFile`, page 299) All objects, referenced by an identifier of the interface-type `java.io.DataInput`, do signal that they have been programmed to read simple Java data-types from an underlying input-stream.



Each of the previously mentioned interfaces has been extended¹³⁰, to respective sub-interfaces with the names `java.io.ObjectInput` and `java.io.ObjectOutput`. The added signatures indicate which method-calls should control the serialization and deserialization of an object or an entire object-graph.

data-unit	INTERFACE INput-stream-class (read)	INTERFACE OUTput-stream-class (write)
is of primitive-type or of class-type	ObjectInput	ObjectOutput
	<i>SimpleType</i> readSimpleType()	void writeSimpleType(<i>Type identifier</i>)
	String readLine()	void writeChars(String s)
	Object readObject()	void writeObject(Object o)
	int available() returns the number of bytes available	flush() calls the flush()-method of underlying stream
	close() closes, deactivates the stream	close() closes, deactivates the stream

The supported simple types, substitute the placeholder *SimpleType* above: Boolean, Byte, Char (writes int), Double, Float, Int (read or write integer-values), Line (ONLY read Strings), Long, Short, UTF (read or write strings in Unicode Transformation Format (UTF)). Observe again, that the two stream-types, which have been introduced above, are given by Java-interfaces. Up to now nothing has been introduced except conventions! Objects of this type have to be implemented by other stream-classes! (referenced on page 187)



Actually implemented methods for serialization (`writeObject()`) and deserialization (`readObject()`) are found in the following classes of the `java.io`-package:

data-unit	INput-stream-class (read)	OUTput-stream-class (write)
an object	ObjectInputStream	ObjectOutputStream
	ObjectInputStream(InputStream s)	ObjectOutputStream(OutputStream s)
	Object readObject()	void writeObject(Object o)
		size() returns how many bytes have been written to that stream

Both stream-classes above implement the previously introduced interfaces `java.io.ObjectInput` and `java.io.ObjectOutput`, respectively! Here the reading and writing includes two additional primitive-

¹³⁰An example of extending interface-types, as can be also inferred from the syntax-diagram on page 187.

types: `UnsignedByte` (read or write unsigned 8-bit values, here 8 bits code numbers 0 to 255, instead of -127 to 128), and `UnsignedShort` (read or write unsigned 16 bit values, same as above only with 16 bit), a `char`-value.

So, one interface and two classes have to be consulted when serialization becomes necessary: The interface `java.io.Serializable` and the classes `java.io.ObjectOutputStream` and `java.io.ObjectInputStream`.

An elementary example of how to use the classes `ObjectInputStream` and `ObjectOutputStream` is given below:

```
class DemoObject implements java.io.Serializable
    // otherwise expect a java.io.NotSerializableException
{ static byte num; // static members remain UNserialized
  transient String t; // fields marked transient remain UNserialized
  String s;
  DemoObject o;
  DemoObject( String t, String s, DemoObject o ){ this.t=t; this.s=s; this.o=o; }
  DemoObject(){ // needed below (super(); implicitly added)
  void print()
  { System.out.println("static num="+ num + "\nt= " + t + "\ns= " + s + "\no= " + o );
  }
}

class Serialize
{ public static void main( String unused[] )
{ // graph building :
  DemoObject leaf = new DemoObject( "nonserialized leafdata", "leaf", null);
  DemoObject root = new DemoObject( "nonserialized rootdata", "root", leaf);
  DemoObject.num = 1;
  // serializing:
  try
  { java.io.ObjectOutputStream objectOutToFile =
    new java.io.ObjectOutputStream          // >>>>----->>>>
    ( new java.io.FileOutputStream( "persistentObjectGraph" )
    );
    objectOutToFile.writeObject( root );
  }catch(java.io.IOException e){ e.printStackTrace(); }
  // reading:
  DemoObject deserialized = new DemoObject();
  try
  { java.io.ObjectInputStream objectFromFile =
    new java.io.ObjectInputStream          // <<<<-----<<<<
    ( new java.io.FileInputStream( "persistentObjectGraph" )
    );
    deserialized = (DemoObject)objectFromFile.readObject();
    deserialized.print(); deserialized.o.print();
  }catch(java.io.IOException e){ System.out.println(e); }
    // or print more information with
    catch(java.lang.ClassNotFoundException e){ e.printStackTrace(); }
  } }

/*
C:\WINDOWS\JavaCodeFiles\Basics2\StreamsPipes>javac Serialize.java

C:\WINDOWS\JavaCodeFiles\Basics2\StreamsPipes>java Serialize
static num=1
t= null
s= root
o= DemoObject@1f9e93
static num=1
t= null
s= leaf
o= null

C:\WINDOWS\JavaCodeFiles\Basics2\StreamsPipes>
*/
```

(In the print-out of the program above, the static value reoccurs because the class hasn't been reloaded.) Here class-loading has been made unnecessary, because deserialization and serialization take place in the same session of the Java Runtime Environment. Remember also that serializing (String-)objects and making them persistent on a hard-disk already has been demonstrated by the source-code on page 298!

OUTLOOK: Serialization becomes a dominating issue when sending an object's state, conserved as data, over networks; especially in the context of distributed computing or when storing data in a database (page 321). The actual network-connection may be organized by simple sockets (page 273) or by basic

infrastructures like RMI, an acronym for Remote Method Invocation (page 414). For some more remarks on serialization¹³¹ and class-loading see page 443.

Summary: Streams

Summary: Streams — The basic stream-classes `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader` and `java.io.Writer` have been declared abstract and therefore merely act as templates. Extensions have to implement the `int read()`- or `write(int i)`-methods. — A rule for denotations of stream-classes may read as follows:

- Input or Reader indicate that data is read in from somewhere else, by calling a `read()`-method.
- Output or Writer means that data is written to somewhere else using a `write()`-method.

Extensions of basic stream-classes often are wrapped with buffer- and filter-streams. A buffered stream, where the end of a data-block is marked by a linefeed, is given by the `java.io.PrintWriter` stream. `PrintWriter` streams are wrapped around Output- or Writer streams. The various `print(...)`, `println(...)` and `write(...)`-methods feed such stream-objects with data. `PipedStreams` and `BufferedStreams` transfer chunks of bytes with `flush()` and `readLine()`-methods. `Object`-, `File`- and `DataStreams` furnish `read()` and `write()`-methods that interpret a larger chunk of bytes right away as a structured piece of data of a definite type. Usually, these streams need no Buffer or Reader/Writer filters. The `StringReader`-class instantiates a stream, from which are “`read()`” individual characters. That special Reader-stream wraps only objects of `String`-class. The `StringWriter`-class instantiates the kind of streams that wrap an output-stream of `Writer`-class and write individual characters or entire strings to some external resource (hard-disk, printer, network).

What class-type of stream can wrap which type of stream?

- Consider the constructor of a stream-class, there can be seen that if an underlying stream goes in and its (base-)class can be identified, then the current stream can wrap that other type of stream.
- Consider the possibly differently defined `read()` or `write()`-methods, they give the types or classes of the data that is written to the stream or that can be read from the stream. For example, a Java stream that reads out objects may use a `readObject()`-method.

Streams can contain a highly variable amount of data in form of byte- or two-byte-sequences. The highly variable amount of data and the fact that results may be obtained from input-streams, even if the entire amount of data has not been sent yet, makes the stream-concept differ from those aggregational concepts of an array or a vector. Streams implement the concept of a fifo-container (first-in-first-out) in many variants. The wealth of Java-stream types additionally is divided into sets of input- and output-streams. This discerns streams for input/output purposes with respect to the Java Runtime Environment or for wrapping each other in view of filtering- or transformation-purposes.

¹³¹Consider also the introduction on serialization given within the Java TechTip from February 29th of the year 2000.

Code-Structure: Software-Components

Code-Structure: Software-Components — Up to now several ways to structure procedural code have been mentioned: Proceduralization, object-orientation (includes type-definition and -extension), multithreading (concurrent programming), source-event-drain communication as a generalization of a simple method-call (event-object routing and streams).

A raw and short characterization of a software-component may attempted as follows: Software-components are serializable objects which tend to communicate mainly by exchanging messages, instead of interacting through method-calls. If software-components interact by method-calls, then they use highly standardized (accessor-)methods. Communication between objects by sending messages means to retreat from a type-oriented communication of calling methods to a more protocol-oriented communication of sending and receiving messages. In Java, software-components are called Beans or Java-Beans¹³².

Concept: Software-Components

Software-components are pieces of software with the following characteristics: Software-components are **entities of (source- or byte-) code**, in the sense that isolating them does not render them unusable for further programming-tasks. Like a mosaic can be constructed from little pieces, applications can be constructed by using software-components. — Software-components may have a state, that means they may contain information. — Software-components have an application-specific programming interface. The **application-specific programming interface** is designed according to some standards and conventions, often called design-patterns. For example, such a **design-pattern**¹³³ may specify the name of methods that retrieve or set the values of a state-variable. — States together with their accessor-methods are called **properties** of the software-component. — Software-components can be made **persistent**, properties that means, together with their state, they can be stored on a hard-disk. Software-components thus are given as reusable software elements. — Software-components furnish information about their Application Programming Interface (API) (**introspection**). These requirements make software-components simple objects with a lot of additional characteristics.

Due to the design of their Application Programming Interface, software-components can be registered and analyzed by graphical **software-development-tools** which are also called builder-tools; so that the software-components can be composed graphically (by a **visual representation**) into a regular application-program (**visual programming**). This text does not show how a visual programming-tool, like the Java-Bean-Box, or a similar visual development-environment can be used. This text solely introduces the Java-specific design-patterns for programming and recognizing software-components in Java.

Concept: Object-Oriented Approach to Software-Components

Concept: Object-Oriented Approach to Software-Components — Remember that object-orientation, essentially concerned with type-definition (aggregation of variables and methods) and type-extension, allows to structure code according to a model of notions taken from a real-life-problem. So that ideally, a programmer easily can establish a problem-specific programming interface, by choosing proper member-variables and the appropriate methods to handle them. Nevertheless for standard tasks, like retrieving and storing values, the Java programmer may follow conventions, that may give the source-code an additional uniform structure. In some cases that approach may prove as giving an advantage.

¹³² In Windows-based programming-environments the programming-concept of software-components is known under the name “ActiveX”.

¹³³ In object-oriented software production, design-patterns give additional conventions and standards for the design of objects and their interaction. The area of design-pattern broadens vastly, and has its own specific list of literature. In this text, design-patterns are only touched superficially. The idea of a singleton (page 160) represents a design-pattern. As design-patterns can be seen further: The Java interface-type (around page 187), the factory-methods (page 151), the observable-observer infrastructure (page 284) and the delegation event model (concept on page 276, with its partial Java implementation on page 286 and its full implementation in the AWT-context around page 572).

accessor
(-method)
property of a
software-
component

(INSTANCE-METHODS and ACCESSORS) Standard (instance-)methods of such an instantiated class RETRIEVE OR CHANGE THE VALUE OF A MEMBER-VARIABLE. Those so-called **accessor**-methods, together with the variable they access, can be seen as a notion of its own: a **property**. Properties (variables with their accessor-methods) and other characteristics mentioned in the introduction make an **object** a so-called **software-component**:

Let public member-variables (attributes) of objects be accessible by using methods like the **get...()**-method below:

```
getmemberName() // read accessor,
```

If the property is of type `boolean`, then the **get...()**-method may be replaced or supplemented by

```
ismemberName()
```

Change the property with a **set...()**-method:

```
setmemberName(Object x) // write accessor
```

software-
components only
use the accessor-
methods of their
counterpart

Member-variables of objects that are accessible in the way described above are also called **properties of the software-component**, these specific methods are called **accessors of the property**. Software-components are meant to see only each other's properties and access the associated data only by using the associated accessor-methods.

So-called **Read-Only-properties** only have **get...()**-accessors (If they are `boolean`, then they should also have an **is...()**-accessor).

Read/Write Properties have **set...()**- and **get...()**-accessors (If they are `boolean` they also have the **is...()**-accessor).

Examples for properties in Java are given by any non-static member-variable of a class, for which have been defined public **get**- and **set**-methods as shown in the program "SimpleBean" below at the beginning of the next section. Of course the software-component's inner workings may rely heavily on non-public members, but this according to object-orientation guideline, has to be unnoticeable from outside (another occurrence of the "black-box"- or encapsulation-concept).

(EVENT-COMMUNICATION) Next to the accessor-method-calls indicated above, software-components additionally may communicate with each other by exchanging event-objects. For this task, software-components use standard source-event-drain communication-methods like

```
addEventNameListener( Listener ... )  
removeEventNameListener( Listener ... )  
fireEventName( EventObject, ... )
```

For more consult page 314.

(PERSISTENCY) Furthermore, objects which are also software-components should be sendable through networks or made persistent and thus be retrievable during different computer sessions. That means: Objects that are software-components have to be serializable. In Java, the memberless interfaces are also used for tagging purposes: The interface `java.io.Serializable` has to be implemented as a signal of serializability. (Of course, before instantiating a serialized object, its class has to be loaded!)

Basic characteristics software-components (objects derived from classes) are thus:

(Properties) Properties as members with standard accessor-methods.

(Event-Communication) Event-Communication between software-components.

(Design-Patterns) Standard conventions for names and functions of the object's methods. The Property- and Singleton- (page 160) concepts are such design-patterns, these often can be seen as helper-structures on top of the object-oriented design. Design patterns have become a wide field.

(Persistence) Serializability of the software-components makes it possible to distribute them and to make them persistent.

Thus some other characteristics become easily inferable:

(Reusability) Due to the persistency of individual objects, due to the uniformed design of the source-code, as already suggested when applying the object-oriented programming style.

(Discovery and Registration) The introspective standard of the class-design makes it easier to administer the software-components with a software-development-environment.

(Visual Programming) Due to their standardized structure (public properties with standard accessor-methods, event-communication, serializability), software-components often can be (re-)used in graphical application-development using proper Integrated Development Environments (IDEs).

Java-Beans: Software-Components in Java

Java-Beans: Software-Components in Java — Java objects that have the characteristics of software-components are called **(Java-)Beans**.

```
public class SimpleBean implements java.io.Serializable
{
    protected int value;                                // value of the property
    public SimpleBean(){}
    public void setSomeProperty( int newValue ){ value = newValue; } // accessor method
    public int getSomeProperty(){ return value; }         // accessor method
}
```

The term "SomeProperty" can be exchanged for any denotation deemed fit for the context of the programming task. The integer-value can be exchanged for any variable of primitive-type or class-type.

Bean-properties are given by member-variables: Those member-variables that usually are declared `private` or `protected` and which are accessible by accessor-methods following design-patterns¹³⁴.

Bean, Bean-Class: A Java class, given by source-code that follows special writing and naming conventions. This makes the code easier to use; and a class's instantiation, a Bean, can be handled by a special programming-tool called Bean-Box: This Graphical User Interface allows to construct larger programs (applications) by combining Java-Beans.

Standard Property

Standard Property — A member-variable of primitive-type or of class-type represents a standard property inside a software-component, in case its accessors satisfy the following design-pattern:

```
public void setPropertyName( PropertyType reference )
{ ...
}
public PropertyType getPropertyName()
{
    return ...
}
```

For properties with the primitive-type `boolean` the get-method may be supplemented by an `is`-method:

```
public boolean isPropertyName()
{
    return ...
}
```

Indexed Property

Indexed Property — In Java, indexed properties are given by class-variables that are declared to be arrays (page 234). The design-pattern of an indexed property varies accordingly from that of a standard property in Java:

¹³⁴ Design-patterns are conventions that concern themselves with the immediate formulation of source-code. Conventions are rules, in the way that they are not required to be followed, but to follow them may turn out to be rewarding — sometimes.

```

public void setPropertyName( PropertyType[] reference )
{ ...
}
public PropertyType[] getPropertyName()
{
    return ...
}

```

With additionally introduced conventions, single entities in such a collection become accessible individually:

```

public void setPropertyName( int index, PropertyType reference )
                                // index, no brackets!
{ ...
}
public PropertyType getPropertyName( int index )
{
    return ...
}

```

Apart from the convention there exists no special functionality to be observed. Before considering the types of “bound-property” and “constrained-property”, consider the programming interface for communication between software-components, furnished by the package `java.beans`:

Event-Communication between Software-Components

Event-Communication between Software-Components — The communication by a simple *method-call* is supplemented by the *event*-communication between the given software-components. This kind of event-communication follows the rules already established with the source-event-drain concept and the delegation event-model (page 276): The class-declaration of the drain/listener-object implements an extension of the `java.util.EventListener`-interface. Then the source-object can register such listeners by calling an `add...Listener()`-method, finally the source can instantiate event-objects and the source transmits their references automatically to the registered listeners. Firing a reference of an event-object can be done explicitly by calling a `fire...Event()`-method or implicitly by some connection of the software-component to the computer’s operating-system that delivers keyboard- or mouse-events to the software-component; such relaying is treated later in the part about single-user-interfacing (page 573).

```

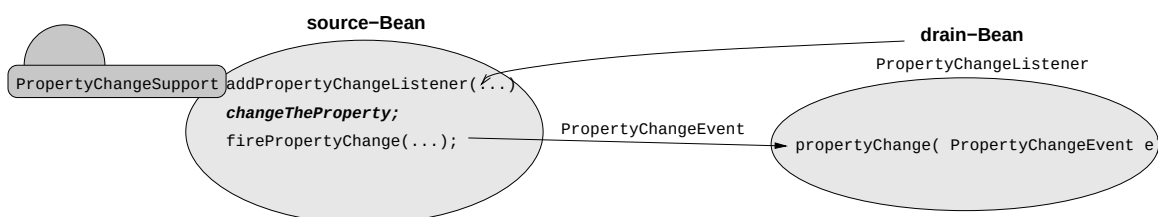
public void addEventListener( ListenerType ... )
public void removeEventListener( ListenerType ... )
notify...()
fireEventName( EventObject, ... )

```

Note that the interface-types or class-types of the arguments indicate a call-by-reference, not a call-by-value (page 119). The use of these methods is shown in the sections below:

Bound and Constrained Properties

Bound and Constrained Properties — The Java-Beans-package `java.beans` supports some functionalities exclusively pertaining to software-components. The identifier `java.beans.PropertyChangeSupport` denotes a class whose instances furnish basic switchboard-services for rerouting fired references of class-type `java.beans.PropertyChangeEvent`s to the appropriate `java.beans.PropertyChangeListener`.



In the program below is given an array of Beans: In this array each Bean notifies all other Beans of its property change. These properties of source-Beans notify their change to drain-Beans as follows:

With the source-Bean is associated an instance of `java.beans.PropertyChangeSupport`; then Beans of interface-type `java.beans.PropertyChangeListener` are registered to that instance. A `set...()`-method that explicitly calls the `firePropertyChange()` thus notifies each listener-Bean by calling the handler-method with the standardized name `propertyChange()`.

```
// informs listener Bean-objects of state-changes
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

class BeanBound implements java.io.Serializable, PropertyChangeListener
{
    int numberOfBean;
    public int getNo(){ return numberOfBean; }

    PropertyChangeSupport changeRegistry;
    // constructor and registering facility supplying a method addPropertyChangeListener():
    BeanBound( int number )
    { numberOfBean = number;
      changeRegistry = new PropertyChangeSupport( this );
    }

    int boundProperty; // ----- the value
    public void setBoundProperty( int newInt ) // ----- set...
    { // set... changes property and notifies listeners
      int old = boundProperty;
      boundProperty = newInt;
      changeRegistry.firePropertyChange(
          "the Bound Property", new Integer(old), new Integer(newInt) );
    }
    public int getBoundProperty(){ return boundProperty; } // ----- get...

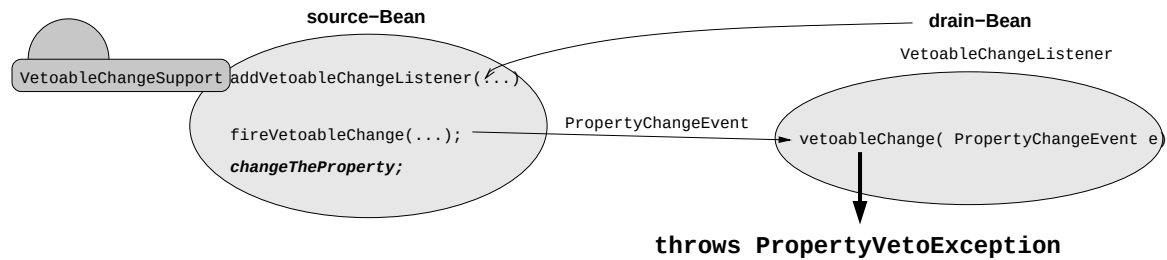
    // ----- implementing the interface PropertyChangeListener:
    // ----- means defining the handler of change notification:
    public void propertyChange( PropertyChangeEvent e )
    { System.out.println( "Bean No."+ this.getNo() + " knows: " + e.getPropertyName() +
        " of Bean no." + ((BeanBound)e.getSource()).numberOfBean + " has changed!" );
    }
    public static void main( String unused[] )
    {
        int n = 5; // choose n larger than 2
        BeanBound bean[] = new BeanBound[n];
        for( int i = 0; i < n; i++ ){ bean[i] = new BeanBound(i); }
        for( int j = 0; j < n; j++ )
            for( int k = 0; k < n; k++ )
                { bean[j].changeRegistry.addPropertyChangeListener( bean[k] );
                  // System.out.println( j + " " + k );
                }
        bean[1].setBoundProperty( 10 );
    } }

/*
[localhost:~/JavaCodeFiles/Basics2/CSSoftwareComponents] marycosway% javac BeanBound.java
[localhost:~/JavaCodeFiles/Basics2/CSSoftwareComponents] marycosway% java BeanBound
Bean No.0 knows: the Bound Property of Bean no.1 has changed!
Bean No.1 knows: the Bound Property of Bean no.1 has changed! (<- self notification!)
Bean No.2 knows: the Bound Property of Bean no.1 has changed!
Bean No.3 knows: the Bound Property of Bean no.1 has changed!
Bean No.4 knows: the Bound Property of Bean no.1 has changed!
[localhost:~/JavaCodeFiles/Basics2/CSSoftwareComponents] marycosway%
*/
```

Properties with that kind of infrastructure (an event-firing `setProperty()`-method, `PropertyChangeSupport`-instance and registered listeners of the interface-type `PropertyChangeListener`) are also called **bound properties** because other drain-Beans can bind their state or their actions to those “bound” properties of the source-Bean. bound properties

A different kind of properties are those which cannot be changed unless all the listener- or drain-Beans consent; the so-called **constrained properties**. Java classes which support constrained properties are `javax.swing.JComponent` and `java.beans.VetoableChangeSupport`. constrained properties

Constrained Property: Any property of the source-Bean that explicitly calls a `fireVetoableChange()` BEFORE it changes its value and may be stopped in changing its value by an exception thrown by one of its listeners.



This is done by introducing the class `java.beans.VetoableChangeSupport` as switchboard-service associated with the `source-Bean` and the interface `java.beans.VetoableChangeListener` as a type of the `drain-Bean`.

Constrained-properties are given when an instance of `java.beans.VetoableChangeSupport` has been associated with a `source-Bean`. Here the `drain-Beans` registered to the `source-Bean` are not only notified of an imminent property-change in the `source-Bean`, but additionally have to consent unanimously to the proposed change. If one single `Bean-object` resents the change, then it throws an exception, namely a `java.beans.PropertyVetoException`! On objection, the entire process automatically is repeated and the former value is reassigned.

The setting for the demonstration program is the same as with the previous example, except that the change is permitted only if consented to by each associated `Bean`. A `Bean` consents to the change only if the new value is smaller than a special maximal value furnished individually for each `Bean`.

```

/* Enter an arbitrary number of integers
   if one of them is smaller than 7, then the assignment is stopped and reverted.
*/
/* An instance of type java.beans.VetoableChangeSupport
   provides a method fireVetoableChange():
   1. When the programmer calls the method in the source-code,
   this triggers the dispatch of references of type VetoableChangeEvent
   to all the registered listeners
   (bean[j].vetoRegistry.addVetoableChangeListener( bean[ k ] ));
   2. Successively each single listener runs the code inside
   the handler-method vetoableChange().
   3. If one listener throws a java.beans.PropertyVetoException,
   the fire..()-method catches it and refires it,
   containing the old property-value, to all listeners.
   4. Thus all changes are reverted.
   5. Finally the fire..()-method rethrows the java.beans.PropertyVetoException.
*/
import java.beans.VetoableChangeSupport;
import java.beans.VetoableChangeListener;
import java.beans.PropertyVetoException;
import java.beans.PropertyChangeEvent;

class BeanVeto implements java.io.Serializable, VetoableChangeListener
{
    private int numberOfBean, maximum;
    public int getNo(){ return numberOfBean; }

    VetoableChangeSupport vetoRegistry;
    BeanVeto( int tag, int defaultPropertyValue, int upperLimit )
    { numberOfBean = tag; vetoableProperty = defaultPropertyValue; maximum = upperLimit;
      vetoRegistry = new VetoableChangeSupport( this );
    }

    private int vetoableProperty; // ----- the value

    // ----- set...
    public void setVetoableProperty( int candidate ) throws PropertyVetoException
    {
        int old = vetoableProperty;
        vetoRegistry.fireVetoableChange
        ( "the Vetoable Property", new Integer( old ), new Integer( candidate )
        );
        vetoableProperty = candidate; // ---- PROPERTY-CHANGE, if no VETO has been thrown!
        System.out.println
        ( "vetoable property of Bean no." + this.getNo() +
          " changes from " + old + " to " + candidate );
    }

    public int getVetoableProperty(){ return vetoableProperty; } // ----- get...

    // ----- implementing the interface VetoableChangeListener:
  
```



```

public void vetoableChange( PropertyChangeEvent e ) throws PropertyVetoException
{
    int candidate = ((Integer)e.getNewValue()).intValue();
    // work with candidate, any VETO will revert this by rethrowing a reverting event
    System.out.println
    ( "\nHANDLER of Bean no." + this.getNo() +
      " with propertyValue " + vetoableProperty +
      " received candidate " + candidate + " < " + maximum + "? else throw exception "
    );
    if( candidate > maximum)
    { System.out.println("Exception to be produced");
      // ----- V E T O if suggested change exceeds the given maximal value!
      throw
      new PropertyVetoException
      ( "Bean no." + this.getNo() +
        " vetoes because " + candidate + " > " + maximum, e );
    }
    System.out.println
    ("This Bean allows: " + e.getPropertyName() +
     " of Bean no." + ((BeanVeto)e.getSource()).numberOfBean +
     " has permission to be changed!" + " Works with: " + candidate
    );
}

public static void main( String upperLimit[] ) throws PropertyVetoException
{
    int n = upperLimit.length;
    BeanVeto bean[] = new BeanVeto[n];
    for( int i = 0; i < n; i++ )
    { bean[i] =
new BeanVeto(i, -i-300, Integer.parseInt( upperLimit[i] ) ); } /// revert-values ///
    for( int j = 0; j < n; j++ )
        for( int k = 0; k < n; k++ )
            { bean[j].vetoRegistry.addVetoableChangeListener( bean[ k ] );
              //System.out.println( j + " " + k ); // (k+j+1)%n
            }
    try{ bean[0].setVetoableProperty( 7 ); }          /////// maximum /////
    catch( PropertyVetoException e )
    { System.out.println(e.getMessage() +
      " thus reverting the value of the property of bean[0].");
    }
    // ----- some drain-Bean has thrown a V E T O
} }

/*
[localhost:~/JavaCodeFiles/Basics2/CSSoftwareComponents] marycosway% javac BeanVeto.java
[localhost:~/JavaCodeFiles/Basics2/CSSoftwareComponents] marycosway% java BeanVeto 12 5 34

HANDLER of Bean no.0 with propertyValue -300 received candidate 7 < 12? else throw exception
This Bean allows: the Vetoable Property of Bean no.0 has permission to be changed! Works with: 7

HANDLER of Bean no.1 with propertyValue -301 received candidate 7 < 5? else throw exception
Exception to be produced

HANDLER of Bean no.0 with propertyValue -300 received candidate -300 < 12? else throw exception
This Bean allows: the Vetoable Property of Bean no.0 has permission to be changed! Works with: -300

HANDLER of Bean no.1 with propertyValue -301 received candidate -300 < 5? else throw exception
This Bean allows: the Vetoable Property of Bean no.0 has permission to be changed! Works with: -300

HANDLER of Bean no.2 with propertyValue -302 received candidate -300 < 34? else throw exception
This Bean allows: the Vetoable Property of Bean no.0 has permission to be changed! Works with: -300
Bean no.1 vetoes because 7 > 5 thus reverting the value of the property of Bean[0].
[localhost:~/JavaCodeFiles/Basics2/CSSoftwareComponents] marycosway%
*/

/*
marycosway@linux:~/JavaCodeFiles/Basics2/CSSoftwareComponents > javac BeanVeto.java
marycosway@linux:~/JavaCodeFiles/Basics2/CSSoftwareComponents > java BeanVeto 45 78 56

HANDLER of Bean no.0 with propertyValue -300 received candidate 7 < 45? else throw exception
This Bean allows: the Vetoable Property of Bean no.0 has permission to be changed! Works with: 7

HANDLER of Bean no.1 with propertyValue -301 received candidate 7 < 78? else throw exception
This Bean allows: the Vetoable Property of Bean no.0 has permission to be changed! Works with: 7

HANDLER of Bean no.2 with propertyValue -302 received candidate 7 < 56? else throw exception
This Bean allows: the Vetoable Property of Bean no.0 has permission to be changed! Works with: 7
vetoable property of Bean no.0 changes from -300 to 7
marycosway@linux:~/JavaCodeFiles/Basics2/CSSoftwareComponents >
*/

```

That concludes the introduction of the Java-Beans-specific event-handling.

Persistence (of Objects)

Persistence (of Objects) — For details and prototypes see the section about serialization on page 306. For packing single files (for example: text-files of Java source-code (*name.java*), class-files of Java byte-code (*name.class*), or **serialized objects**, images, ...) into one file see the discussion of the *jar*-tool of the Java Runtime Environment (JRE) on page 325. The associated design-patterns read as follows:

```
private void writeObject(java.io.ObjectOutputStream oos) throws java.io.IOException;
private void readObject(java.io.ObjectInputStream ois) throws java.io.IOException;
```

Software-Design using Components with Reflection and Introspection

Software-Design using Components with Reflection and Introspection — Any software-development-tool can use the services provided by the `java.lang.reflect`-package to get information on classes and their objects. (For hints, see page 252, for more consult the section beginning on page 629.)

```
Class c = Class.forName( TypeOrClassName );
// or Class c = TypeOrClassName.class
Class c = ObjectName.getClass(); // basis of any reflection-process
```

Ideally the Java programmer who programmed the Bean-components adhered to the design-patterns, these are the additional conventions that supplement the Java language rules in Java-Beans programming. Then, ideally, in component-based software-development the software-development-tools can analyze such information and present it to the composer of components.

```
setProperty( PropertyType value );
value = getProperty(); // typical design-pattern
```

Just by following the design-patterns the programmer of the individual Java-Bean gives additional information to the development-tool, which then automatically identifies properties and intended event-infrastructures.

The person who composes Bean-components into fully-functional programs uses that information given by the (visual) component-based program-development-tool. For example, according to the specifications of the component-composer, the development-tool automatically furnishes additional classes that provide the event-infrastructure between the components, so-called adapter-classes.

These services are intended to simplify software-development considerably when using components.

Besides the services of the package `java.lang.reflect` and the design-patterns of the Java-Beans (reflection and use of design-patterns), there still exists another way for furnishing information about a Bean: Introspection — the programmer of the original Bean may associate with the Bean and its class, an instance of an extension of the `java.beans.SimpleBeanInfo`-class (`java.beans.SimpleBeanInfo` implements the interface `java.beans.BeanInfo`). For example, that `BeanInfo`-class may contain an icon, that represents Beans of that type in the visual-composing-environment. The details of how to associate a Bean with an explicit `BeanInfo`-class are not presented in this text.

The development-tool gets its information in about the following way:

```
java.beans.BeanInfo bi = java.beans.Introspector.getBeanInfo( c );
bi.getEventSetDescriptor();
```

Summary: Software-Components

Summary: Software-Components — Why are software-components useful? Their source-code or byte-code follows special guide-lines in design, especially the design of their programming interface, so that these pieces of software can be used as entities for application-development. This focuses especially on visual software-development within specially designed visual development-environments. What are Java-Beans? Objects given by Java source-code that follow the guidelines for programming software-components in Java. Consider the source-code of a software-component: What characteristics qualify the source-code as being that of a software-component?

(Persistence) All class-declarations implement the `java.io.Serializable`-interface. This empty interface represents a tag that allows the Java Runtime Environment (JRE) to transform objects of

this (interface-)type into a sequence of bytes by using an object-output-stream. Thereby providing the means to make objects persistent. (Source-code of classes can be made persistent by storing its text-file (*name.java*) on a persistent storage-device, like a hard-disk. On compiling that source-code with a Java compiler the resulting byte-code automatically is made persistent as a class-file (*name.class*). Java objects within a Java Runtime Environment can be made persistent, first by wrapping an object-output-stream around a file-output-stream, then by writing that object into the stream, as demonstrated beginning with page 306.)

(Property-Infrastructure) Instance-members being variables are accessible only through accessors like standard or indexed versions of `set-/get- or is-PropertyName()`-methods. Such instance-variables may represent standard-, indexed-, bound- and constrained-properties. Indexed properties use variables of array-type and accessor-methods which allow to specify an index of an individual array-element. Bound-properties can be identified easily by looking for `java.bean`-classes, where a part of the name is “PropertyChange”: There should be an instance of `java.beans.PropertyChangeSupport`; then `java.beans.PropertyChangeListener`-Beans should be registered to that instance. A `set...()`-method that explicitly calls the `firePropertyChange()` notifies each listener-Bean by calling the handler-method with the standardized name `propertyChange()`. The same applies to constrained properties when substituting “PropertyChange” by “VetoableChange”.

(Communication that follows Rules) Communication between objects is done by using accessor-method-calls or by firing event-objects, or rather their references.

(Reflection) The object furnishes means to retrieve information about its properties.

The list below attempts to collect the essential differences between classes, Java-packages and software-components (Beans):

class aggregation of data and associated methods for data-manipulation, a kind of generalized type-notion, usually used for instantiating objects.

package (=module) software library with a name-space of its own. Usually a collection of class-definitions.

software-component an object derived from a given class. The object exhibits characteristics like state given by member-variables, behavior given by methods and identity given by a hashcode. The object is a software-component, if the source-code of its class adheres to additional rules like using design-patterns (property, messaging), allowing the object to become persistent and introspective.

Java Beans are used extensively in the Java 2 Enterprise Edition (J2EE) programming-environment.

Persistence

Persistence — Usually the contents of the Random Access Memory (RAM) of the computer are lost when the computer is disconnected from its source of electrical power. Any process or thread that is terminated this way, loses values of variables and objects.

Considering **persistence** means considering how to store information on devices that store data longer than — and independently of — a computer's working session or a process's time of existence.

Outside the Java Runtime Environment (JRE) every end-user is well-acquainted with commands which make data persistent: Saving a file, downloading an Internet-file onto the local hard-disk are examples for storing data that can be retrieved in another computer-session. Within the Java Runtime Environment persistence already has been discussed when trying to save or load a file; remember reading and writing data into files on the local storage-device beginning with page 298. Or remember storing little pieces of data on the local file-system using random-access-files, using the class `java.io.RandomAccessFile`, which has been introduced on page 299. Making data persistent, may also include to explicitly serializing the data-bearing objects, but serialization can also be used for pure transmission of data without any intent of making it persistent (page 309).

What is a Database? A collection, an aggregation of individually accessible objects already gives a simple form of a non-persistent database. An array can serve as a prototype of a non-persistent database. Contrastingly an object of class-type `java.io.RandomAccessFile` represents the prototype of a (persistent) database in Java:

Database: Collection of individually accessible entities of data, usually persistently stored or persistently storable. Real-life example: Any address-book.

Usually objects or data-records in a database are accessible with “keys” like the number of an array-element or the customer identification-number in the case of a firm's database. Often those collections furnish multiple ways of access, they allow “different views on their data”. For example the database of a firm may enable a user to select all customers of ages under twenty and over fifty and so on.

A truly rough classification of databases divides them in relational-databases and object-databases. Basically relational databases store data in a format predefined within the context of the database as data-records. This predefining of formats, storing and retrieving of data is usually done with an elementary (“programming”-) language called Structured Query Language (SQL). Java's package for accessing databases which are SQL-compliant is called Java DataBase Connectivity (JDBC). Object-databases usually allow to introduce a greater variability of data-formats, therefore they often are not as fast as their relational counterparts.

Documenting and Archiving Code

Documenting and Archiving Code — Generally humans have an intuitive knowledge of the tasks which can be assigned to a computer: Automatically formatting a letter, giving templates of data-sheets, connecting to the Internet and handle the monotony of calculations necessary in accounting are a few examples. Application-programmers have to translate these everyday problems into a machine-solvable form.

As mentioned at the beginning of the book, one of the main reasons to use a processor-independent programming-language presents the portability of the source-code of programs. The languages often are also designed to allow an effective use of the computer's resources.

So, programming-languages have to reconcile two areas of thought: First, the differing human views onto (real-life-)problems that are to be solved with a computer and, secondly, the technical restrictions of the processor-commands. Thus, a programming-language has to be general enough to give freedom in programming, and specific enough to allow precise unequivocal formulations. That reconciliatory character of language-design and the higher priority of unequivocal formulations, usually makes the programs rather formal, tending not to be self-explanatory. Therefore many programs reach the complexity of a mathematical proof rather than the readability of a recipe or an easy readable list of instructions.

The question “What does the program do or what were the conceptional ideas at the time it was programmed?” can be answered by writing a separate documentation or by writing smaller remarks and observations right into the source-code. The description of the full functionality of Application Programming Interfaces (API) and application-programs fills entire reference books, as can be seen with this text in the case of the Java programming-language and -environment.

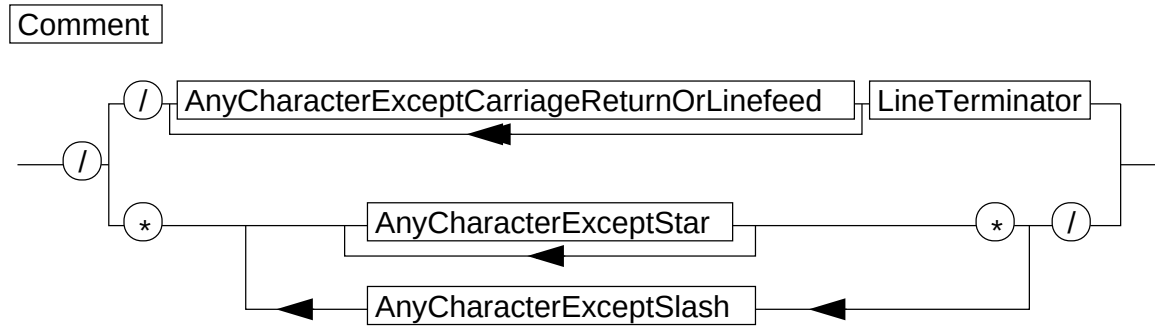
Comments and Documentation

Comments and Documentation — Texts and source-code usually are written with text-editors (some editors are part of Integrated Development Environments (IDEs)). **Literate Programming** means writing a text and making the code part of the text. Then, running the code means that the code is automatically extracted from the text and compiled or interpreted. More on literate programming can be found in the  Internet.

Conversely, source-code may be interspersed with special tokens that mark commenting text, so that interpreter or (Java) compiler ignore that part of the source-code. In Java, as could be seen numerously before, the token “//” prefixes the rest of lines of commenting text. Multiple lines or part of lines of commenting text is enclosed by token “/* ... */” pairs¹³⁵.

```
// comment on the rest of the line
/* comment that can go over
multiple lines
*/
/** A comment that can be used in automated documentation with
symbol "@" for the following keywords: @author and @version,
and for other automated documentation purposes.
*/
```

¹³⁵Nested token pairs /* ... /* ... */ ... */ produce a compile-time error. Often, when testing source-code, the application-programmer wants to comment out larger pieces of code with such a token-pair. If single-line commenting tokens “//” are commented out in testing of source-code they produce no compile-time error!



Other conventions for writing source-code more readable for fellow-humans are:

- Use meaningful names (and NO ABBREVIATIONS if the meaning cannot be simply derived)!
- Class-names are capitalized, they often are nouns.
- Method- and object-names have a leading small letter. These names often are verbs, like “thread-Lists”.
- Names of constant values (constant-names) are almost always written in capital letters (“A_CONSTANT”).

Exceptions prove the rule to be a rule only.

Java: javadoc

Java: javadoc — Another way to document source-code presents the tool `javadoc`, that comes together with the Java Development Kit (JDK). The command `javadoc fileName`, issued inside a terminal-window, extracts text from special comment token pairs `/** @ ... */` and combines that text with class-, interface- and class-member information. Then the tool `javadoc` compiles that information into a file of linked HTML¹³⁶-documents, that can be used as (online-)documentation.

ORGANIZE (Windows, Unix, Linux, Mac OS X): In command-line-oriented environments enter the command “`javadoc -private fileName.java`” to get an automated extensive documentation in HTML-format; provided that the appropriate comments have been inserted into the source-code.

```
/**
--- DOCUMENTATION COMMENTS begin with "/*".
--- Documentation comments FACE THE DEFINITIONS of:
    classes, interfaces, members(methods and variables)
--- SPECIAL TAGS prefixed by the at-symbol "@"
    contain additional information or even links
    in the remaining part of the line:
    @author Andrea di Pietro (This tag followed by the author's name
                             is used only in class or interface documentation comments.)
    @version 1.0 (This tag followed by a version-code
                 is used only for classes or interfaces.)
*/
public class DocumentedHelloWorld
{ /** A four-byte class-variable of type integer. */
  static int i;
  /** A method that maybe used by existing code, but shouldn't
      be used by newly written code.
      The "@deprecated" tag marks only such classes and members.
      @deprecated */
  void oldVersion(){}
  /** Only for methods are meant the three tags "@param", "@return" and "@exception".
      @param    parameters of the method, here none.
      @return   returns nothing, because declared "void".
      @exception Exception thrown by the method plus a closer description that
```

¹³⁶The HyperText Markup Language (HTML), furnishes a set of commands, that can be interspersed into a standard text-file, so that browser-software can format the text-output. Concrete language-elements, the so-called HTML-tags, are cursorily introduced in this text's part about networking (before page 371). HTML-files can be written with any text-editor. The HTML-format makes it possible to link text-passages to other text-passages or documents; the links can be used when reading the HTML-formatted texts in HTML- or Internet-browsers. These pieces of software display HTML-files, properly formatted on-screen, let the user follow links by clicking on marked parts of the text and usually also enable the user to print those formatted files with printers on paper. Often Internet-browsers even enable the user to edit the formatted HTML-files without having to labor through individual HTML-commands (a rudimentary WhatYouSeeIsWhatYouGet routine).


```

        can comprise multiple lines.
        The following should generate a hyperlink
        if the class-name is known and the API documentation
        is in the CLASSPATH.
        @see java.lang.String
        @see java.lang.System */
    public static void main( String unused [] ) throws Exception
    { System.out.println("Documented Hello World!");
    } }

/** Here the test for a simple class-structure */
class Test extends DocumentedHelloWorld
{ /** Another 32-bit integer variable, non-static */
    int j;
}

/* This generates files that can be found in the current directory
   (=directory where the source-code is to be found).
   For more see the Java Online Documentation from Sun and its tool section.
*/
/*
[localhost:~/JavaCodeFiles/Basics2/ArchiveDocument] williammartin% javadoc -private DocumentedHelloWorld.java
Loading source file DocumentedHelloWorld.java...
Constructing Javadoc information...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating index.html...
Generating packages.html...
Generating DocumentedHelloWorld.html...
javadoc: warning - Tag @see: Class or Package not found: System
Generating Test.html...
Generating serialized-form.html...
Generating package-list...
Generating help-doc.html...
Generating stylesheet.css...
1 warning
[localhost:~/JavaCodeFiles/Basics2/ArchiveDocument] williammartin%
*/

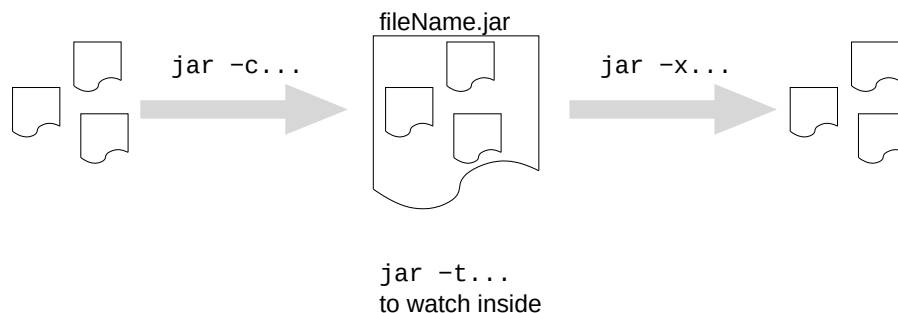
```

The newly designed Java 1.2 javadoc-tool provides more control over the output format by producing a raw output that can be changed and formatted using a doclet-Application Programming Interface (API) and the Java language. The resulting Java plug-ins for the javadoc-tool are called “doclets”. The standard doclet of course is given by the pre-written HTML-doclet. For more see the ➡️ ➡️ Java Platform API Specification [JPAPIS].

Java: jar-Tool for Archiving

Java: jar-Tool for Archiving — Essentially Java’s jar-tool takes a set of files, any type of file, usually class-files of byte-code of a Java-package, and integrates them into one single file.

jar-tool in action



Advantages of such a functionality are

single file entity This entity may contain a collection of somehow related files. This avoids tedious class-wise transfer over the Internet or over other communication-channels. Thus also signature-files, public-key-files and the signed data can be transferred as one file, see also page 489.

data-compression The automatic compression feature is highly relevant when the amounts of data are disproportionate to the channel-capacity: Transferring a large file over a comparably slow network

or loading a huge graphics-file from the hard-disk through the computer's bus-system into the computers RAM. In those cases, the overhead of compression and decompression of the data, that occurs inside the sending or receiving (part of the) computer, may still make the overall process faster!

Java JAR-files, also called JAR-archives, can be constructed by users and administrators when using the `jar`-tool that comes with the Java Development Kit. Or those JAR-archives can be constructed and read by an application-programmer who uses the `java.util.jar` or `java.util.zip` Application Programming Interfaces (API). The JAR-file uses the ZIP-compression-algorithm, for more general information see the paragraphs below.

jar-tool: Software that comes with the Java Development Kit and is used to put files and even a relative directory-tree inside a single file. The files even are automatically ZIP-compressed. These two features simplify the moving of packages or other file-collections over networks or to storage-disks. This software also allows to unpack such JAR-files and to list the files contained therein.

JAR-file: A file produced by Java's `jar`-tool. Or some file produced by Java software that uses the `java.util.jar`-package. Such files can be used as containers for multiple other files, which are compressed on packing and decompressed on unpacking. JAR-files use the Java ARchive file-format.

ZIP: 1. A compression-algorithm, specified in  <http://www.pkware.com/download.html>, implemented and made ready for use by the `jar`-tool and the `java.util.jar`-package. 2. A file-format of those files that have been compressed using the algorithm above.

From where to get the jar-tool? The `jar`-tool comes as a terminal-command which is part of the Java Development Kit (JDK). Other terminal-commands include: `javac`, which activates the Java compiler to transform source-code in text-files to byte-code in "binary"-files (=class-files). Or the `java`-command which starts the Java Runtime Environment for running the byte-code of the class-files.

How to use the jar-tool? Open a terminal-window of the operating-system; choose the desired directory; and enter the `jar`-command with the appropriate options. The options are listed and commented in the online-documentation of the Java Development Kit (JDK). Some of the common combinations of options are given below:

```
currentDirectory % jar -cvf JARFileNameOutput.jar FILE1 FILE2 ...
```

where `FILE1 FILE2` denote those files in the current directory, to be packed together into the JAR-file to be created. Or

```
currentDirectory % jar -cvf JARFileNameOutput.jar FOLDERNameInputInCurrentDirectory
```

The previous commands create a JAR-file, the following command expands (unpacks) a given JAR-file into the current directory:

```
currentDirectory % jar -xvf JARFileName.jar
```

There exists even an option that makes the `jar`-command list the names of the files contained in a given JAR-file:

```
currentDirectory % jar -tvf JARFileName.jar
```

In some cases, names of JAR-files may lack a post-qualification ".jar". This post-qualification is solely meant as a help for the end-user.

Java: Running Java Byte-Code

Java: Running Java Byte-Code — One file of source-code, in which have been defined several top-level-classes and interfaces, produces at least one class-file of byte-code per top-level-class and per interface. If not already done, inspect the folders in which are stored the files of source- and byte-code to verify that statement.

Thus almost every Java program is given by a collection of several class-files of byte-code. Larger Java programming efforts may produce programs made up by hundreds or even thousands of class-files. If (parts of) these class-files exist within a package-structure, then this collection of class-files may even display a directory-structure. Supplementary localization-information may be given by adding text-files; images may be added as image-files.

The jar-archiving-tool can be used to compound such a directory-structure together with its class-files and the auxiliary files into a single JAR-file. This single JAR-file — easy to handle because not a single file of the program can get lost, or accidentally erased — then represents the entire Java program.

Such Java program can be easily downloaded or copied some way else from a persistent storage-device to the hard-disk of the current platform. **IDEALLY**, running such a Java program should prove to be as easy:

(Unpacking the JAR-file) Unpacking the JAR-file, taking the name and relative directory-path of the class with the `main()`-method out of some “ReadMe.txt”-file and issuing a command in the terminal-window of the operating -system:

```
relativeDirectoryPathOfClassWithMainMethod# java classWithMainMethod
```

Alternatively, for example:

```
directoryPathOfExpandedJarFile#  
java -classpath relativeDirectoryPathOfClassWithMainMethod classWithMainMethod
```

(Without unpacking the JAR-file) By furnishing the name of the class with the `main()`-method in a “manifest”-part of the JAR-file, the Java program may simply be started by entering:

```
directoryPathOfJarFile# java -jar nameOfJarFile.jar
```

For details see the ➡📖 ➡📖 Java Platform API Specification [JPAPIS] on producing a JAR-file with a “manifest”.

Summary: Java Messaging

Summary: Java Messaging — Remember the conceptionally simpler methods of communication between pieces of source-code: A simple assignment, a method-call, the mutual use of the same interface-type by different classes (see the first introductory subsection of this part of the text, it refers to the example given on page 404). With the help of some of these simpler concepts have been constructed variants of the source-message-drain-concept. Some of these conceptual variants have been designed as a fully implemented infrastructure, ready to use for the Java programmer, others are given as to be implemented frameworks.

Why use a source-message-drain-concept anyway? — This concept offers the ability to consider the communication-process outside of a concrete programming context. This makes that process open to standardization. The individual objects can focus on the real-life problem, instead of additionally managing internal communication. The learning- and coding overhead is thought to become neglectable in really voluminous programming-tasks. Especially in such large programming-tasks the source-message-drain-concept introduces another way to structure and divide the source-code (encapsulation).

This compilation contains also Java event-mechanisms which are introduced in subsequent chapters. Nevertheless the overall information is gathered here to give a central point of orientation and to furnish an easy way of comparing the different mechanisms:

Thread-Communication

Thread-Communication — An example of a simpler realization of the event-model in Java gave the two threads in the threading example that sent and received references of `Integer`-objects moderated by a buffer-object. The sender-thread represents the source. The `Integer`-object was the message. The buffer-object represented a simple switchboard-object which just buffered the messages. And the receiving-thread represented the drain. (page 228)

Some special event-communication can be seen with threads using the following instance-methods to interact: The `wait()`, `notify()`-methods of the `Object`-class that can be called from `synchronized`-blocks within a thread, exchanging access-rights of a given object between the calling threads. The `join()`-method of the `java.lang.Thread`-class stops the calling thread until the qualifying thread finishes. One aspect of this method-calling between threads can be seen as a highly standardized communication between the involved threads. A sort of event-exchange without an event-object.

Source: The `run()`-method associated with a `Thread`-object.

Event-Object: None intended in the original sense of messaging.

Switchboard: Thread functionality of the Java Runtime Environment (JRE).

Drain: The `run()`-method associated with another `Thread`-object.

Trigger: **PUSH** with `notify()` because the waiting thread initiates the transfer of access-rights to the notifying thread,
PULL by `join()` because the caller thread waits for the qualifying thread to finish and to be informed about this finishing (both cases have been described beginning with page 220).

Event-communication between threads is investigated further in the chapter about the single-user-interfaces when discussing questions of proper timing within a Graphical User Interface on page 586.

Exception Throwing

Exception Throwing — Another example of the event-model was given by objects of the class-type `java.lang.Throwable` (that includes sub-classes). There the Java Runtime Environment (JRE) throws errors

or exceptions from some piece of code within a (source-) object. The programmer can do the same by using a `throw new Throwable();`-statement. Most message-objects of class `Throwable` have to be thrown inside a `try{}-block` of the source-code, otherwise the throwing thread may terminate. Ordinarily, the switching-mechanism `try-catch-finally` channels the `Throwable`-object, the message, to the appropriate catch-clause. The object that contains the code of the catch-clause, triggered by throwing the `Throwable`-object, becomes the drain. Here the Java Runtime Environment (JRE) also furnishes most of the basic communication-facilities.

Source: Any block of code (within some method) containing a statement of the kind

```
throw new java.lang.Exception("some message");
```

Or any block of code in which the Java Runtime Environment (JRE) encounters a situation that it can handle only by throwing a checked exception (for example a Java Input-Output exception: `java.io.IOException`).

Event-Object: An instance of class-type `java.lang.Exception`, or more general of the class-type that produces instances for “throwing”: `java.lang.Throwable`.

Switchboard: Already fully implemented in the Java Runtime Environment; this process is given by a simple (“exceptional”) return-process along the previous method-calls, supported by declaring the affected methods as “throwing exceptions”:

```
modifiers methodName( parameterDeclaration ) throws java.lang.Exception  
until the exceptional return happens within a try-catch-finally-clause:
```

Drain: Any block of code that contains a `try-catch-clause`. The code that is invoked on receiving the exception is found inside the `catch-clause` reserved for the received kind of exception (as a concrete example is chosen an Input-Output exception):

```
try{ blockOrAndMethodCallThrowingExceptions }  
catch( java.io.IOException e ){ IOExceptionHandling }  
catch( java.lang.Exception e){ handlingAnyOtherException }  
finally{ cleanup }
```

Trigger: PUSH: The source-code sends the exception by instantiating it inside a `throws`-statement, or the Java Runtime Environment throws such an exception on its own.

An extensive treatment can be found beginning with page 279.

Exception throwing and the observable/observer-mechanism have fully-implemented switchboard-functionalities. Java relies heavily on the so-called delegation event-model (page 276) also named callback event-model or event-transfer through registered listeners:

Observable/Observer-Mechanism

Source: An object instantiated from a class that extends the class `java.util.Observable`. An observer can be registered and de-registered by calling the methods `addObserver()` and `deleteObservers()`. The event is sent by calling the instance-method `notifyObservers()`.

Event-Object: An arbitrary object or a mere notification of a change.

Switchboard: Already fully-implemented in the Java Runtime Environment (JRE).

Drain: An object instantiated from a class that implements the interface `java.util.Observer`. This requires the class to implement a method `update(java.util.Observable o, java.lang.Object event)`, that contains the code to be run on receiving the reference of the event-object. This means the code of the `update()`-method serves as the event-handler.

Trigger: PUSH: A call of `java.util.Observable.notifyObservers()` after having called the method `setChanged()`.

This has been presented more extensively beginning with page 284.

Framework: `java.util.EventObject`, `java.util.EventListener`

Source: A Java programmer has to produce the sources usually by implementing methods like `addListener()` and `removeListener()`. Sources may also have to implement methods with names like `notify...()`, `dispatchEvent()` or `fireEventType()`.

Event-Object: Any instantiation of the class `java.util.EventObject` or any sub-class: An object of class-type `java.util.EventObject`.

Switchboard: TO BE FULLY IMPLEMENTED BY THE Java programmer! This means implementing `addListener()`-methods and `fireEventType()`-methods in such a way that they can be called by the event-source, thereby establishing the connection to the handler-methods of the drain-object that implements the interface `java.util.EventListener`. The ENTIRE design of the switchboard is left to the Java programmer!

Drain: Any object that implements the tagging interface `java.util.EventListener` (or extensions thereof). The extensions specify the names of the handler-methods that contain the code to be run when the listener receives an event-object.

Trigger_PUSH: The `notify...()`-or `fireEventType()`-methods usually are implemented by the event-source.

See page 288 for the organizational overhead, see page 633 for a demonstration of a fully-implemented switchboard, see page 571 or see below for an implementation within the context of the Abstract Window Toolkit (AWT).

Java-Beans — Software-Components

Source: Any Java-Bean; that means any object that is derived from a class whose source-code has been written according to the conventions of the Java-Beans component-software specification. Standard ways to send standard events means to call methods like `firePropertyChange()` furnished by objects of class-type `java.awt.Component` or to invoke the method `fireVetoableChange()` furnished by objects of class-type `javax.swing.JComponent` or `java.beans.VetoableChangeSupport`.

Event-Object: Any instantiation of the class `java.util.Event` or an extension, especially the event-type `java.beans.PropertyChangeEvent`.

Switchboard: An object, given by an instance of the class-type `java.beans.PropertyChangeSupport` or of the class-type `java.beans.VetoableChangeSupport`; any further switchboarding is supposed to be done automatically by visual component programming-environments according to the component-arranger's specifications.

Drain: Any Java-Bean; that means any object that is derived from a class which has been defined according to the conventions of the Java-Beans component-software specification. An object that implements the interface `java.beans.PropertyChangeListener` or the vetoable property-change-related interface `java.beans.VetoableChangeListener`, or any other event-listener.

Trigger: PUSH: Effectively, the source sends.

This has been introduced beginning with page 314.

Abstract Windowing Toolkit (AWT) and Swing

Source: Any object that is a screen-component; any object instantiated from `java.awt.Component` or extensions thereof. Event-objects are generated and sent either automatically by the Java Runtime Environment (keyboard-signals, mouse-movements, ...) or within Java Beans-context (including Swing), events can be sent by by the Java programmer by calling `fireEventType()`-methods.

Event-Object: Any instantiation of the class `java.awt.AWTEvent`¹³⁷ or its numerous extensions in the packages `java.awt.event` and `javax.swing.event`.

¹³⁷An extension of the class `java.util.EventObject`.

Switchboard: Already fully implemented in the Java Runtime Environment (JRE); using default classes like `java.awt.EventQueue`, accessible to a programmer by calling methods, qualified by references of class-type `java.awt.Toolkit`.

Drain: Any object that implements the tagging-interface `java.util.EventListener` or its numerous extensions in the packages `java.awt.event` and `javax.swing.event`. Any methods to be implemented have to contain the event-handler code!

Trigger: PUSH: Usually and effectively, the source sends.

To be introduced beginning with page 575.

Summary: Java Language

Summary: Java Language — The Java language is a high-level (processor-unspecific) programming-language. Java's source-code is given as simple text-files, to be compiled (an end-user enters the command `javac` into a terminal-window of the native operating-system) by the Java compiler into so-called byte-code, stored in class-files. This platform-independent byte-code only can be run on a virtual platform called Java Runtime Environment (an end-user enters the command `java` into a terminal-window of the native operating-system). Any Java Runtime Environment (JRE), started by the `java`-command, is run from the files that make up the Java Runtime System (JRS) for the underlying, native platform.

Apart from the Java programming-language and its environment, in the last two parts of the text have been introduced eight concepts for structuring source-code: Proceduralization, object-orientation, multithreading, aggregation, packaging, event-objects (messaging), streams and software-components.

What makes the programming-language Java special?

- Java has no conventional pointer-concept, in contrast to C++
- Java allows object-oriented code-structuring, so do ObjectPascal, Smalltalk, C++
- Java supports multithreading, IF the underlying operating-system supports multithreading AND IF the Java Runtime System uses the underlying multithreading support! Or IF a multithreading-infrastructure is custom-programmed by the Java programmer.
- Java has an integrated Application Programming Interface (API) thus furnishing an extensive platform-independent software-library.
- Java is a virtual-machine-based language and therefore, its source- and byte-code is deemed portable, so was Apple Pascal in the year 1982.
- Java's virtual-machine-concept is supplemented by a software-library that makes the programming-environment an entire virtual platform, the Java Runtime Environment. Ideally, this Java virtual platform permits to write source-code that uses portable input-/output routines.
- Java's security-infrastructure allows to restrict access to resources (like file-system, input- and output-streams) *from outside* the Java-code, via policy-files! Before Java Version 1.2, restricting code *from outside* was only possible by using applets which had to run in an applet runtime-environment, the so-called applet-sandbox.

OUTLOOK: The remaining three parts of the text consider communication between computers, security and single-user-interfaces. (As already mentioned in the preface, there is no imaging, no document-processing or related topics found in this text.)

Part V.

Communication between Computers Networks and Distributed Computing

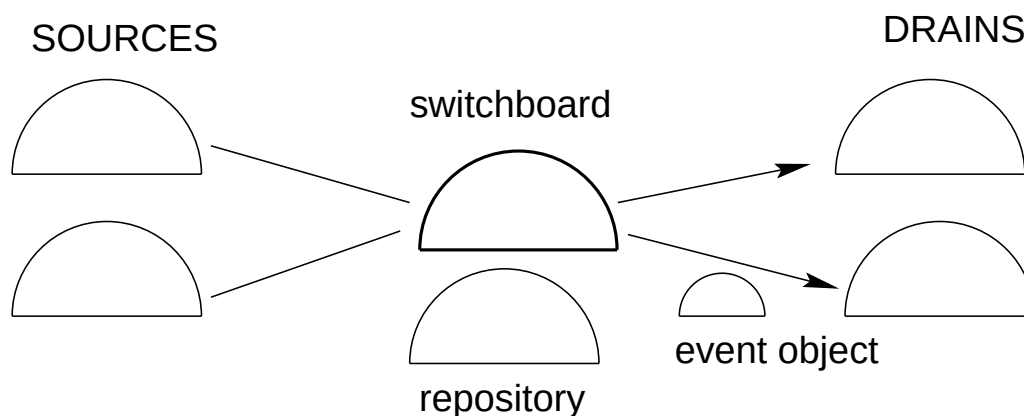
Communication between Computers, Networks and Distributed Computing — Remember that the very first chapter presents a computer-model visualized on page 28; exchange of data already occurs in that computer-model: The internal bus allows the processor to access large amounts of data in the Random Access Memory (RAM) or in the address-space (port) where keyboard- and mouse-inputs have to be handled; as well as in the address-space (port) where the outputs to the monitor are written. So basically, in a computer the hardware-bus connects, and handles the communication of different parts of the computer itself. — As the graphic on page 30 already indicates, it is possible to hardwire¹³⁸ different computers together and let them communicate with each other. The first chapter of this part of the text introduces models of computer-networks.

A single computer, without possessing a more or less constant communication-channel to another computer, is called an *isolated (computer-) system*. Contrastingly, a collection of more or less autonomous computers is called a *network*, if each computer has a means of communication with any other computer of that collection. — In information-technological-context, the notion “network” describes a number of independently working computers which can exchange data between each other.

For example, the Internet represents a very large network of computers. According to its name the Internet actually is given by an interconnection of many networks, each network may be constructed technologically differently. Parts of the Internet may also be furnished by organizations and firms with their computer-networks, as well as single computers connected only temporarily to the Internet over a telephone line.

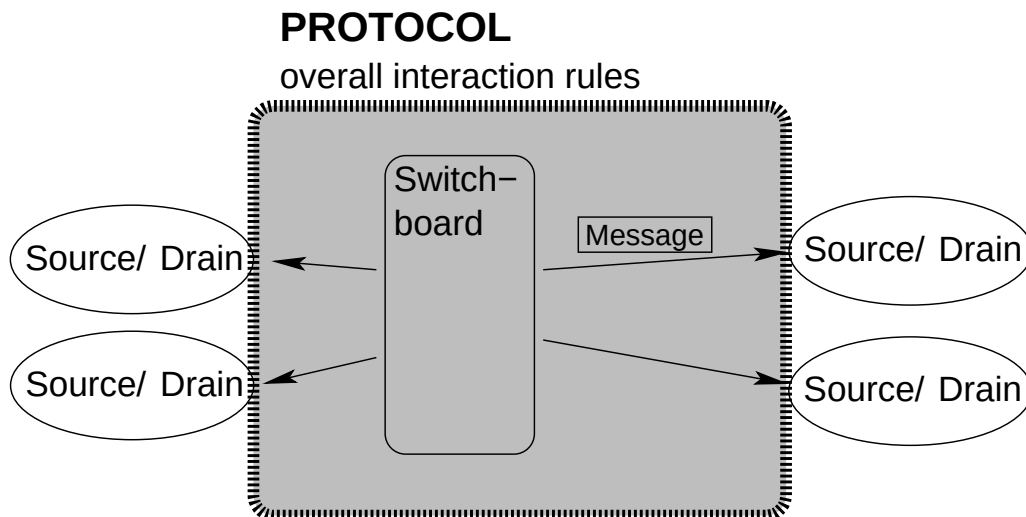
Networks of computers ease data-exchange; potentially making work simpler and faster and thereby increase productivity, giving a higher standard of living, which should help to improve the overall quality of life — ideally.

The classical (analog) telephone system can be seen as THE basic example of a large network. In the classical (analog) telephone system the endpoints are telephones! Well known non-computer-networks of everyday life are the postal- and the telephone-systems. Functional elements of the postal system are the sender of a letter or a packet, the receiver, the postal organization and your list of addressees that may have been compiled from letters you got or from other resources. Functional elements of the telephone system are the telephones of the caller and the to-be-called, the classically switched telephone system and the telephone book or the telephone operator. All those functional elements for both systems can be recognized in the graphic of the source-switchboard-drain and database-model on page 272. This idea remains the basic idea of computer-networks.



If, in this model, the sources and drains exchange their roles dynamically, and the sources and drain are individual, connected computers, then this reflects the idea of a computer-network.

¹³⁸“Hardwiring” may include cable-connections as well as radio- or satellite connections.



In computer-to-computer communication the notions of a protocol (page 274) including the addressing of drain-entities usually cover and substitute the technicalities of switchboarding. (In the same way as dialing a telephone-number gives access to the telephone-network without the need to know any underlying technicalities.)

Remember the layer-model on page 53 for the hard- and software on a single computer. This model divides a Java-enabled computer-system as given by operating-systems (hardware-related software), runtime-environments and application-code. In the case of Java, source- as well as byte-code have a platform-independent status. This software- and mental structure should ease the overall handling of such a computer-system. A very similar layer-model, concerning protocols, also intends to separate hardware-specifics, network-management and network-usage; so that a programmer or even an end-user can be oriented by a simple network-model when using the services of a real network. (Just like the telephone-system.)

Here the source-message-drain-switchboard-trigger model will be seen mostly as a host-protocol-trigger model.

Two basically different means of using networks should be clarified right at the beginning, often inter-mixtures of these two alternatives are realized:

Exchange of Data The network-communication is used to transfer data of application-programs (Internet pages, database-entries). Here the operating-system solely manages local processor-timing and memory-access. Network-access may be part of the operating-system, but usually the operating-system does not use resources outside the local system.

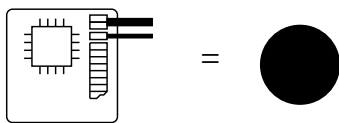
Exchange of Programs If the data exchanged, contains code that can be run (automatically) on another computer of the network, then this means EXCHANGE OF RESOURCES (MEMORY-SPACE, PROCESSOR-TIME). This is also called distributed computing. Distributed computing includes the case when the computer-user manually downloads a file that contains a program-code to run it on the computer. But distributed computing may extend to the case where the operating-system automatically delegates computing tasks into the network and the user does not need to bother where the resources come from. In this case operating-systems (OSs) also may interact with other operating-systems on managing network-load and on assessing whether it is justified to share processor-time and memory.

Aspects of distributed computing include Java-applets running on client-computers, Java-servlets running on servers, JavaScript (a language embeddable into HTML-text-files, which has little or nothing to do with Java) on application-level running on client-computers, Java RMI (Remote Method Invocation) on language-level.

Distributed Computing: A task is handled by multiple computers that coordinate their work over a network. See also the section about distributed programming (page 423).

Concept: Classification of Computer-Networks

Concept: Classification of Computer-Networks — A single autonomously working computer-system as described beginning with page 28 is called an **isolated system**. The operating-system of such a computer manages local processor-timing and memory-access. If that computer happens to be a multiprocessor machine, as introduced in the chapter on multithreading, then processes and threads may run truly parallel. An appropriate operating-system tries to organize the threads for efficiency (waiting-time and memory-use).



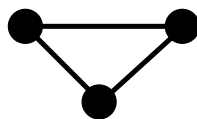
an isolated system means no connection to another system

If such a computer is given a physical¹³⁹ connection to another computer so that they can exchange data (and thus also program-code), then the two computers already form a simple network. During the connection-time these computers can address each other in that network. Computers which are able to communicate within a network are called **host** or **node** of the network. So, a **(computer-)network** is given by several computers which have been mutually connected by at least one physical connection, so that they can access parts of each others memories.

Host-Name: Name of a computer in a network. For example, this can be an → IP-address of the computer, even a → domain-name may act as a host-name, or the hardware-associated code of an Ethernet-card that has been installed in a computer. Usually an IP-address or a domain-name are just names to be registered with some switchboarding-software, see page 360. Or, Ethernet-addresses are hardwired to Ethernet-cards at their manufacturing time.

A host-name is used to address one single computer inside the network. One computer may use several host-names, but each single one of them should make it possible to identify the computer independently from the other chosen names. A host-name never identifies two computers inside a network.

(In a telephone-network, the telephone-numbers are the host-names. — The postal system uses street-names, zip-codes, post-office-box numbers and country-names for the identification of a source or a drain of a message.)



Computer-networks can be roughly described by answering four questions:

Number How many computers are part of the network and which physical connections exist between them? What maximal number of hosts can be supported by this network?

Distance What is the maximal geographic distance of two computers in the network? What maximal distance is attainable?

Quality of Service How much data per time can be transferred without too many losses and can a minimal data-flow be guaranteed?

Protocol(s) What mechanisms handle the addressing and guiding of the data when passing the network?

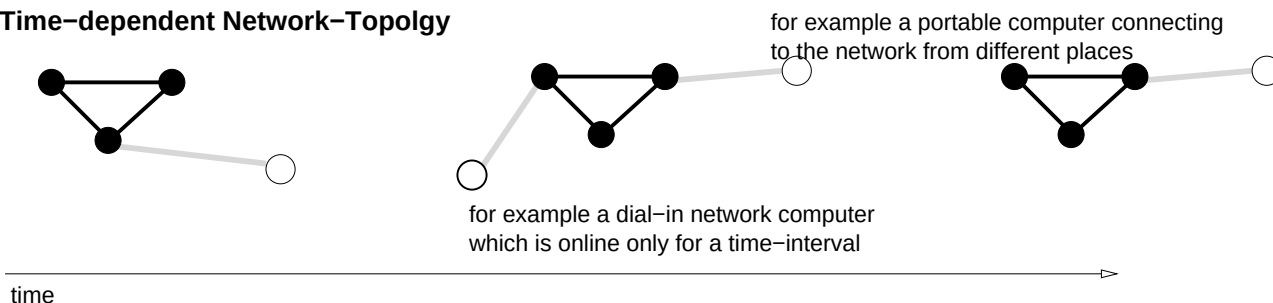
The following sections give some ideas what to expect when getting an answer to these questions:

¹³⁹That includes radio connections like those of mobile-phones, satellite-receivers or short-range radio-connections (household-range or city-range).

Maximal Number of Hosts and Network-Graph

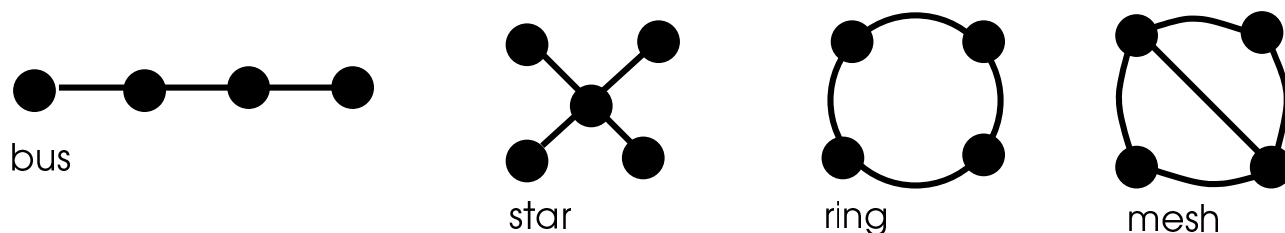
Maximal Number of Hosts and Network-Graph — One of the first informational data on a network may be seen to be the number of members or nodes of the network. In computer-networks the single computers are also called hosts or nodes if their relation to the network is to be stressed. Of course there may be nodes that are part of the network only at certain times and connecting at varying places. Such computers are members of the network only during sessions (page 277)!

Time-dependent Network-Topology



graph of a network, topology of a network

The entity of all existent physical¹⁴⁰ connections in a computer-network is called the **graph of the network** or the **topology of the network**. Two aspects should be stressed: First, in most cases, except some satellite-links, the **physical connections are two-way (duplex)-channels**. Second, **basic topologies** are those of a bus, a star and a ring. Nodes that vary their presence in time and connection, usually leave the basic network-topology unchanged.

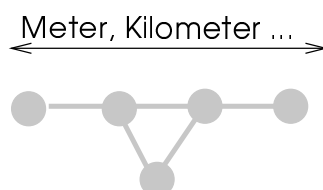


Many real networks exhibit considerably more complexity. But those basic topologies already demonstrate the advantages and disadvantages of some kinds of network-structures: If one connection in a bus topology fails, then a large part of the bus may remain unreachable. If the center of a star-shaped topology fails, then the entire star-network fails. A ring topology always gives two different ways to reach another node and works therefore more failsafe. Still, meshlike network-topologies, which allow to connect their nodes in many different ways, are the most reliable ones in case of any local failures. (Meshlike topologies usually are the more expensive ones and therefore not very popular with some people.)

Network topologies of some more locally confined networks are following: SCSI (Small Computer Systems Interface) and some Ethernet networks (10BASE2 and 10BASE5), allowing around ten hosts, have a bus topology. Other Ethernet specification (10BASET) and ARCnet use a star-topology with a hub or concentrator in the center. TokenRing nets have been given a ring topology, although the cabling looks star-like. To become fully-functional, such networks have to use software running on their hosts.

Geographic Extension

Geographic Extension — The longer the distance between two computers, which have been connected by a wire for data-transfer or by a link that uses radio waves, the more physically rooted disturbances may occur. So the geographical extension of a network determines the different technologies that may be used to cope with these problems.



¹⁴⁰Physical means also a radio connection!

This gives the following classification:

clustered computers (meter) represent a network with high capacity reliable network-links among each other, often even installed in one rack (Technology-name: USB (Universal Serial Bus), IEEE1394 / FireWire / iLink, Ethernet, SCSI)

Local Area Network (LAN) (1km) located in one building or one city (Technology-name: Ethernet, Token-Ring, ISDN (Integrated Services Digital Network), ADSL)

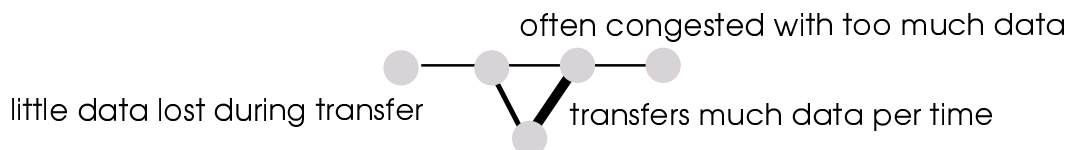
regional network (10 km) network or linked-together networks in one region or city (Technology-name: FDDI, ATM, ISDN)

Wide Area Network (WAN) (100km and more) network of networks linked together on a continental or even inter-continental range (Technology-name: T1, T3, FDDI, ATM)

The length of physical connections can be longer than specified by inserting so-called repeaters into network-connections; these hardware-devices refresh the physical signal.

Quality of Service QoS (Flow Capacity, Load and Reliability)

Quality of Service QoS (Flow Capacity, Load and Reliability) — A regular phone-line, used with conventional analog electrical signal pulses, may allow communication between computers with 14000 bps (bits per second) up to 56000 bps. The same copper-wires used as ISDN channels have a maximal capacity of 64000 bps. Large intercontinental connections have a thousand times that capacity. But the capacity means little if the connection is blocked by network-traffic or leaky by data-loss. Low reliability or low maximal capacity means low effective capacity. Ideally, a network-service should be supposed to guarantee a minimal flow-capacity.

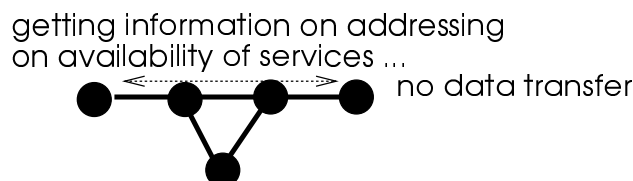


These questions are especially important for cases of real-time data-transfer which arises when transferring audio- or video-data interactively or live for the end-users.

Some technologies furnish extended Quality-of-Service-(QoS)-features, others do not: Ethernet and ARC-net networks furnish no guarantees except bit-redundancy checks. TokenRing nets can transfer data with a choosable priority. The ISDN telephone system can guarantee 64Kbps on one single line, a similar service offers ATM. Here the character “K” abbreviates a factor thousand, being used in the context of physics-related hardware description. Other uses of the symbol “K” (“K”=factor 1024), especially in bit- or byte-counts of memory-spaces or data-volume, have been mentioned on page 34.

Communication in a Network — Protocols

Communication in a Network — Protocols — Remember the chapter about communication, there the message/data-flow between sources and drains was channeled or routed by a switchboard. The rules for interaction between source and drain were given by a protocol.



Now consider multiple computers connected, and thus forming a computer-network. The principles of transferring data resemble those imaginable being relevant in the telephone system or in the postal system; therefore the very first question may be:

How are the (Host) Computers identified in a Network? (Host Identification)

Of course, on this level of generality the answer is simple: By giving each host an address, so that the computer's **address** is unique in the entire network. (Like a telephone-number in the telephone-system. Additionally, in the "real world" of the telephone system, a network self-information-system is given by the conventional telephone book or even online, the telephone operator who informs about a friend's telephone number. To be more concrete, for example: Ethernet network-addresses are hardwired into each produced Ethernet card and thus each single ever-produced Ethernet-card is given a unique address of the hexadecimal form like this one: 00104B384F08.) In the source-drain-switchboarding model the naming problem hasn't been considered as relevant. But addressing alone does not get a letter on its way, the mechanics of sending it, routing it, delivering it, or in case of problems, returning it to the sender are functions vital to every network-technology. This may provoke the next question:

Inside the Network, how to find the Path from the Source of the Data to the Drain? (Pathfinding)

Using the already mentioned source-drain-model for a non-trivial network immediately poses the switchboarding-problem. But in a larger network a centralized switchboard would be a very vulnerable object, so networking concerns itself essentially with decentralized switchboarding-mechanisms. Therefore many hosts have to use their computing capacity and some of the network's capacity for establishing a somewhat decentralized switchboard system. This system, also called **signaling system** is almost invisible to the end-users; just like the telephone system technology that opens channels from one telephone to another telephone or the postal-service mechanisms for delivering letters and packets which are unknown to many users.

So-called **signaling-protocols** for computer-networks are the Routing Information Protocol (RIP) or the Simple Network Management Protocol (SNMP). Usually no application-programmer or end-user needs any conscious contact with that kind of protocol.

Suppose that the pathfinding and/or channel-switching in the network of computers does not represent any problem:

How is the Data actually sent, in what Form does the Data pass the Network? (Path Using)

In the postal service, letters and packets have to follow certain rules according to size and weight, to priority (air-mail or sea-land transportation), they have to be addressed properly and may need some additional forms like a customs declaration. In the source-drain-model the sender and the receiver have to agree on the format of the message or even the rules of establishing the entire communication (protocol). Counterparts to these examples, in a computer-network, are the **transport-protocols**, here the data has to be supplemented with source- and drain-addresses and further organizational data before being dispatched.

Transport-protocols, which are widely used, are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

These are three basic questions (host identification, pathfinding, path using) to ask, when surveying a specific network-technology, whether it is software-emulated or hardware-based. Already those three issues on network-communication lead to significant complexities on a purely conceptual level.

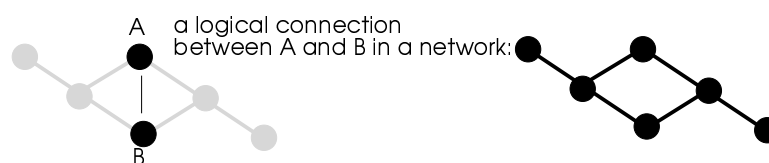
Concept: Logical Connections, Virtual Networks

Concept: Logical Connections, Virtual Networks — When telephoning, the sender and receiver speak to each other without bothering what line of cables transfers the call. — Instead of knowing that the call is routed over several automated switchboard stations and the signal is converted to electrical pulses that may also be transformed into electromagnetic waves for going over a satellite link and then may be transported over an optical fiber as light-pulses, — instead of bothering with these technicalities, a telephone-user only handles very effectively the logical connection that is established between the two telephones: 1. Dial the number, 2. Wait for the answer, 3. Talk! Thus the end-users can concentrate on their area of interest without being burdened by technical complexities. This represents another example of the idea of abstraction or encapsulation, as already mentioned alongside the introduction of proceduralization and object-orientation. And another real-life example is given when sending a packet through the mail: Put address, return-address and stamps on it, hand it over to the clerk in the post-office and wait for the things to happen.

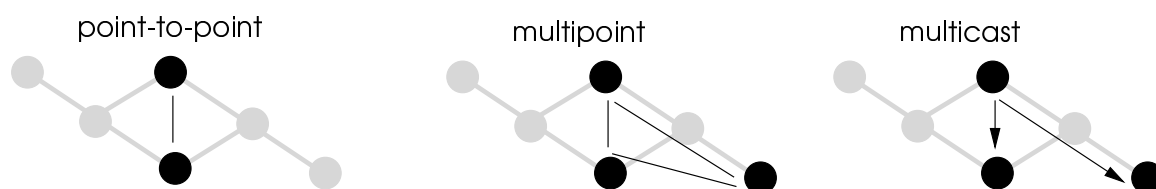
Such idea of a connection in a network that ignores application-irrelevant background matter is called a **logical connection**.

Observe that the physical connections, making up the network (of computers), give the paths for data to be transferred between nodes. Considering only the act of the data-transfer between application-relevant-end-nodes establishes a so-called **logical connection** between these end-nodes.

Example: The telephone network as a whole gives a lot of (physical) telephone-to-telephone connections. But between each pair of telephones, for end-users there exists only one logical connection!



Logical connections can be **point-to-point connections**¹⁴¹ like in the case of a regular telephone-call. But a logical connection also can be a so-called multicast connection. **Multicasting** in a network means sending the same information to a selected set of different receivers, much like a mailing-campaign that addresses itself to a set of potential car-buyers. **Broadcasting** in a network means to send a message to each and every host in the entire network, much like a television station reaches a lot of viewers. (If wanted, extend the graphic below with the case of a broadcast:)



Multipoint connections bind selected nodes of a network into a two-way communication-process, much like a conference-call on a telephone. Multipoint connections may be conceptionally close to what can be called a (virtual) sub-network: A given subset of hosts are connected to each other by a set of logical connections (from whatever kind of network they come), so that each computer has a means of communication with any other computer of that collection AND usually excluding all other computers of the network which are outside the given subset. Such a “logical” network is also called **virtual network**. Due to the fact that only logical connections are seen in a virtual network, in this case any topology-related signaling-tasks remain out of consideration!

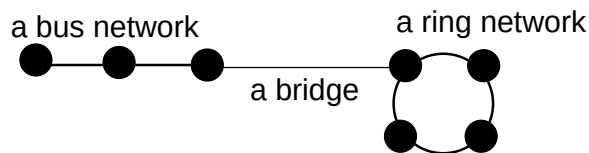
A **sub-network** of a given network either is a virtual network or a part of the network that can function sub-network

¹⁴¹Further specifying a point-to-point connection, the notion “unicast” may stress a *one-way-flow* of data, as insinuated by the suffix “cast”.

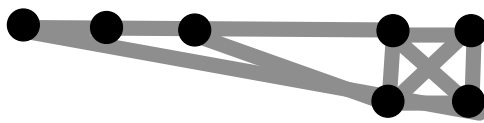
as a network on its own. An example of an isolated network inside a network is given by an isolated network on the basis of the Internet Protocol (IP), this is called **Intranet**.

The last paragraphs illustrated the idea of a sub-network — the following paragraphs consider the construction of a larger network by connecting multiple smaller networks:

Another direction of approach is to consider (even technologically) different networks: Those networks may be connected so that a software common to all hosts may establish a single large – a virtual – network. This gives the conceptual basis of the Internet: The Internet Protocol (IP) covers the technical differences of many network-technologies, so that end-users, application-programmers and many system-programmers see only one homogeneous virtual network: The **Internet**!



an inhomogeneous network of multiple network topologies becomes an IP virtual network, a single and homogeneous network!



As can be seen in the graphic above, on the level of the Internet, on the level of the virtual network, the network-topology may be ignored.

Concept: Data-Transfer with Packets and through Channels

Concept: Data-Transfer with Packets and through Channels — What is a protocol?¹⁴² What are packet-oriented protocols and what are connection-oriented protocols? Why use packet-routing instead of line- or channel-switching? How do such networking protocols differ from transport-protocols? In short, what is the idea behind data-transfer in packets (routing) and data-transfer through channels (switching)?

First reconsider two examples: The classical (analog) telephone system and the postal letter and packet service. — A telephone connection establishes a channel (an uninterrupted line of communication) between the end-users with a guaranteed capacity for the transmission of the voice signals. Thus the analog telephone system represents a connection-oriented network. — Whereas the postal service routes letters and packets until they reach the destination or are returned to the sender. This already describes the two basic ideas for transferring data in a computer-network:

Channel-switching and packet-routing.

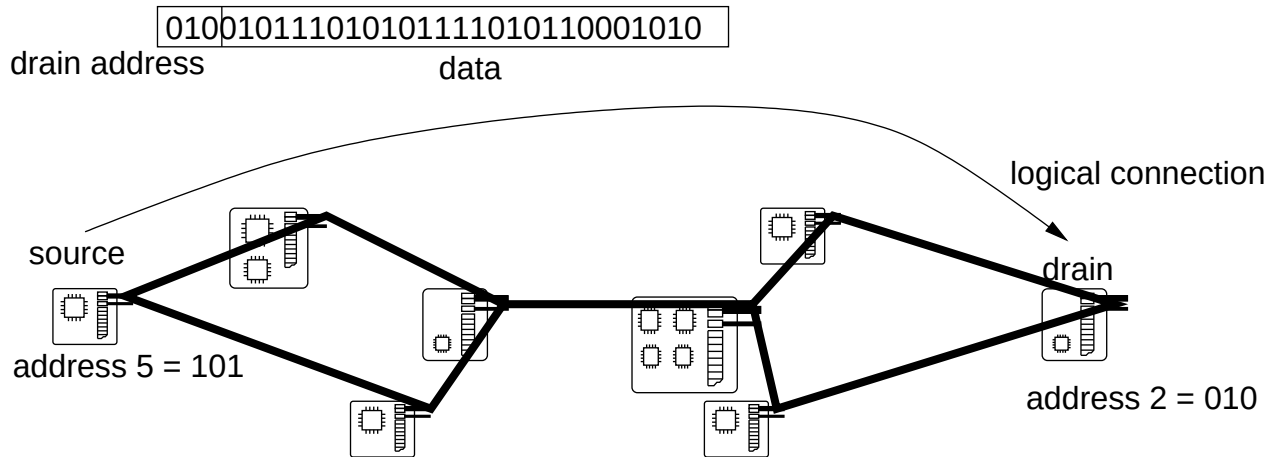
Channel-switched networks are also called **connection-oriented networks** because data is exchanged through temporarily established connections like those in the classical (analog) telephone system.

In so-called **connectionless networks** data is transferred in addressed packets. Data-transfer by packet-routing requires larger amounts of data to be broken down to little chunks to be packed, addressed and sent individually to their destination. After being received the data-chunks are unpacked, ordered and the original piece of data is reconstructed. The organizational overhead of packet routing often is justified by resulting in a less congested network. Such networks may also be called packet-routing networks.

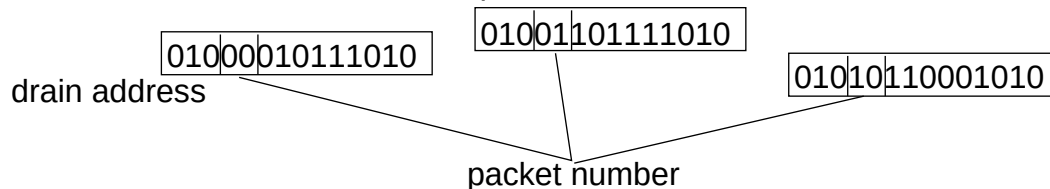
Since, in every computer-network, the capacity of connections for transferring data is limited, the inherent Quality of Service (QoS) characteristics for the channel-switching- and the packet-routing-approach have to be assessed.

¹⁴² A Java access-modifier like `private`, `public` and `protected` are not part of the type-specification but rather a “protocol”-part of the defined class-member! These modifiers tell who, from where can access the data; the type-declaration does not really change if these modifiers are omitted!

Packet Routing



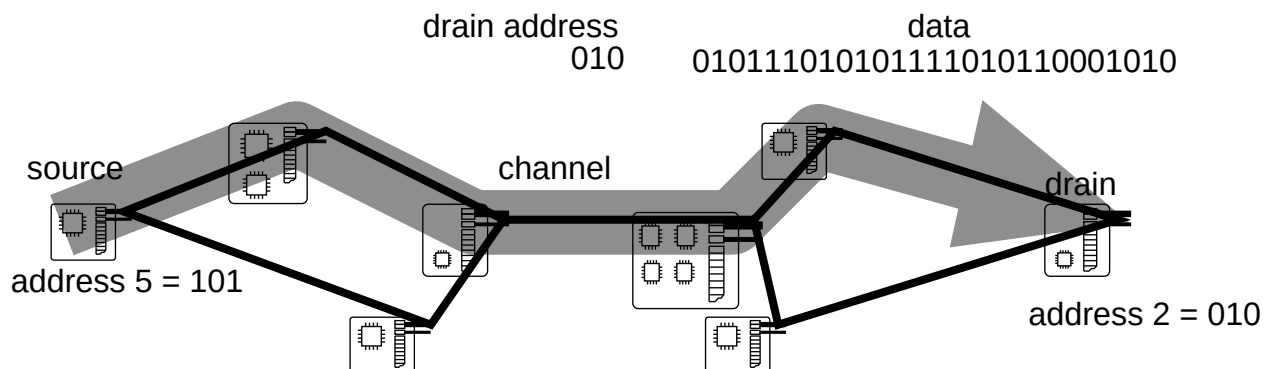
same data in form of three enumerated packets:



Routing Advantages are: Less network congestion due to greater flexibility in routing the data-flow. Blocking is very unlikely to occur; only the overall transport might get slower if the data-lines are used to their full capacity.

Routing Disadvantages: Organizational overhead of (un-)packing and individual routing. Quality of Service (QoS) guarantees require more organizational overhead.

Channel Switching



Switching Advantages: Quality of Service, like flow-capacity can be easier guaranteed. Large amounts of data are transferred faster when channeled.

Switching disadvantages: Channels, even unused ones, can easily congest or block the network and produce the equivalent of a busy sign on the telephone system.

The act of establishing, working with, and closing such a channel is also called **session**. (see also page 277)

Most network-technologies prefer the routing approach (SCSI, Ethernet, ARCnet, Token Ring). Some network-technologies combine both approaches (ISDN, ATM). In these networks, channel-like paths can be reserved for some kind of packets. That means this technique does **tunneling** or establishes so-called **virtual channels**. Establishing virtual channels with guaranteed Quality of Service (QoS)-characteristics in connectionless networks needs even more organizational overhead.

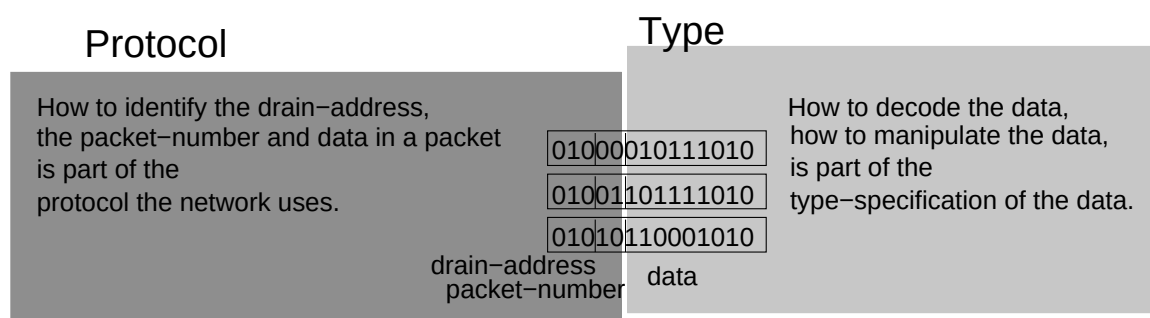
FAX-messages transferred over classical (analog) telephone lines constitute a purely channel-switched way of transferring data.

Concept: Protocols and their Layer-Model

Concept: Protocols and their Layer-Model — Reconsider the notion of a protocol as introduced on page 274. The notions defined there can be immediately extended to networks. The set of rules for establishing network-connections and transferring or exchanging data is called a **protocol**. That includes addressing, data-format of message or data-stream. So, a **protocol** includes all conventions necessary to establish a working communication between two computers of the network, completely independent of any host-computer. Nevertheless, host-computers have to implement and follow the protocol to be able to communicate with each other properly.

Protocol and Type

Protocol and Type — Thus a protocol represents a clear extension of the question that gave rise to the notion of a type of a variable: How is the information coded into a sequence of bytes? The original coding of the — to be transferred — data, together with methods of its manipulation, is called **type** of the data. Below is given a visualization of the difference between protocol and type taking the example of a data-packet:



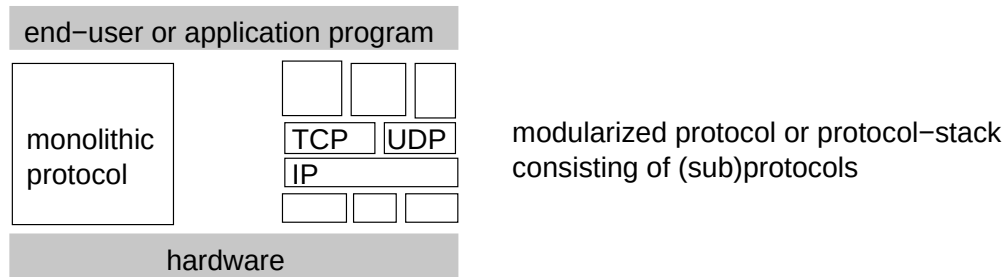
The type of the letter a person writes is roughly characterized by the language used, the protocol used to transfer the letter includes taking an envelope, addressing it, stamping it and knowing how to send it.

In that way a protocol can be seen also as an extension of the type-notion, specialized for the problem of data-transfer. This extension significantly exceeds simple coding and decoding, because usually procedures and processes of transfer-management are part of the protocol. Another view may see a protocol totally unrelated to any type-notion, because it may solely concern itself with transferring chunks of data between communicating entities.

Layers of Protocols

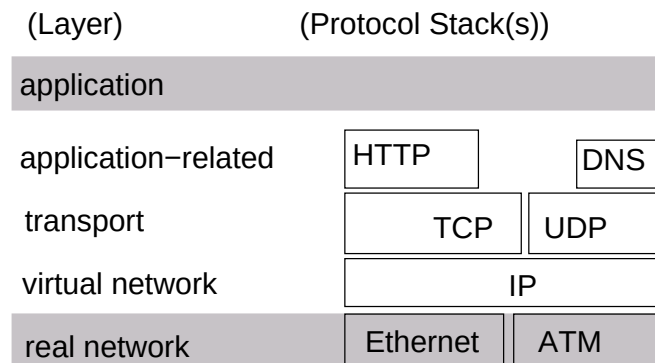
Layers of Protocols — A single monolithic protocol can handle all aspects of a communication-process in a network. But if the (hardware-)technology of the network changes or some user-side functionalities change, then readjusting may become difficult¹⁴³. Rewriting an entire monolithic protocol usually proves more difficult than using a modularized set of (sub-)protocols that organize a connection by working together using defined interfaces. Such modularized protocols, which are meant to interact, are called **open protocols**. For system-programmers and hardware-producers, this modularization makes it possible to reconfigure network-functionalities with relatively little effort and with much more flexibility, as long as they keep the interfaces to other protocols the same (a form of encapsulation, or black box concept. — In this context also recall the description of the platform-independence of Java and its layer-model on page 53.)

¹⁴³This has considerable similarities with the situation programming-languages have to face, when the processor-technology changes. (see page 37)



Therefore a single monolithic protocol usually is substituted by a so-called **protocol-stack** of open protocols. The graphic above already presents the TCP/IP sub-stack which is found in most of today's network-architectures, especially the IP-sub-stack forms the defining part of the Internet.

As already indicated by the graphic above, the protocols of a protocol-stack can be classified according to their position closer to the technological base or closer to the applicational side. This is a first step towards the so-called **layer-model of protocols**:



Again observe that the layer-model, like any other model, cannot communicate the full complexity of the protocol-structures and inter-relations. But the simplified approach should give a first and elementary sketch of the associated software- and hardware-technologies. Such an overview usually completely suffices when writing application-programs in such high-level languages like Java.

Application-protocols communicate on the applicational level over the network, that means:

Information about the application is exchanged together with the applicational data. The user-data, platform-specifically coded, may be transformed to a platform-unspecific code. (That assures a general compatibility between different operating-systems.). Application-protocols usually operate as if they were communicating over a pre-existent virtual channel.

HTTP (Hypertext Transfer Protocol) transfers type-information, the name of the source- and drain-operating-systems and of course the data.

DNS (Domain Name Service) handles the higher levels of addressing and contacting of hosts, this protocols transforms strings like "java.sun.com" into machine-readable network-addresses (IP-addresses like 192.168.0.1). (Just like any modern telephone can be programmed to translate the name of a friend into his or hers telephone-number.)

Transport-(or transmission-)protocols transform applicational data into a form that is transferable over the given network (packetizing, transportation error-check).

TCP (Transmission Control Protocol) breaks data into numbered packets, gets receipt messages about received packets, resends packages if receipt-messages return delayed. Although packets are exchanged, this protocol establishes a (truly) virtual connection. The UDP (User Datagram Protocol) just emits single unrelated packages without much auxiliary checking (no virtual connection is established).

The **virtual-network-protocols** control the routing of the transportation-readied data in a way that is independent of the concretely used hardware network-technology. Such virtual-network-protocols encapsulate the entire network into some easier-to-handle black-box, into which is put an address and the network offers its service to transfer data to that address or receive data from that address, using a unified addressing and routing procedure.

This describes what the Internet Protocol (IP) does.

The concrete transfer of data, then is handled by the physical networks:

Technology-specific network-protocols arrange the data-transport over the actual data-lines and organize the data-transfer between networks of different technologies. Those real network-protocols are bound to the technology that is used with the actual network-hardware.

Catchwords on this level are Ethernet, ARCnet, Token Ring, ISDN, ATM.

Mostly unseen but working behind the scenes are signaling-protocols which organize the addressing and the transport of the actual data within the network. These protocols are only seen by system-programmers.

Some of them are mentioned here only to prove their existence: ICMP, RIP, OSPF, ARP, RARP.

Implemented protocols, closely related to hardware functions, usually are a fixed part of the hardware-design. Whereas implementations of protocols closer to the applicational level, are usually given as programs, also called protocol-handlers:

protocol-
handler

Protocol-Handler: The implementation of a protocol as a program. A protocol-handler furnishes a service-program that helps to manage the communication of computers using a specified protocol. Protocol-handlers for publicly known application-protocols like HTTP and FTP are activated by the first keyword that appears in a Uniform Resource Locator: The call `http://java.sun.com` activates the HTTP-protocol-handler, which contains the software that manages HTTP-connections. (HTTP = HyperText Transfer Protocol: A protocol that manages the transfer of hypertexts in the HTML-format (“HTML” is an acronym formed from “Hypertext Markup Language”). FTP (File Transfer Protocol) implements file-copying services network-wide. Another example may consider a URL of the type `rmi://www.someHost.for/Java/remote/objects`; there the name of the application-protocol is “rmi” which causes a Java Runtime Environment (JRE) to be contacted and a reference to an object to be established.

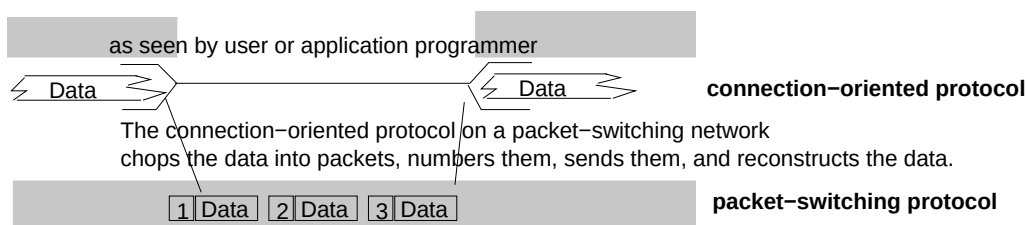
OUTLOOK: Protocol-handling in Java will be indicated rather crudely by the programs on pages 386, 387 and with some generality in the context of the `java.net.URL`-class around page 394.

More Classifying Criteria for Protocols

More Classifying Criteria for Protocols — Protocol-stacks bind two basically different areas of (technical) communication: The network-technology that codes bits into electrical signals and the concept of virtual channels between end-users, who transfer real-life information. Protocols, used within such a stack, may be also classified according to the criteria presented below: Connection-oriented, connection-less, packet-sending, state-bound, state-less, client-server, peer-to-peer.

A protocol may send its data in several smaller packets as indicated in the preceding section or as shown in the graphic below (**packet-oriented protocol**). Alternatively, a protocol may send its information as one entire chunk or the protocol even may be able to manage an entire communication-session, either in a connection-oriented network or through a virtual channel (**connection-oriented protocol**).

The graphic below essentially shows a two-layered protocol-stack: The packet-switched network uses a packet-sending protocol. On top of that low-level protocol has been installed a connection-oriented protocol that uses a virtual channel!



Packet-oriented Protocol, Packet, Frame: The data used for communication has to be delivered in chunks of bytes with a maximal number of bytes. The protocol adds bytes coding drain- and source-addresses and possibly other bytes of control-information (“packing”). The result is given by a chunk of bytes called **packet** or **frame**, to be sent, routed and to be freed from the control-bytes at the destination (“unpacking”). Such protocols may just “pack and send”, “receive and unpack”; but they may also furnish a lot of additional services, like integrity-checks, guaranteed flow of data and the like.

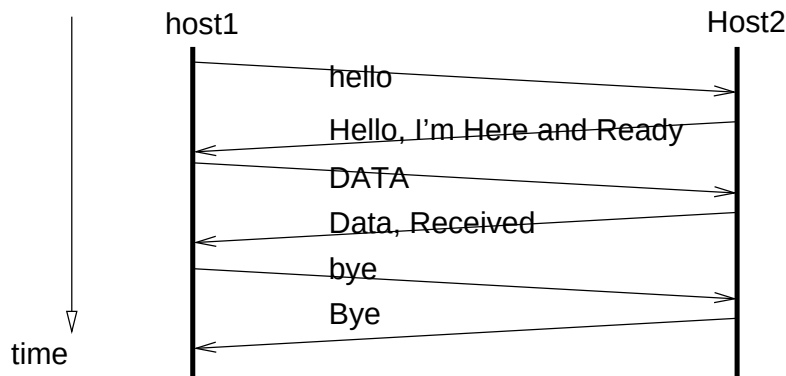
The Internet Protocol (IP) is a packet-oriented protocol, the same as the UDP-protocol. Networks, and their technologically related protocols, almost always are working packet-oriented: Ethernet, ATM, ISDN etc.

Connection-oriented Protocol: These are protocols that check

- whether the destination-host is connected to the network,
- if the destination-host is ready to be involved in a communication-process,
- if the data has been received and they
- tell the destination-platform that there may be more information coming until told otherwise.

Such protocols are called a **connection-oriented protocol**. Connection-oriented protocols always furnish "open" and "close" directives on partner-platforms, "open" for initiation of the connection, implicitly telling that there is more data to come and "close" for telling that the data-transfer is finished. This means that connection-oriented protocols always establish some kind of real- or virtual-channel and possibly manage an entire communication- → session. Connection-oriented protocols either work with a line-switched network or mimic a line-switched network by shielding the complexities of packet-transport from the application-programmer or user (virtual channel).

Connection-oriented Protocol:
Controlled exchange of signalling data and user data.

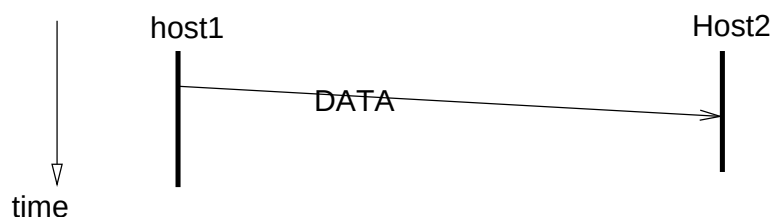


If you are told how to use a telephone, how to dial (+48+local code+number) to a foreign country, what language (French) and what words to use to start a conversation "Hallo, parlez vous anglais?" (Hello, do you speak English?), then you are working with a connection-oriented protocol.

ISDN (Integrated Services Digital Network) and, if in the appropriate mode, also ATM (Asynchronous Transfer Mode), are connection-oriented protocols used for data- and "voice"-transfer. But both transmit data in packets! And both can establish virtual channels with fixed transmission capacities! Another connection-oriented protocol that establishes virtual connections in computer-networks is the widely used TCP (Transmission Control Protocol), which also breaks data down into chunks and sends them as sequences of packets.

Thus, connection-oriented messaging does not exclude packet-oriented transfer on a lower level. And a packet-oriented data-transfer may include features of assuring that the data reaches its destination, but contrary to including some of the services of a connection-oriented protocol, a packet-oriented protocol may also emit its packets into the network without further consideration. The pure notion used for that behavior is "connectionless data-transfer":

Connectionless Protocol:



Connectionless-protocols send appropriately formatted data, that's it. If the data gets lost, then the resending has to be organized by whoever is interested. Generally there exists no overhead of asking whether the destination can be reached or other communication prior to the sending of the data.

Of course, connectionless protocols are simpler, resource-saving but also less reliable than connection-oriented protocols. A unit of data sent by a connectionless protocol is called datagram:

Datagram: 1. Data sent by a connectionless protocol 2. An entity (usually a packet) of data moving inside a network. The only contextual- or control information concerns the network (sender, destination, network-specific security-checkings). There exists no other higher level contextual- or control information, like announcement, receipt, further error checking.

The postal service does not recognize a letter that has been sent as part of a continuous contact between two persons. For the postal network a letter that has been sent is delivered or lost. (Apart from the guarantee that almost every properly addressed and stamped letter has reached the receiver, the postal system grants the Quality of Service of a return receipt.)

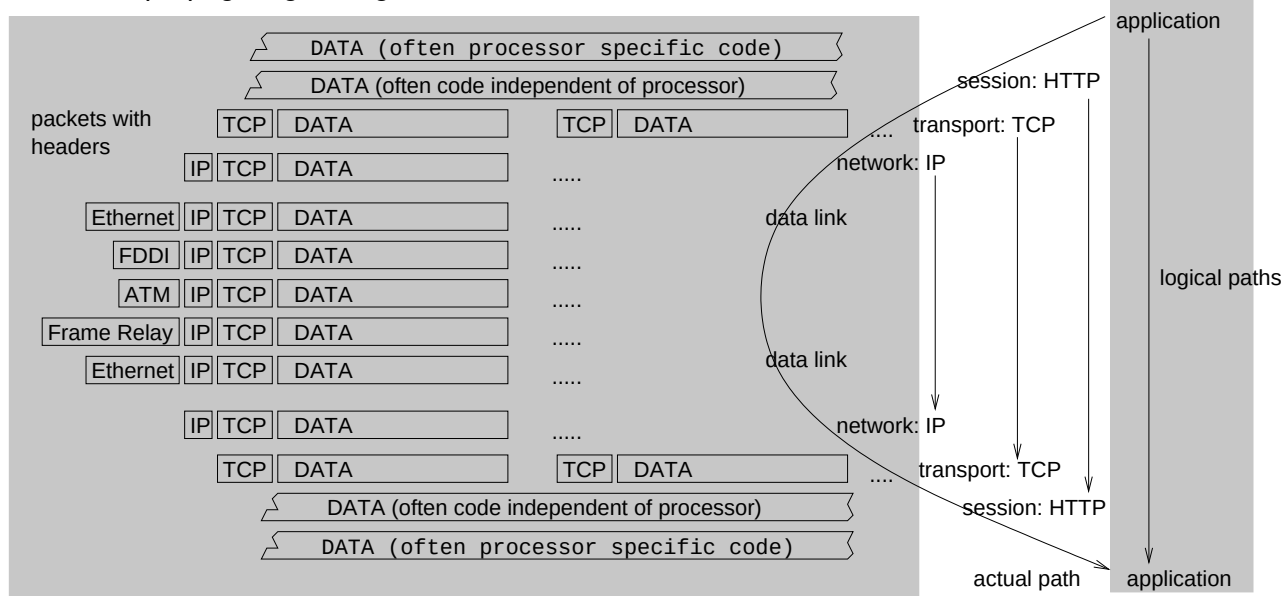
A protocol, that transmits single packets of data, is exemplified by the UDP protocol (Unified Datagram Protocol). This is a widely used connectionless protocol. Its use in Java is demonstrated on page 384.

Some protocols allow the source- or drain-platforms to memorize past actions. Protocols permitting states or working-modes are called **state-bound protocols**. Protocols that cannot memorize past actions are called **stateless protocols**. Stateless protocols are easier and resource-saving, but less comfortable than state-bound protocols. stateless and state-bound protocols

A widely used stateless protocol is the HTTP (Hypertext Transfer Protocol). This protocol is known, at least by name to almost all Internet end-users, because Internet-viewing-software (browsers) use "Uniform Resource Locators" like `http://java.sun.com` that contain the name of the application-protocol HTTP. The statelessness of that protocol makes it problematic to recognize Internet-clients that revisit a shop. A solution to that problem represent Cookies, small pieces of data saved on the client-computer's hard-disk that enable the server to identify the client.

The different transformations, packing and unpacking of the data and its packets during the transfer between two nodes of a network are indicated in the following graphic:

Data propagating through TCP/IP networks:



There can be seen, that the transport-protocol – being a connection-oriented protocol – transfers its connection-related information also in packet-headers in case it uses a packet-oriented network-protocol on some lower level!

Especially the interplay of connection-oriented protocols and packet-oriented protocols makes it possible to introduce the notion of a **logical- or virtual connection**. The real paths of the packets within the mesh of the network are irrelevant to the transfer of data. Therefore a connection (given by the connection-oriented protocol) in a network that routes data is also called a logical connection. Logical connections are the kind of connections seen by an end-user. In the case of packet-switching-protocols a logical connection may not use a single path through the network. Depending on network load, packets may be routed differently but reach their destination altogether.

Furthermore the scheme above reflects the level-structure of protocols: Connection-oriented protocols that initiate the connection first, terminate the connection last (see the application- and transport-protocols above). Packet-oriented protocols that pack chunks of data first, unpack those chunk last. In the graphic above, chunks of data are packed three times until actually sent, and during the transfer they are multiply repacked. If the packet lengths of the technical level differ, then those packets have to be broken down to smaller packets and to be reconstructed when leaving that level (the breaking down of packets into smaller ones is called segmentation). So there is much work done on those low levels, hopefully unseen by application-programmers and users.

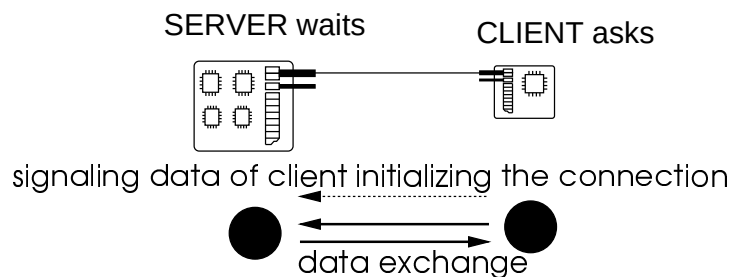
Client-Server, Peer-to-Peer Communication

Client-Server, Peer-to-Peer Communication — Remember the basic communication-concept of client-server- and peer-to-peer-computing as described on page 275. Although network-connections usually allow bi-directional data-transfer, a logical network-connection can be characterized directionally according to the way it has been initiated: Communication often implies a kind of asymmetry; initiating the communication by addressing a network-node is the first asymmetry to be observed. The same asymmetry has already been described as the client-server-model was introduced on software-level:

A CPU (Central Processing Unit) or rather a communication-software on that platform that waits for a communication to be established (often by more than one CPUs) makes the entire computer be called **server**. Servers offer some kind of information or service, in that sense servers are somewhat passively — waiting.

Those CPUs that do the contacting, often need some data and therefore initiate the communication. Such contacting CPUs are also called **clients**. Once a connection has been established, both partners can send and retrieve information, ideally having equal rights.

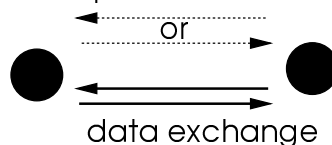
(Back up the server with a database-hosting computer which is consulted by the server when sending out information, then this setting (client-server-database) is also called three-tier-model. Those cases are managed preferably by Java servlets within the context of programming with the Java 2 Enterprise Edition (J2EE).)



If each computer is capable of calling the other computer to initialize a communication-process, then that kind of communication-setting is also called **peer-to-peer communication**.

Peer-to-Peer Communication

any of the network computers can initialize the communication



Peer-to-peer communication is also characterized by the same communication-software on both sides and same rights on both CPU's that communicate.

Example: The telephone system is a typical peer-to-peer network. Each standard user has the same means to contact any other user. — But even here can be seen client-server-structures: If you call the information-desk at a telephone company for a telephone number of an acquaintance of your's, then you get the relevant information delivered by a client-server structure; you initiate the connection, the company waits for that connection and transfers the data. Call-centers on the phone-line system also act more as servers, they wait for active clients to call and fetch information.

The last part of the example shows that the client-server or peer-to-peer networking are seldom, if never, realized as a pure concept.

Peer: 1. Communication-partner with the same rights and restrictions to initiate a communication-process as every other communication-partner. Contrary to client-server-communication, in which the server waits until contacted by the client. In client-server-communication only the client has the means to initiate a communication-process. After the initiation both partners can exchange information freely.

2. In the layer-model of networking: Functional unit in the **same layer** of a layer-model. Peer-communication on hardware-level of networks is supported by bridge-hardware. On the level of protocols, peer-communication is made possible by gateways. Whereas software that enables applications to communicate is called middleware.

3. In other layer-models: Corresponding pieces of software in **different layers**: For example, considering the layer-model of platforms as introduced by the graphic on page 53, with the Java Abstract Window Toolkit (AWT): The native screen-components (peers) are associated with the Java specification of → screen-components and used for on-screen representation. Those native-peers define heavyweight screen-components. Lightweight screen-components use graphics-routines from native-toolkits to be drawn and thus have no native-peers. (For more about heavyweight- and lightweight screen-components see page 548.)

Examples for such concepts cast into Java source-code can be seen on page 384 for a case of peer-to-peer communication and beginning with page 385 for a case of client-server-contacting.

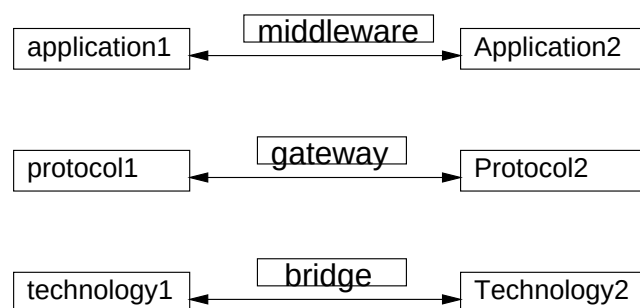
Communication within the Same Levels of the Layer-Model

Communication within the Same Levels of the Layer-Model — Above have been introduced already three additional, frequently used, notions that can be described using the layer-model:

Middleware: Software that manages **inter-application communication** (this even includes application-related programming-languages). Middleware doesn't cover the notions of an operating-system or a network-protocol, neither that of an application.

Gateway: Software for translating data-formats for exchange between different protocols. The protocols are assumed to offer about the same services.

Bridge: Hardware that translates data between different networking technologies.

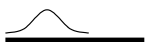


The layer-model can be refined to variants of the so-called OSI layer-model (Open Systems Interconnect) which won't be used here further, but nevertheless is presented due to its wide use. So the next section can be seen as a further illustration of the networking-notions presented so-far:

OSI-related Layer-Model

OSI-related Layer-Model — In a network, computers exchange information in form of byte-sequences. This simple sentence bears a technical overhead of considerable complexity. That complexity should have been hinted at by the concepts introduced so far. A commonly accepted layer-structure to describe the functionalities of networks is given by the OSI (Open Systems Interconnect) layer-model. This model extends the layer-structure which already showed up in the graphic on page 348.

physical layer



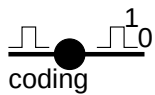
physical wire

and physical signal

One of the first things to consider with a network is to know in **what physical form** of the information is to be transferred: Electrical currents in copper wires, light-pulses in glass-fibers, electro-magnetic waves in space (radio-signals)? Typical problems to be solved on this level are:

What frequencies guarantee a high undisturbed flow of coded information, what cable-design (twisted-pair, coaxial) furnishes a high Quality of Service? How to effectively design plugs and antennas? These problems keep entire research-institutes and associated industries going! This technical level has little to do with bytes bearing coded information, it just searches for the means to efficiently physically code and transfer sequences of bit-signals. This is also called the PHYSICAL LAYER.

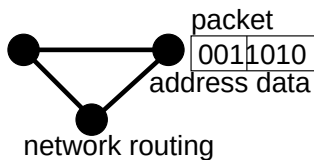
data link layer Of course, the physical context is chosen mainly in considering the coding of bits and bytes:



How can bits of information be modulated into the physical means of transportation? How can the information coded as bits and bytes be transformed between different byte-signal coding-schemes? And maybe how to detect transmission errors on signal-level? These questions of **coding and transforming bits and bytes when sent as physical signals** produce answers like hardware specifications: ATM, FrameRelay,

FDDI, Ethernet, TokenRing, ARCnet, UMTS, Bluetooth and others. A hardware device that translates information-signals of such different specifications is called a **bridge**. Hardware devices that connect USB-ports of computers with ISDN-telephone lines are typical bridges, of course there are also Ethernet- and ARCnet-bridges and the like. This area of interest is also called DATA LINK LAYER.

network layer



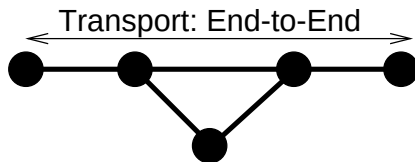
How are the bits and bytes sent through the network?

How does the information find its receiver? Which addressing-scheme, which switchboarding-scheme (channel-switching, packet-routing) is used on hardware level? Functioning **addressing and transfer of chunks of bits and bytes from a source to a drain** are also given by the previously mentioned hardware specifications like ATM, Ethernet and ARCnet.

Devices that implement these network-functions are called **routers**.

Networks of a different technological base, can be connected by bridges and made a single virtual network with an appropriately implemented network-protocol like the Internet Protocol (IP). (For communicating information about the network (topology, available capacity, shorter paths) routers have their own protocols, like the Routing Information Protocol (RIP) or the Open Shortest Path First Protocol (OSPF)) This level is also called NETWORK LEVEL.

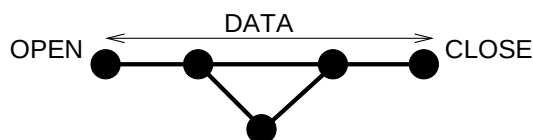
transport layer



Here is assumed that a kind of VIRTUAL CHANNEL HAS BEEN ESTABLISHED, even if that channel furnishes only sequences of data-packets. Here, actions and protocols try to **transform the received chunks of bytes into files or data-streams and reverse**, maybe checking the integrity of the data too. The connectionless User Datagram Protocol (UDP) sends and receives single packets (datagrams). The Transmission Control Protocol (TCP) breaks files into numbered

packets of bytes and sends them. On receiving packets, the TCP tries to sequence them properly, reorder lost ones, and to reconstruct the original file. This level is also called TRANSPORT LAYER.

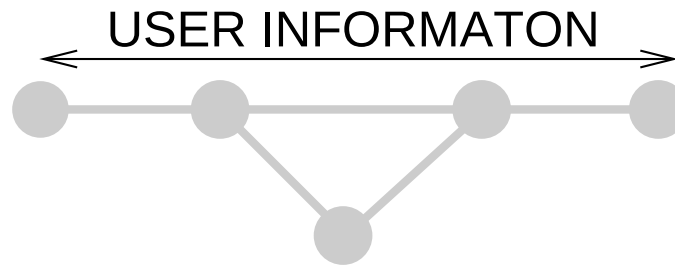
session-,
presentation
layer



The next field of consideration concerns itself with **establishing and closing down a (virtual-) channel** and in the mean-time, with transferring files and streams of data through that channel. The data may be processed to make it independent of the computer's processor or the data may be treated with cryptographic routines for security purposes.

Files or streams are transformed with respect to their data-format or their type (MIME (Multipurpose Mail Extension) types like HTML, PDF). Protocols on this level are the Hypertext Transfer Protocol (HTTP) or the Secure Hypertext Transfer Protocol (SHTTP). Or the File Transfer Protocol (FTP). Even the Domain Name Service (DNS) works on this level. This range of interest may be circumscribed with the denotations SESSION LAYER OR PRESENTATION LAYER. Protocols like TELNET or Java's Remote Method Invocation (RMI) even may dispatch commands that affect the operating-system or the Java Runtime Environment (JRE) of a remote computer; these two protocols also may be associated with the next layer:

Example: The transport level of the commonly used telephone system incorporates telephone books and the dialing-process until the line is established, then waiting for someone to answer and identifying oneself. "Is there somebody?" ... "Good Bye!" And "Do You understand English?"



The last level covers the aspects of networking that may be of interest for an end-user or a programmer who wants to use the **computer-network as a tool**. This includes applications like Internet-browsing, email-clients, FTP-client usage, streaming videos or music and more. This end-user-related layer is also called APPLICATION LAYER.

Example: Telephone system: "This is the message."

This layer-model only serves as a guideline. It presents a suggestion for categorizing the notions encountered in networking. The network layer for example may be divided into an hardware network layer (Ethernet) and a layer of virtual networks (IP). Layer-models are helpers and do not represent eternal truths. See page 367, for another visualization of network-layers.

application
layer

Networking-Protocols

Networking-Protocols — Programming with Java in a network-related context may explicitly involve aspects of the following protocols: The addressing of the Internet Protocol (IP), so-called ports of the Transmission Control Protocol (TCP), packets of the User Datagram Protocol (UDP), the association of IP-addresses and domain-names realized by the Domain Name Service (DNS), the HyperText Transfer Protocol (HTTP) for retrieving mostly HTML-text-files, the File Transfer Protocol (FTP) for copying files across the network, maybe also the Simple Mail Transmission Protocol (SMTP) for sending emails and the Post Office Protocol (POP) for receiving emails. Transport-protocols like the TCP (Transmission Control Protocol) and the networking-protocol IP (Internet Protocol) of the TCP/IP protocol-stack are usually incorporated into the operating-system. The Java programmer has contact only with the related addressing-schemes (IP-addresses and URL (Uniform Resource Locator)) and the Java-sockets which are the software-objects that actually establish the network-connection to other computers.

The following chapter gives no specification of any of the mentioned protocols, there exist books or sets of books on each of these protocols. Only the most relevant aspects for Java application-programming are mentioned, this predominantly includes the consideration of addressing-schemes on networking- and transport-levels. For more information on protocols see also: ➡ <http://www.w3c.org/protocols>.

The Internet Protocol (IP) — a packet-routing-, connectionless Protocol

The Internet Protocol (IP) is a **packet-routing-, connectionless protocol**. It is a protocol on network-level. Where does an end-user see the Internet Protocol? A, to be introduced, Uniform Resource Locator (URL) of the kind `http://192.168.0.6/` contains an IP-address given by the four-digit-number! So, not only application-programmers encounter IP-addresses but also pure end-users can encounter them.

The Internet Protocol represents underlying network-technologies as a **virtual network**. This virtual IP-network has an own addressing concept and gives the user the impression of an automatic routing of IP-packets in which data can be transferred.

This means that the protocol works **independent of the concrete network-technology** and therefore can be used together with different kinds of network technologies (after having been suitably adapted by “native” software!). Unseen on this level, are those different kinds of networks connected by bridges; but this protocol hides the technological detail and lets treat this technologically inhomogeneous network as if it represented one network. Therefore the name “Internet” Protocol.

IP-Addressing of Hosts

IP-Addressing of Hosts — There are two versions of the Internet Protocol: IPv4 (Internet Protocol Version 4) and IPv6 (Internet Protocol Version 6). Both versions differ considerably in package- and addressing format. An application-programmer might see this difference primarily in the different form of addressing and maybe in some better Quality of Service (QoS) of IPv6 connections.

(Data-Transfer using packets) Usually a sequence of bytes that represent data is packed into a frame, that means the sequence is given a header of some bytes that contains the address of the receiver and the number of data-bytes.

An IPv4 packet has a header of about 20 to 60 bytes and a maximal payload of 65,515 data-bytes. (A two-byte-value in the header of the packet codes the length of the packet.)



(Addressing in IP-Networks) Each computer in an IPv4-network has at least one unique four-byte address. The address of the destination-computer is found in the header of an IP-packet. Parts of IP-addresses fix entire networks:

IP-addressing identifies individual computers in an IP-network

IPv4-address (4 byte address)

For example: 11000000.10101000.00000000.00000001 (binary)

C0.A8.0.1

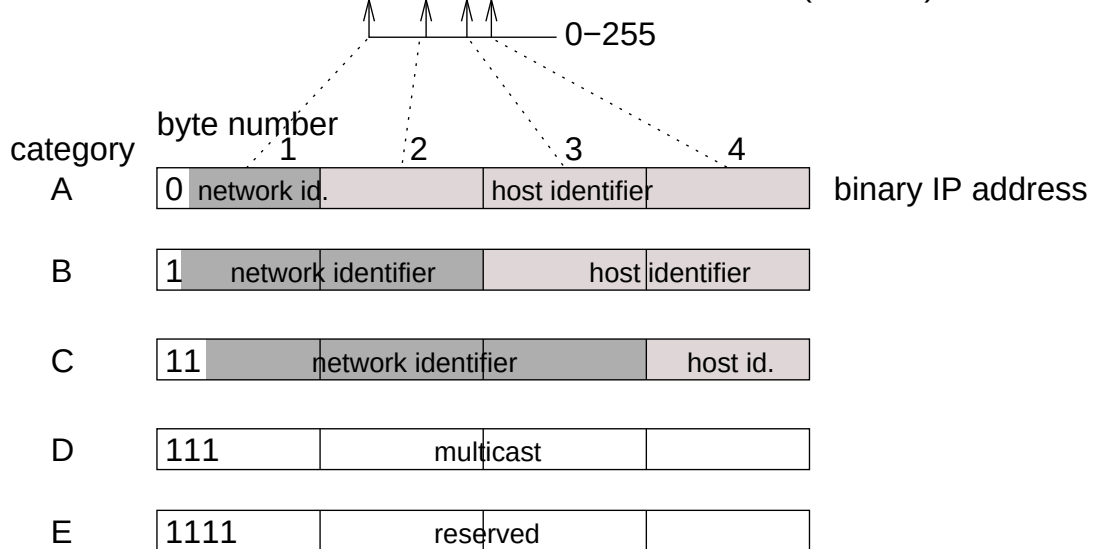
(hexadecimal)

300.250.0.1

(octal)

192.165.0.1

(decimal)



- The IP-addresses beginning with decimal 0_{10} (dual 00000000_2) and 127_{10} (dual 01111111_2) are reserved, therefore there remain 126 “networks” addressable with class A addresses.
- All IP-addresses with the **last byte** equaling the decimal 0_{10} (dual 00000000_2) or 255_{10} (dual 11111111_2) are reserved and therefore there remain 254 possibilities out of 0 - 255 ($2^8 = 256$ possible choices)
- All Internet-addresses beginning with decimals 10_{10} (dual 00001010_2) and 192_{10} (dual 11000000_2) are unassigned and therefore can be used in building local IP-based networks without addressing conflicts when accessing the Internet.
- The IP-address 0.0.0.0 is an alias to the originating host, IP-addresses beginning with 0.0. refer to a host on the same local network.
- An ordered range – an interval – of IP-addresses can be given as follows:

interval of IP-addresses	can be coded as	IP-address and IP-mask
192.168.0.100		192.168.0.100
...		and the subnet
192.168.0.150		255.255.255.205

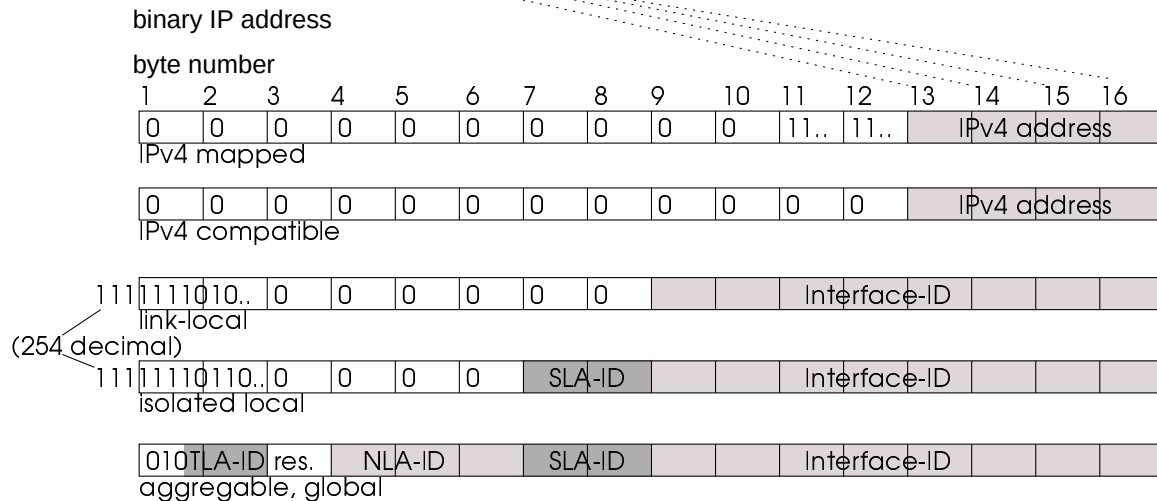
The formula for this kind of strange offset procedure, when describing intervals of IP-addresses, is given bitwise in decimal notation:

$255 - (\text{byte of end-address} - \text{byte of start-address})$. That strange notation enables a fast calculation on the binary level by using logical operators. The reason for introducing subnet-masks is to change the host-bytes of class A and class B IP-addresses into network-bytes. This essentially helps in a more effective IP-packet-routing and helps especially in keeping locally sent IP-packets inside local networks. These considerations should not bother any application-programmer.

An extension of the IProtocol is the Internet Protocol IPv6 that uses 16 byte addresses:

IPv6-address (16 byte address)

For example: 00000000 00000000:00000000 00000000:
 00000000 00000000:00000000 00000000:
 00000000 00000000:11111111 11111111:
 11000000 10101000:00000000 00000001 (binary)
 ::FFFF:C0A8:1 (hexadecimal :: = :0:...:0:)
 ::255.255:192.165:0.1 (decimal) 0-255



TLA, NLA, SLA = Top-, Next- and Site-Level-Aggregator

IPv4-addresses are 32-bit (=4 **byte**) **addresses** and can at best address about four billion computers ($2^{32} = 4,294,967,296$). The new Internet Protocol (IPv6), that is backward-compatible with the present one and uses 128-bit addresses (=16 **byte** **addresses**) and thus can address more computers that are practicably imaginable right now ($2^{128} \approx 3.4 \cdot 10^{38} \approx 2.06 \cdot e^{88}$). Detailed information on IPv6 can be found in the document RFC1725 or by referencing the Internet-site <http://www.ipv6.org>.

ORGANIZATION

A person or company can have assigned IP-addresses by the Internet Assigned Numbers Association (➡ IANA <http://www.iana.org>), overall information is given by the Internet Corporation for Assigned Names and Numbers (➡ ICANN <http://www.icann.org>) or by local Internet Service Providers (ISP) who have purchased blocks of IP-addresses. The central institution IANA assures that each IP-address denotes a unique CPU worldwide!

IP-Address or Internet-address: Sequence of numbers assigned uniquely to a computer that implements the Internet Protocol (IP). Then this computer can be connected to any sort of network that has an Internet-compatible level of software-communication. An IP-address is an arch-example of a → host-name.

The Java classes inside the package `java.net` are pre-written exclusively for the Internet Protocol (IP). The remaining questions concern end-to-end aspects of the communication between CPU's:

IP-Addressing of Hosts: Domain Name Service (DNS)



IP-Addressing of Hosts: Domain Name Service (DNS) — Reconsider the example of the telephone-system. Each telephone has one standard telephone-number under which this telephone can be reached. (Of course, a telephone may have been assigned – and may be reachable under – other telephone-numbers too.) But dialing one telephone-number, usually triggers no conference-call, and usually does not activate a forwarding-service, but lets the user establish a connection with exactly the other telephone, which has been assigned the indicated number. That relation of telephone and telephone-number exemplifies the relation of host and host-name in a network of computers. Now consider a programmable telephone, which can associate names of persons with telephone-numbers. But names of persons do not identify a telephone, the name of a person may be associated with a mobile-phone, with some wired phone at the work-place and with a wired phone at

home. Such a programmable telephone may automatically try to contact one of these phone-connections and if nobody answers, tries another number until the connection to the person has been established. A coarse equivalent to this, represents the Domain Name Service on the level of an IP-network (name relates to domain-name, telephone-number to IP-address).

host-name

Remember the definition “host-name of a computer inside a network” on page 339. A **host-name** is a name for one computer (host) in a network of computers; this name makes it possible to identify the computer inside the network. Nothing prevents a host to use several host-names as long as the use of each individual one of them suffices to identify the computer.

Individual computers in an IP-network, like the Internet, can be identified by a given IP-address, a concrete form of a host-name. For example computers with two Ethernet-cards may be given two IP-addresses. In an IP-network, each computer has a different IP-address, two computers inside the network should NEVER use the same IP-address. This makes the IP-address a typical host-name. But IP-addresses, like

130.6.5.123

usually are more difficult to memorize than names or words given by strings:

www.sun.com

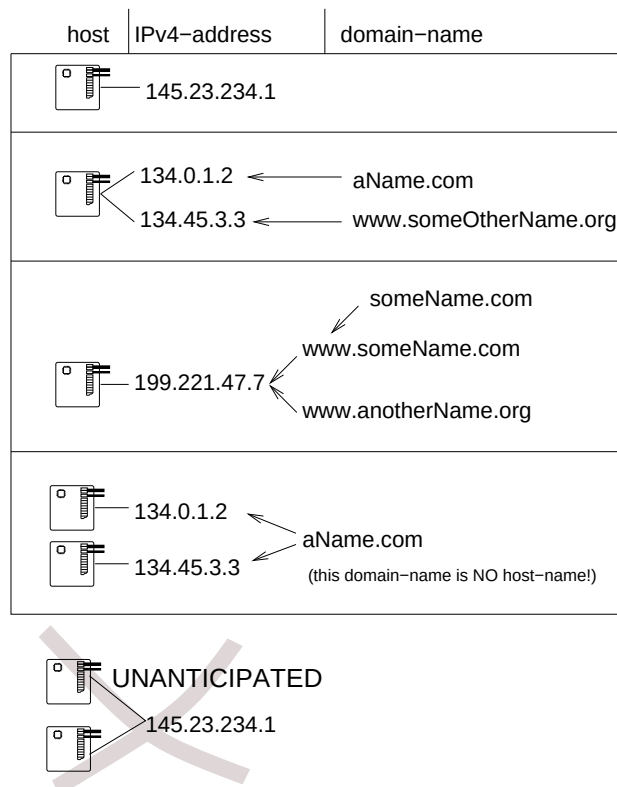
domain-name

So a translation service (like that of a telephone operator's) has been established which uses the network's resources (servers and network flow capacity) to TRANSLATE AND RETRANSLATE IP-ADDRESSES TO SO-CALLED **domain-names**. In the example above, the substring sun.com is called domain-name, www is called a subdomain (indicating the World Wide Web services, see page 372) and the entire string also is called a domain-name. In some cases, domain-names are used without subdomains, for example http://berkeley.edu, is forwarded by the DNS-name-server to http://www.berkeley.edu.

Institutions, mentioned later, register those names that can be mapped onto IP-addresses like www.sun.com and associate the respective IP-addresses with the registered domain-names. Coarsely, the Domain Name Service (DNS) is formed by a group of decentralized repositories or registries (a distributed database) of so-called name-servers with tables of IP-address and domain-name associations¹⁴⁴:

Host, Host-Name and Domain-Name

EXAMPLES:



¹⁴⁴Regular Internet-servers usually also offer those services that make them name-servers too. The Linux operating-system usually uses a name-serving software called bind to furnish DNServices. An average Internet-user actually never should see that software.

The graphic above shows host-computers in an IP-network which can be given a host-name in several different ways. As mentioned before, each host or node in an IP-network has to have at least one IP-address. Hosts, especially those with multiple network connections may carry multiple names, indicated by multiple IP-addresses. But two hosts NEVER should be given the same IP-address.

- Some CPUs connected to the Internet need no domain-name, for example a personal computer connected only temporarily and directly to the Internet via an Internet Service Provider (ISP). These computers, temporarily are given an Internet-address; and as long as they are connected to the Internet, they can act as a fully-functional IP-host. As temporary client-computers, they do not need to act as fully-functional Internet-servers.

Domain-names, which are supposed to be easier memorizable, can be used as host-names. In the case of an IP-network, DNS-name-servers run software configured to associate domain-names with IP-addresses. As can be seen in the graphic, this association leaves some freedom:

- The case where a computer has many different IP-addresses occurs usually with servers of Internet Service Providers (ISPs) that administrate the Internet-sites of many different customers. There, each customer appears to be present in the Internet with an own domain-name, thus apparently having a sever of their own. CPUs with multiple IP-addresses act as a virtual server-set.
- Some CPU may have a single permanent IP-address, enabling the computer to act as a server, but that CPU may have multiple domain-names attached. The Domain Name System even may associate different domain-names with another.
- Internet-sites, which are in high demand, often let the Domain Name Service (DNS) distribute the clients requests across multiple servers which offer identical information-services. Thus the DNS makes one domain-name pointing to several IP-addresses, distributing requests alternately to the different hosts. In other words: Frequently visited Internet-sites sometimes let the DNS-server distribute the load of client-contacts onto several hosts offering identical ("mirrored") sites. This is the case where a domain-name clearly is no host-name!
- But the same IP-address should NEVER be assigned to different hosts!

domain-name is
no host-name

As already indicated in the examples, the names in the Domain Name System can be given a hierarchical structure as follows¹⁴⁵:

Domain-Name := { optionalNameExtension. } Name . TopLevelDomain(TLD)

(Upper- or lower-case letters can be used, but are not discerned.) An example of such a domain-name hierarchy is given by the domain-name of the Internet-SETI project: "setiathome.ssl.berkeley.edu" with two subdomains. The Top Level Domains have to be chosen from a rather restricted spectrum of possibilities: The Top Level Domains fall into two groups: Generic top-level-domains (gTLD) and country-code-top-level-domains (ccTLD). The generic ones are:

edu for US educational institutions,
gov for US governmental institutions
mil for US military Internet-sites and
int for international organizations. Whereas those that can be used relatively free
com is suggested for denoting commercial sites,
net for network-related sites
org for non-commercial sites
aero aeronautical sites, air-transport industry
biz is suggested for denoting commercial sites
coop to be used by cooperatives
info general use
museum for museums
name for private persons

¹⁴⁵Brackets "[,]" indicate zero or one occurrences, braces "{,}" indicate zero or more occurrences.

pro for accountants, lawyers and physicians

There are a lot two-lettered country-code-Top-Level-Domains (ccTLD) like "ca" for Canada, "uk" for United Kingdom, "fr" for France, "de" for Germany etc. (A list of those country-code-Top-Level-Domains can be found at <http://www.iana.org/cctld/cctld-whois.htm>).

Optional domain-names, also called Second Level Domains (SLD), may denote a concrete institution and subdivisions independently from any underlying network-structure. As could be seen above, even the domain-name need not identify an individual computer. A low-level domain like "www" may identify the special services rendered by a server, here the services of the World Wide Web (a collection of application-protocols and application-related services mentioned later) or like in the domain-name "setiathome.ssl.berkeley.edu" the service for a project of distributed computing.

ORGANIZATION

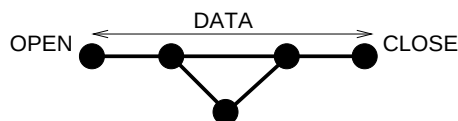
Where can be found the owner of a registered domain-name? Search for WHOIS¹⁴⁶ services in the Internet or use a WHOIS-client like the one on page 382. How can be seen the IP-address properly resolved for a given Domain-name? See for sites that do a DNS Reverse Lookup or a Reverse IP Lookup or consider the Java program on page 382. From where to get domain-names unless given by the local Internet Service Provider (ISP)? See the ➡ Internet-sites <http://www.dns.net> and <http://www.iana.net> or <http://www.icann.net>.

Domain-Name: A computer-network, with a functional Domain Name Service (DNS) offers the use of domain-names for identifying host-computers. Domain-names identify computers of a network. A domain-name is given by a string of characters, usually representing a human-memorizable name: "www.sun.com", "java.sun.com" or "cnn.com" are domain-names. The format of a domain-name has been given a few paragraphs ago (case-independent, ASCII-characters, ...).

Host-names identify hosts, domain-names identify hosts too, but there are cases where the DNS lets one domain-name channel requests to different hosts. Thus domain-names need not be host-names!

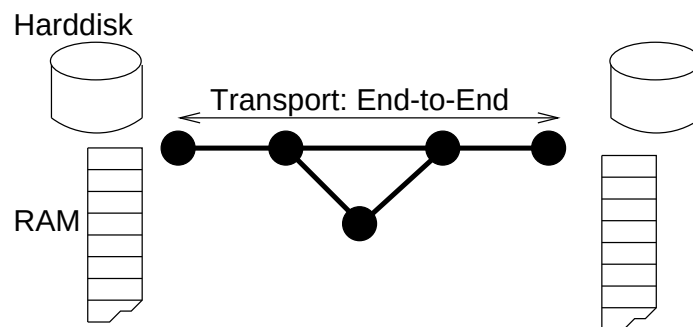
For more information about the conventions governing the use of domain-names, see <http://www.dns.net> and <http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1034.txt> (page 10).

TCP/UDP Transport-Protocols



TCP/UDP Transport-Protocols – Having the communication-lines, physical connections between the lines and the hard- and software functionality that routes the data along the lines of a network over bridges to another network; having especially the (virtual) homogeneous IP-network; then some new basic questions may sound like: What commands are used to dispatch data or how to request

the sending of data? How has the data to be converted to be able to dispatch it; especially in packet-oriented communication. And which commands chop the given data into packets? How is the data received and has it to be converted to be used by the destination-CPU? Exists there a way to check the integrity of the data? Especially in packet-oriented communication, can lost packets be identified and requested for re-transmission? How are the packets reassembled and unpacked when reconstructing the original data?



Such questions are answered by transport-protocols like TCP (Transmission Control Protocol) and UDP (User Datagram Protocol):

¹⁴⁶The WHOIS service answers the question "Who is?".

TCP - The Transmission Control Protocol — connection- and packet-oriented

TCP (Transmission Control Protocol) acknowledges received datagrams (=packets), re-orders lost ones, sorts the datagrams and unpacks them to rebuild the original file. TCP is a connection-oriented protocol, it makes both communicating computers wait for the other to send more data or to send a closing signal. Java programmers may concentrate on being knowledgeable of addressing specifications; these are presented in one of the next sections.

UDP - The User Datagram Protocol — connectionless and packet-oriented

The UDP (User Datagram Protocol) just specifies how to send and receive IP-packets of data. There exist no means for the checking of data-integrity, no numbering or ordering, no time-checking included in the specification of that protocol. This can make packet-exchange less time-consuming, but error prone. Packets here are independent messages that are sent and received without controlling their relation to each other. This protocol is used to transfer real-time-data like images or speech, that means life audio- and video-data. UDP is also used for sending time-signals or testing the quality of the connection on software-level by sending packets and checking whether they all reached the receiver. Java also enables the programmer to send and receive single IP-packets, using the following addressing conventions:

Addressing in TCP/IP-Networks: Sockets and URLs

Addressing in TCP/IP-Networks: Sockets and URLs — The denotation TCP/IP generally includes the UDP, for remembering the proper positioning in the protocol-stack consult page 348. — In an IP-network each computer (CPU, host) has its unique address. In a TCP/UDP context each IP-address is supplemented by a two-byte number, giving 65536 sub-addresses¹⁴⁷(0-65535). These are purely logical sub-addresses meant to enable a preselection of incoming packets. The two-byte number resembles the address of a post-office-box on the host-computer. The two-byte numbers are also called **ports** or **TCP/UDP-ports** (no physical ones like the serial or parallel ports of a computer, only logical ones. Ports have been introduced on page 231). Many of the TCP-port numbers 1 to 1023 are reserved for standard services (realized by application-protocols). For example the application-protocol HTTP (Hypertext Transfer Protocol) usually communicates on port 80 (seldom on port 8080). The application-protocol SMTP (Simple Mail Transfer Protocol) uses port 25. Especially server-CPU's can relocate client-connections to different ports, so that servers can manage many connections (quasi-)simultaneous (usually server-software relies heavily on multithreaded programming techniques).

So, addressing host-computers in TCP/IP-networks, involves

IP-addresses (possibly substituted by the use of DNS domain-names),

TCP/UDP-ports that represent logical post-office-boxes on a host-computer and finally

path-names in the local Hierarchical File System that refer to single files. (see page 47)

All these information will be compiled into a so-called **Uniform Resource Locator (URL)**, which is introduced below.

(IP-ADDRESSING) Addressing in IP-networks has already been discussed, there the idea was to identify host CPU's in an IPv4-network using **four-byte** IP-addresses (199.23.4.1) (in IPv6-networks **sixteen bytes**) or letting a DNS-name-server translate a registered domain-name¹⁴⁸ into such an IP-address. The TCP/UDP-protocol extends this addressing-scheme:

(TCP/UDP PORTS) Adding a TCP/UDP-port-number to the IP-address usually specifies a TCP-input-channel, just like a mail-box number. Remember that TCP/UDP-ports are identified by a two-byte number and that, out of the 0-65535 range, some ports of the first hundreds have been reserved as channels for standard application-protocols (port 80 denotes HTTP):

¹⁴⁷TCP/UDP-ports, coded by two bytes (16bits) of information, introduce a set of 65536 ($= \overbrace{2 \cdot 2 \cdot \dots \cdot 2}^{16 \text{ times}} = 2^{16}$) virtual parallel channels on each computer in the network.

¹⁴⁸For example `www.aname.com`, maybe plus sub-domains like `www.subdomain.aname.com`.

TCP/UDP-port	Encoding	Application-protocol	Service rendered
13	tcp/udp	daytime	gives ASCII representation of current time on the server
20	tcp	ftp-data	channel for exchanging the contents of files
21	tcp	ftp	channel for exchanging FTP-commands (signaling)
23	tcp	telnet	interactive remote command-line sessions on a virtual terminal
25	tcp	smtp	exchange email between CPUs that are permanent hosts
43	tcp	whois	Who Is gives more information on IP-addresses, domain-names
80	tcp	HTTP	load HTML-text-files with links and multimedia-elements
110	tcp	pop3	transfer accumulated emails to clients with sporadical IP-connection
119	tcp	nntp	post email like messages publicly in newsgroups

There are many servers that have only a few ports active, mostly the port 80 for HTTP-connections! On computers working under Unix/Linux/BeOS operating-systems, listings of tcp-ports are found in the directory `/etc/services`. (A list of all assigned TCP-port-numbers can be found at <http://www.isi.edu/in-notes/iana/assignments/portnumbers>) Figuratively speaking: If the IP-address is to your computer what a regular street, city, zip, state-set is to an apartment-house, then the TCP-port number added to an IP-address is the apartment-number of the house.

The combination of IP-address and TCP/UDP-port-name is also called **TCP/UDP-socket**¹⁴⁹.

$$TCP/UDPSocket = IPAddress \text{ (or } DNSDomainName) + TCP/UDPPortNumber$$

A host-computer with an operating-system that furnishes a working TCP/IP-sub-stack and an Hierarchical File System (HFS), makes it possible to address individual files, even markers in files:

(URL-ADDRESSING IN TCP/IP-CONTEXT) Addressing in TCP/IP-networks can be even more precise if the operating-system of the computer-platform has implemented a Hierarchical File System (HFS) (like BeOS/Linux/Unix, Mac OS, Windows), then single files can be addressed network-wide with the following addressing-scheme: A path-name of the Hierarchical File System with a file-name at its end fixes that specific file on a host-computer's persistent storage-device, usually a hard-disk. Observe that, **folder-separators of path-names have to be substituted by slashes** when used within the URL-notation! The application-protocol that should be used with this file is prefixed to the IP-address or the TCP-socket as follows¹⁵⁰:

```
URL := applicationProtocol :// IPAddressORHostName [: portNumber ]/
      HFSpahtSlashSeparated / fileName [# markerInHTMLFile ]
```

Such a construction is called a **URL**, a **Uniform Resource Locator**. (The suffix `#markerInHTMLFile` is used to jump directly to a marker in an hypertext in HTML-format, under the HTTP-application-protocol.) Again, for more examples of Uniform Resource Locators see the list following page 373.

Usually URLs are the only way end-users need to concern themselves about addressing on the Internet. Uniform Resource Locators can also occur as "`http://www.someName.com`" because in this notation the server automatically loads the default page "`http://www.someName.com/index.html`" of the Internet-site of organizations or persons. For some further illustrations and examples of URLs see page 373.

➡ Further information about Uniform Resource Locators can be found in <http://www.ncsa.uiuc.edu/demoweb/url-primer.html>. The HTTP-protocol and the HTML-text coding type are presented in the following sections.

¹⁴⁹A broader idea of the various meanings of "socket" has been given on page 273.

¹⁵⁰Brackets "[,]" indicate zero or one occurrences, braces "{,}" indicate zero or more occurrences.

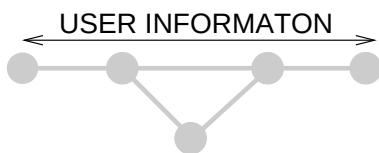
Uniform Resource **Locator** (URL): An identifier for uniquely naming and locating a resource within a network (like the Internet). A more concrete description has been given in the preceding paragraphs.

Uniform Resource **Name** (URN): Identifies a resource (file or stream of programs or data) in a network or on a computer. URNs are flexible enough to ALLOW RELOCATION OF THE RESOURCE without changing the identifier. A sort of an URL-alias.

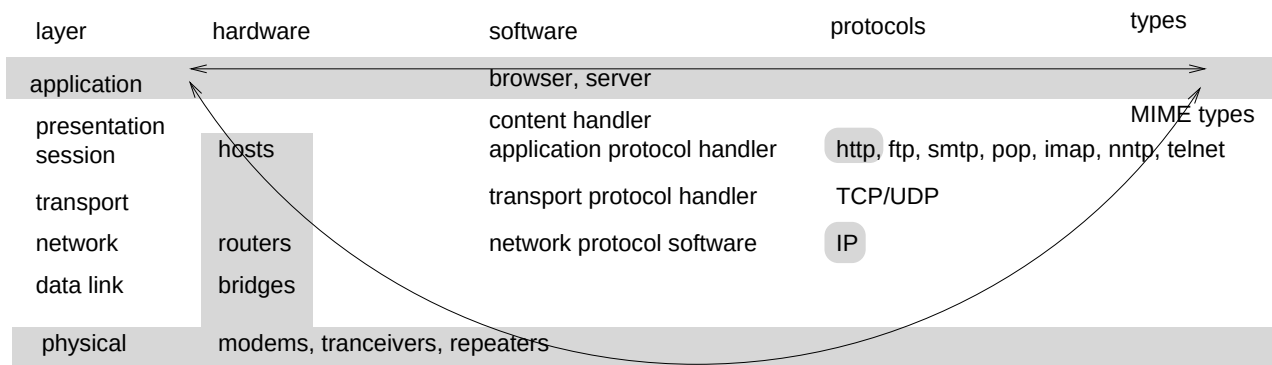
Uniform Resource **Identifier** (URI): Describes the sheer concept of addressing whatever kind of resources by using a string of characters for identifying the resource in question. An example of an URI that is not a URL is given on page 411.

URLs of the kind `protocol://host/localinfo`, like `http://www.javasoft.com/glossary.html` are concrete implementations of that general concept.

Application-Protocols (HTTP, FTP, . . .), Types and Network Applications



Application-Protocols (HTTP, FTP, . . .), Types and Network Applications — The previous paragraphs described some aspects of transferring raw-data over a network. Thereby topics, like addressing and virtual networks, have been selected that may be of special interest for application-programmers and end-users. As can be seen in the graphic below, the full complexity of a network-transfer-process (the curved bi-arrow) can be abstracted to a process that has the complexity comparable to that of a mere telephone-call (the straight bi-arrow, a logical connection):



http, the protocol of the World Wide Web
IP, the protocol of the Internet

This section indicates how application-related data is transformed and to be sent or received. On this level, only notions from the application- and presentation layers of the OSI layer-model become relevant. Now, on this level, special concern has to be given the type of that data: Different host-platforms may have different ways of coding data (due to different operating-systems or different application-software). So, before sending the data over the network it should be transformed into a platform-independent format as indicated in the graphic on page 351.

Transparent: 1. Adjective for actions that are easy for the end-user or application-programmer but require (usually) complex system-manipulations. Examples: Accessing the files on disk-drives through the file-system of the operating-system, instead of using elaborate port- or socket-mechanisms. – Hiding the technicalities of network-connections under protocol- or even application-handlers.

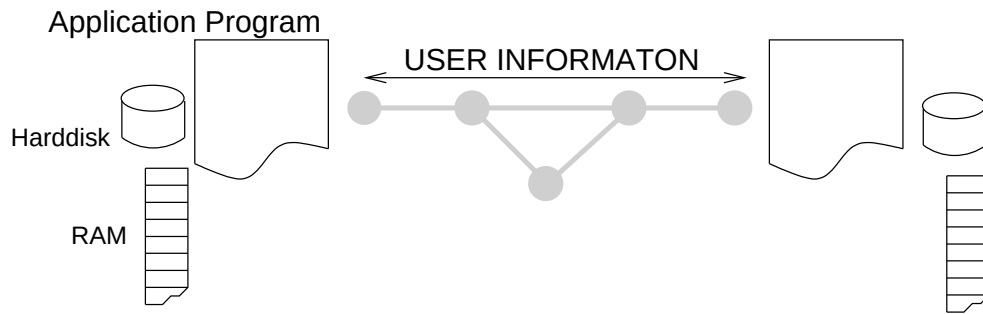
Transparency stresses the simplifying aspect of the abstraction. See also the notion of encapsulation (page 195) which focuses on complexities to be hidden by an abstraction, that should simplify matters. In short: The notion “transparency” reflects the simplicity that arises when lower level machinations have been made unseen to a programmer or end-user.

2. Non-opaque pixels or areas in Graphical User Interfaces or images (page 527).

3. Clearly understandable, fully recognizable in all its complexities.

This is an example of, how even in a language used for purely technical areas, one word may be interpreted as having plainly contradictory meanings.

Using application-protocols should make the network as “transparent” as using the telephone for making a telephone-call: After a connection has been established, the application-specific data should be transferable as easily as a local copy-process, mainly with the help of suitably formulated URLs.



Protocols tell the platform how to consistently communicate with another platform over a given network. Communication means exchanging information. Thus, to guarantee an unhindered exchange of information, the format or type of the encoded information has to be given in a platform independent way:

Universally Recognizable Types and Formats

Universally Recognizable Types and Formats — MIME-types (Multipurpose Internet Mail Extension) are types or rather formats for text-, image-, audio- and video-data supported by virtually every computer-platform which is connected to the Internet. These platform-independent **content-types** describe different kinds of data-files which contain real-life information like text, sound, images.

content-type
and encoding

information		(the formatted document)				
content-type	text/pdf	(how to interpret text tokens)				
encoding	ASCII	%	p	d	f	...
bytes	01101010 11110000 11111011 01010110 ...					
bits	01101010111100001111101101010110 ...					

Those different specifications of application-related formats use common **data-types** and **encodings**: Remember the ASCII-format that assigned bytes to characters as presented on page 33. Or consider page 265 for the names of some other encodings. Even most compression-formats (page 522) fall under the category “encodings”. Contrary to the more bit- and byte-related form of encoding; the application-related form of coding information, also called content-type, has been exemplified in the table below:

MIME-type	MIME-type/subtype	meaning
text		Readable and printable text.
	text/plain	Byte-sequences or files are decoded using the one-byte ASCII-format.
	text/html	Same as above, but interspersed with commands of the Hypertext Markup Language, which can be used for formatting purposes. An example of such a unformatted file is shown on page 370.
message		
	message/rfc822	An email-message with sender- and receiver-information, in the one-byte ASCII-format.
	message/news	The same as an email-message only to be displayed publicly in the receiving news-group.
application		
	application/pdf	Content-type for print-ready texts from Adobe Inc.
	application/java	Files of Java byte-code are recognized that way. Furthermore, the first 4 bytes in Hexcode (CAFEBABE) characterize Java byte-code.
image		
	image/gif	Image-type useful for images with plain colors and defined regular shapes. (Graphics Interchange Format)
	image/jpeg	Image-type useful for images with many colors and many irregular undefined shapes. (Joint Photographic Experts Group)
video		
	video/mpeg	Video format issued by the Motion Picture Experts Group.
	video/quicktime	Video format issued by Apple Inc.

This gives only a small selection; an up-to-date list can be found under <ftp://ftp.isi.edu/in-notes/iana/assignments/media-types/media-types>.

The platform-independent type-conventions introduced above are implemented on a computer-platform as programs for making the information perceivable for the end-user (associable with the presentation layer of the OSI-layer model). The software for accomplishing this is called content-handler:

content-
handler

Content-Handler: A program that translates data from a format (MIME-type) which is independent of an operating-system into another often application-specific format, for further processing on the native platform. Content-handlers include so-called plug-ins, which make texts, images or multimedia-files visible or even editable on-screen.

Content-handlers concern themselves with the perception of the end-user, whereas protocol-handlers (page 349) concern themselves with more or less abstracted control of network-traffic.

OUTLOOK: Content-handlers in this text occur as rather rudimentary constructs (cutting out command- and control-tokens, in the program on page 394) or appear implicitly in the program on page 390.

HTTP (Hypertext Transfer Protocol) the Protocol of the WWW

HTTP (Hypertext Transfer Protocol) the Protocol of the WWW — The Hypertext Transfer Protocol (HTTP) underlies the World Wide Web (WWW) services of the Internet. Due to its importance, some more information about this protocol is presented here. The HTTPProtocol is meant to be **used in client-server-communication**, HTTP doesn't work explicitly as a peer-to-peer protocol.

The HTTPProtocol is a **stateless protocol**, that means it furnishes no means of memorization on both sides. A transaction between client and server using solely this protocol does not influence future transactions in this protocol. (This restriction can be circumvented by making the server store so-called Cookies or IP-addresses on the client's persistent storage-device.)

The protocol transfers data coded by Multipurpose Internet Mail Extension (MIME-) types.

A (client- and server-side) implementation of the Hypertext Transfer Protocol (HTTP) has to recognize the application-specific **MESSAGES IN ASCII-FORMAT**. MESSAGES SENT BY THE CLIENT AND THE SERVER FALL INTO THREE PARTS, **HEADER**, **COMMANDORSTATUS** AND **BODY**. The end of any command or status of the message is marked by an empty line ("`\r\n\r\n`" as Java-string).

(CommandOrStatus) The client may contact the server in about the following form¹⁵¹:

```
aHTTPClientSideRequest := HTTPcommand URL HTTP version (1.0 or 1.1)
```

Example:

```
GET http://java.sun.com/index.html HTTP/1.0
```

The “GET”-command asks the server for a hypertext-file, a “PUT”-command tries to store the body-part at the location indicated by the Uniform Resource Locator (URL) on the server. “DELETE” tries to delete the URL-identified file on the server. “POST” tries to invoke a server-side program which processes data sent in the body-part to the server (CGI Common Gateway Interface programs). There are still more commands to be referenced: “HEAD” retrieves, not the files, but only information about the URL-specified file on the server, this is useful for checking whether the client-cached files are still up-to-date.

(Header) An optional header-part informs the server about the client’s platform and software, like the browser-type used (keyword = “User-Agent”) or supported types (keyword = “Accept”) using the format¹⁵²

```
aHTTPHeader := keyword: value {, value }
```

Example:

```
User Agent: ....
```

```
Accept: image/gif, image/jpeg, text/html, text/pdf, video/mpeg
```

In a message sent from the server, the header-part contains complementary (and complimentary) information about the server.

(Body) An optional body-part, in the case of “POST”- or “PUT”-commands, contains the associated data.

(Server-side-specifics) When receiving the “GET”-command above, the server answers with a status-line as a substitute for a command:

```
HTTPStatusLine := HTTPVersion StatusCode
```

Example:

```
HTTP/1.1 200 OK
```

(A list of status codes can be found in <http://www.w3.org/Protocols/HTTP>) The information above is sent together with an optional header-part in which the server sends information about its own configuration and an optional body-part that contains the application-data:

```
Date: Wed, 26 Feb 1991
```

```
Server: FirstOne
```

```
Content-type: text/html
```

```
Content-length: numberOfBytesOfTheBodyPart
```

An optional body-part which contains data in ASCII-format may look like this:

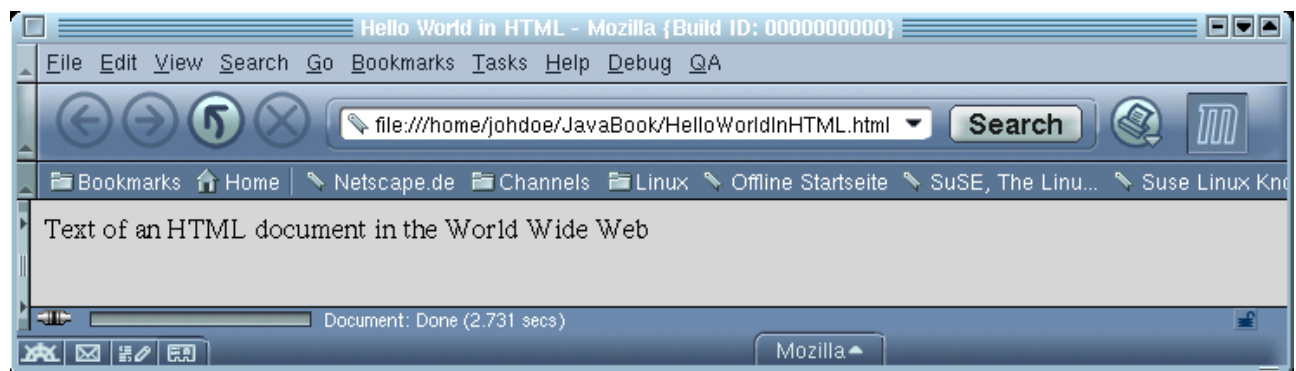
```
<html>
```

```
<head><Title>Hello World in HTML</Title></head>
```

```
<body>Text of an HTML document in the World Wide Web</body>
```

```
</html>
```

Then the body, a text in HTML-format, is presented in a browser as follows:



¹⁵¹This is the trigger used in a programming-environment as demonstrated in the client-source-code on page 387. Whereas an end-user enters an Uniform Resource Locator (URL) into the browser command-line, as discussed in the next section.

¹⁵²The brace “{...}” indicates zero or multiple occurrences of the content.

Some network-specific applications that use application-protocols like HTTP, FTP, SMTP are listed below:

Hypertext

Hypertext — Consider a text-file with (mouse-click-) sensitive passages or images that point to another text(-passage), where their activation (with a mouse-click) loads the other text. Such texts are called **hypertexts** and their (mouse-click-) sensitive passages are called **hyperlinks**. Formats for hypertexts are HTML (Hypertext Markup Language) and PDF (Portable Document Format). The application-protocol that manages their exchange and their linking over TCP/IP-networks, especially in the Internet, is given by the Hypertext Transfer Protocol (HTTP), which works on TCP-port 80. The part of the Internet-servers that provides hypertexts is called the World Wide Web. Usually hypertexts are stored on servers to be accessible by client-hosts. The application- or presentation-software for reading hypertexts and activating their links is also called **browser**. Some product-names are Mozilla, Netscape Communicator or the browser Internet Explorer, Opera or kde-frames in the Graphical User Interface KDE over Linux.

Copying Files over the Network

Copying Files over the Network — A less refined concept, but nevertheless useful due to its independence of the file-format, represents a simple file-transfer function in networks. This is provided by the File Transfer Protocol (FTP) a reliable client-server-oriented protocol that just furnishes a network-wide copying service. Copying files across a network is also denoted as “**downloading**” (client from server) or downloading “**uploading**” (client to server). The FTP-protocol and its implementing applications work on TCP-port 21 and uploading (for signaling) and TCP-port 20 (for data-transfer). The application- or presentation-software is just called “FTP-client” or “FTP-server”.

Private Mail

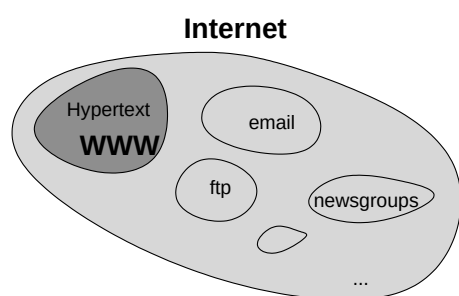
Private Mail — An email can be seen as the equivalent to a postcard or letter on a computer-network. If the receiver-host is only temporarily part of the network, then the sending of such a file requires a sort of post-office-box on a server. When the receiver becomes again part of the network (“is online”), then the mail can be downloaded from the server. The mechanism already has been described on page 277 in the context of protocol-conceptualization.

The Simple Mail Transfer Protocol (SMTP) specifies how files can be sent to another computer in the network using an URL addressing-scheme like `receiver@domain.name.com`. The protocol works on TCP-port 25.

The Post Office Protocol (POP) describes how such a message can be downloaded from the post-office-box computer. The protocol works on TCP-port 110. Application- and presentation-software is just called “**email-client**”.

Initially, email-files were simple text-files in ASCII-format. Later, Multipurpose Internet Mail Extension (MIME-) types have been added, that serve as network-wide normed types. Now emails often are sent in HTML-format.

Other Applications



ment can be found on page 363.

Other Applications — There are still other services: For example, a bulletin-board-system which can receive regular emails which are posted publicly. This uses the Network News Transfer Protocol (NNTP). Or the remote control of computers, that means for example, being able to use a terminal-window of some (Unix- or Windows-) computer on another computer of the network. The TELNET protocol furnishes such a service on TCP-port 23. Other network-related applications may include collaborative working environments (chat-rooms) and the wide field of distributed computing. Listings with TCP-port assign-

World Wide Web (WWW, W3): 1. The notion World Wide Web encompasses the services furnished by the Hypertext Transfer Protocol (HTTP). This includes the serving of text in the HTML- or PDF-format, the (hyper-) linked resources like other texts, images (mostly in the GIF- or JPEG-format) and also exchanging data entered by some user into online-forms.

2. The part of the Internet that communicates using the Hypertext Transfer Protocol (HTTP). Addresses, or rather Uniform Resource Locators (URLs) of this part have the form `http:// ...` or even `http://www. ...`, like `http://www.howstuffworks.com`.

Thus the World Wide Web can be seen as the collection of application-related services which base themselves on the HTTP-Protocol, that means serving hypertexts of the HTML- and PDF-type. Related to this context, the notion “Internet” describes all the application-related services that can be based on the Internet-Protocol (IP): These services include the World Wide Web, but also include email-services, news-groups and more.

Internet: Connecting technologically different networks by bridges, then using software, that implements the Internet Protocol (IP), to cover the technological difference, so that upper-leveled protocols (transport- and application-protocols) and the end-users see only one single (virtual) network represents the gist of a so-called IP-network.

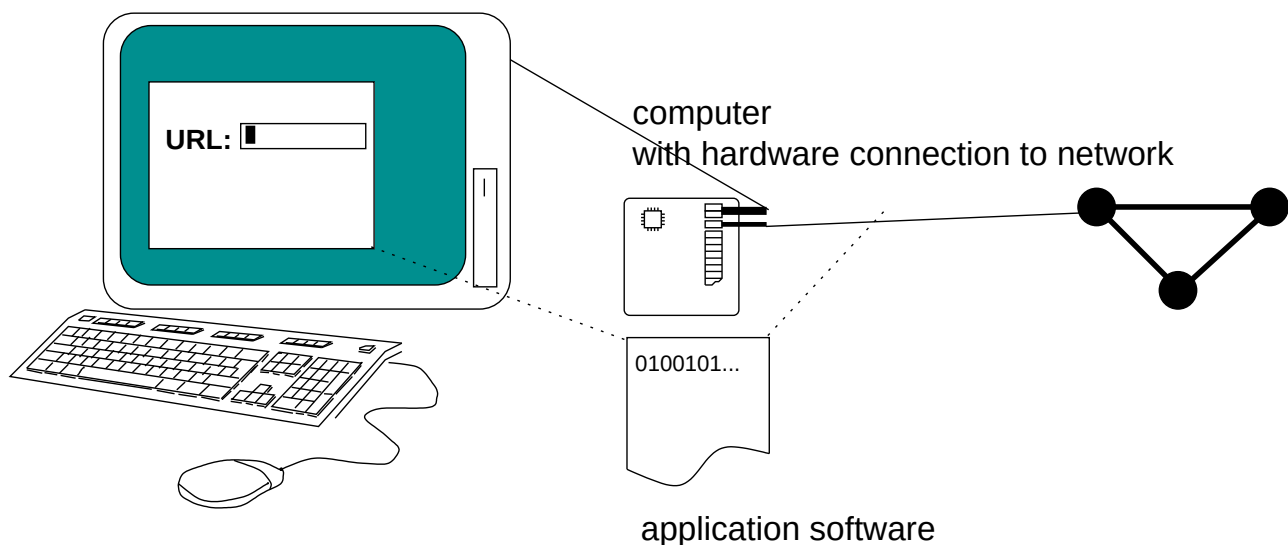
1. The name “Internet” is used for the **IP-network** that (ideally) is globally accessible by an average computer-user. Isolated IP-networks are also called **Intranets**.

2. “Internet” also denotes the different application-related **services** offered by hosts of this network: Copying files, that includes downloading (client from server) and uploading (client to server); the File Transfer Protocol (FTP) usually is the first choice for these activities. Sending and receiving emails (SMTP, POP, ...). Sending newsgroup-messages to be publicized and reading the latest public newsgroup-messages (NNTP). Logging into another computer from a remote terminal using the TELNET-protocol and among other services the most popular one: The World Wide Web that makes it possible to access information and knowledge in the form of hypertext-documents (in HTML or PDF content-types).

Triggering: URL- Network-wide Invoking of Application-Services

Triggering: URL- Network-wide Invoking of Application-Services — Remember the general introduction of the idea of an Uniform Resource Locator (URL) on page 364, in the context of the TCProtocol. The concept of an URL even as introduced there, can be seen as a general **trigger for accessing network-services**. — Invocation application-services, basically needs a computer connected to a network. Then, in a simple case, an application-related network-software, like an Internet-browser, furnishes a command-line into which a Uniform Resource Locator-string (URL) can be entered:

URL:
application-
service
addressing



Uniform Resource Locator (URL) — An Introduction

Uniform Resource Locator (URL) — An Introduction — On programming-level and user-level, how can be invoked application-related network-services? How can network-connections be established and used with application-protocols, like “http” and “ftp”? On the application-level, the end-user enters an Uniform Resource Locator-string into some command-line or terminal-window and/or an application-program dispatches an URL-string into the associated protocol-software. The different software-implementations of the (lower level) transport- and network-protocols, usually the TCP/IP protocol-stack, automatically handle the rest of the network-communication.

What functional aspects are connected to an URL-string?

- A Uniform Resource Locator (URL) represents a **NETWORK-WIDE ADDRESS OF RESOURCES**:
 - single files (HTTP, FTP) or even file-section in files (application-protocol: HTTP for Hypertext Transfer Protocol)
Example: **http://www.sun.com**
(Sections inside the files are identified following a “#” pound sign:)
Example: **file://localhost/text1.html#marker1**
 - remotely existent objects (application-protocol: RMI for Remote Method Invocation)
Example: **rmi://192.168.0.1/theObject**
 - a person’s email post-office-box for sending an email (application-protocols: SMTP for Simple Mail Transfer Protocol and POP for Post Office Protocol)
Example: **mailto:info@sun.com**

- a news group's post-office-box for sending messages (enews) to a bulletin-board of a special interest group (application-protocol: NNTP for Network News Transfer Protocol).
Example: **news:comp.lang.java**
- and many more addressing-schemes of application-related services.
- A Uniform Resource Locator (URL) TRIGGERS THE **EXCHANGE OF DATA**:
 - loading of a HTML-page (*fileName.html* or *fileName.htm*) with GIF- or JPEG-images (*fileName.gif* or *fileName.jpg* respectively) (application-protocol: HTTP)
Example: **http://www.cnn.com/index.html**
 - uploading or downloading any kind of file (application-protocol: FTP)
Example: **ftp://ftp.isi.edu/in-notes/iana/assignments/mediatypes/media-types**
 - sending an email (application-protocol: SMTP)
Example: **mailto: info@sun.com**
 - sending a news-letter to a news-group (application-protocol: NNTP).
Example: **news:comp.lang.java**
 - invoking a method of an object on a remote computer (application-protocol: RMI)
Example: **rmi://192.168.0.1/theObject**
For more see the source-code beginning on page 417, as presented in the part of the RMI-client.
- A Uniform Resource Locator (URL) is a SEQUENCE OF CHARACTERS (often coded as ASCII-byte sequence) that follows an URL-format and the URL-encoding-scheme, as described in the following section, titled "URL Encoding".

The examples already showed that each Uniform Resource Locator begins with the denotation of an application-protocol or a related keyword, followed by a colon (:). The names of services usually follow the colon directly as could be seen above in the cases of email- and newsgroup-services. File-like resources add a double-slash to the colon producing a token `://`, then follows the host-name and the absolute Unix-like slash-separated path-name of the targeted resource, for example a text-file or a Java object. Except for default-pages, like `index.html` pages, generally the resource is specified according to the following scheme:

In a context that can take URL-inputs, like an Internet-browser, even local resources can be accessed just by omitting the host-name or by using a "localhost"-keyword. A file in the root directory can be addressed as follows:

<code>applicationProtocol:// /fileName</code>	<code>http:// /aHypertext.html</code>

If the operating-system of the host-platform implements a Hierarchical File System (HFS), which is the case with most operating-systems (BeOS/Linux/Unix, Mac OS, Windows), then a SLASH SEPARATED PATH-NAME can be used to identify the file independently of the operating-system:

<code>applicationProtocol:// absoluteHFSpath/separated/fileName#anchor</code>	<code>http:// /root/index.html</code>

To address a file on the network, the IP-address of the host or an associated domain-name has to be included as indicated below. If necessary, separated by a colon, even the port-number may be included, this tells the foreign computer on what TCP-port the service of the application-protocol is to be expected:

<code>applicationProtocol://IP-addressOrDomain-name:port/fileName</code>	<code>http://www.gamelan.com:80/index.html</code>

why "www" in domain-names?

relative Uniform Resource Locator (URL)

The token "www" in a domain-name is meant to hint at the HTTPProtocol and thus make a standard URL (`http://www.sun.com/index.html`) reproducible just from giving the domain-name (for example `www.sun.com`), this method of giving a URL, shortened down to the domain-name, can be seen now in almost any contemporary publication.

A path-name (see page 47) in URL-format is also called "relative URL" exactly if this path-name supplements a given string to a complete (absolute) URL. (analogously to relative- and absolute path-names as introduced facing the table on page 48)

A more extensive specification, that includes various schemes of URL's, is given in RFC 1738 (see page 659). The concept of a network-resource and its identifier is generalized in RFC 2023.

OUTLOOK: The Java-specific aspects of the URL are considered on page 388.

URL-Encoding

In some operating-systems like Linux/Unix the use of spaces in file-names is discouraged, other operating-systems like the Mac OS below Version X feel unhappy with colons (:) which appear in the file-names. The URL character-sequences are specially encoded, to make Uniform Resource Locators as independent as possible of concrete naming conventions within different operating-systems: Lower- or uppercase letters, numerals and underscores “_” remain ASCII-coded. Any other ASCII-character is coded by three ASCII-symbols: The percent-character “%” followed by a hexadecimally coded number that gives the ASCII-value of the character. Reserved signs are “+, /, #, =, &, ?”. Spaces are thus coded as “%20” but can be substituted by the reserved “+” sign, the plus sign itself is coded by “%2B”. Usually this coding is generated automatically by the software used, so users or application-programmers need not be overly concerned. (Java furnishes a pre-written class as part of the package `java.net`, the so-called `URLEncoder`-class; this class automates the process of URL-encoding a given string.)

Summary: Internet- and Transport-Protocols, Addresses, Domain-Names

Summary: Internet- and Transport-Protocols, Addresses, Domain-Names — The previous section essentially introduced four conceptual levels that can be identified when thinking about computer-networks (an extension to seven layers has been hinted at): The real networks, the virtual network, the transport-protocol and the application protocols and -software. Those levels have been presented in more detail, concerning network-protocols, especially the Internet Protocol (IP); the transport-protocols, especially the TCP- and UDP-protocols and application-protocols like HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol), SMTP (Simple Mail Transfer Protocol). The HTTP-protocol forms the basis of the World Wide Web service of linked hypertexts.

The role of hardware, software, protocols and MIME-types taking part in transferring data over the Internet has been sketched in the graphic on page 367.

End-users and application-programmers ideally approach only the uppermost, application-related, level of the entire network-communication-process. System-programmers and network-technicians usually have grasped the basic hard- and software levels in such a way that using this infrastructure requires only the knowledge of a (hopefully) relatively simple Application Programming Interface (API)¹⁵³. In the networking context this form of black-box design is often called **transparency of the network**. (generally the description of complex entities by a simpler model is called abstraction; in the area of object-orientation this is also called information-hiding or encapsulation, with all three notions essentially circumscribing the same image-of-thought) To stress this abstraction, an established network-connection may be given the adjective “logical”, making it a so-called logical connection.

End-users of the telephone system also simply dial a number and usually reach their communication-partner. This kind of connection might be called logical because the end-user isn't aware of the technical complexities that make the phone-line system work. The use of a car or an elevator require only a few rules to be memorized, here the complexity is also hidden to ensure an easier and safer use. Remember also the modeling chapter at the beginning of the text around page 28.

The Internet is an IP-based virtual global network from the network-technological point of view. For an end-user the Internet is the global network that has DNS-domain-names as addresses and furnishes services like email, FTP-servers, and HTTP-connections of hyper-linked HTML-documents. The part of the Internet with HTTP-servers and services is also called the **World Wide Web**. The World Wide Web essentially is the name of the hypertext-part of the Internet. The domain-names of HTTP-servers *often* identify themselves by the prefix “www.”, like in “www.sun.com”.

Intranets are IP-based networks that are either isolated from the Internet or only accessible from the Internet with a special access-permission. Many institutions use Intranets as an internal information-system. **Virtual Private Networks (VPNs)** are virtual networks deployed across the Internet and encrypt their data for privacy, they are typical logical sub-networks.

An application-programmer or end-user can visualize the Internet (or IP-based networks) as a network that routes packets of data from one computer in the network to another computer in the network. This packet-routing and addressing is administrated network-wide by the IP-software (Internet Protocol (IP)): Each computer in an IP-network is assigned at least one number, the so-called IP-address (Internet Protocol address). An IP-address is a sequence of four numbers, each number taken out of the range from 0 to 255 inclusive¹⁵⁴.

A typical IP-address might look like 23.233.12.1 

IP-address

The Internet Protocol (IP) takes care of the routing of the IP-packets along the nodes of the network. A secondary protocol (DNS acronym for Domain Name Service) automatically maps

domain-names like “java.sun.com” 

domain-names

¹⁵³The notion of an Application Programming Interface (API) also denotes the documentation-part of code-libraries.

¹⁵⁴These addresses are 32-bit (=4 byte) addresses and can at best address about four billion computers ($2^{32} = 4,294,967,296$). A new Internet Protocol (IPv6), that is backward-compatible with the present one, uses 128-bit addresses (=16 byte addresses) and thus can address more computers that are practicably imaginable right now ($2^{128} \approx 2.05 \cdot 10^{38} \approx 3.4 \cdot 10^{38}$).

to assigned IP-addresses. The only things application-programmers and end-users see of these protocols are the IP-addresses and the domain-names!

The network-protocol called the Internet Protocol (IP) only routes packages according to their destination-address. Transport-protocols like TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are pieces of software on the host-computers that organize the reception and the dispatch of packets, the Internet Protocol channels them through the network. — In TCP/IP¹⁵⁵ networking each IP-address has been assigned software channels numbered 1 through 65535. Those channels are called TCP-ports¹⁵⁶:

TCP-Socket = IP-address + TCP/UDP-port-number

Examples:

“12.23.34.45:80” or “java.sun.com:80”

The software that enables an application-programmer to contact another computer in the network, by using a TCP/IP protocol-stack, is also called socket. A TCP/IP-socket can be used to establish TCP/UDP connections. Such a connection, then can be used to send and receive information for many different services (for example HTTP (HyperText Transfer Protocol) on TCP-port 80, SMTP (Simple Mail Transfer Protocol) on TCP-port 25). Since the TCP-protocol is connection-oriented, once a contact is established, it has to be closed down explicitly. Only if data is sent or received, then some network-traffic is generated.

The packet-oriented UDP-protocol just establishes the endpoints of communication using the IP-routing mechanism. The UDP-protocol only exchanges single, in its view unrelated, IP-packets, also called datagrams.

Application- or service-protocols use transport-protocols and network-protocols to code their data (emails, pictures, serialized Java objects, Java method-calls with RMI) and send that data over the Internet to a destination-computer, where the data can be unpacked and used in the intended way (as emails, pictures, serialized Java objects, Java method-calls with RMI). Usually application-protocols (like HTTP, FTP, RMI, TELNET, SMTP, POP, LDAP) are given as pre-programmed classes ready for use. Seldom an application-programmer has to design, rewrite or implement an application-protocol.

In triggering remote services, a Uniform Resource Locator (URL), like

“http://java.sun.com:80/index.html” 

specifies the application-protocol “http”, the IP-address or domain-name “java.sun.com”, possibly the TCP-port “:80” and the resource as a file in a Hierarchical File System (here simply) “index.html” with a slash as separator and according to the other conventions of URL-encoding.

ORGANIZE: Your Network-Connection



ORGANIZE: Your Network-Connection — To establish a network-connection, the operating-system needs to furnish a working TCP/IP protocol-stack. The fact that you can access the Internet using that operating-system assures the existence of such TCP/IP-software. Most of the private computers may be connectable to the Internet by modem and telephone-line (a dialup-connection) or xDSL-connection.

Often this networking protocol-software runs on Ethernet-based **networking-hardware**. Then the computer’s manual indicates that there exists an Ethernet-card pre-installed, or the user has to select an appropriate Ethernet-card for PCI- or ISA-slots on the motherboard. In their simplest versions, such cards have RJ-45 connectors¹⁵⁷ with eight contacts for 10BASE-T Ethernet networks or coaxial BNC-connectors for 10BASE2 Ethernet networks. In the simpler cases the cables from each Ethernet card are joined by a little box called the hub. — At least try to check whether the platform (hardware and operating-system) is compatible with the specific Ethernet-card otherwise the networking may fail on hardware level. In other cases the network-connection may be established by plugging ISDN-connectors into ISDN-PCI-cards. There are many more possibilities of encountering network-connectivity within a given computer.

Describing the configuration of a working TCP/IP-environment is done independently of any operating-system, due to the relatively rapid changes of concrete procedures: After having the proper hardware, as described above in the case of an Ethernet network, configuring a TCP/IP-stack includes the following general steps: Assure that the **(driver) software for the (Ethernet-)network card** is installed or properly chosen from a pre-installed driver-pool. The (pre-)installed **TCP/IP software** has to be given the **IP-address of the computer** (usually 192.168.0.X X=0...255 in a local network) and the **subnet-mask** (usually 255.255.255.0 in a local network, see page 358). If a server for the resolution of domain-names (page 360) works in that network, then the **IP-address of this DNS-server** has to be furnished too. The

¹⁵⁵Usually the mentioning of the TCP/IP set of protocols means to include the UDP-protocol.

¹⁵⁶TCP-ports are not to be confused with the ports on the level of Unix operating-systems. Each Unix-port represents a unidirectional connection to another computer. See also page 231.

¹⁵⁷Telephone connectors with four contacts are called RJ-11.

server- or/and the client-software (page 386 or/and 387) has to be installed and started. The server has to be given the files to be served, often a set of linked HTML-files in some directory(-structure).

Then, the client can fetch files from a server using a **Uniform Resource Locator**, like `http://java.sun.com` in a network with a Domain Name Service (DNS) or ignoring a possibly non-existent Domain Name Service using the IP-address of the computer in the URL: `http://192.168.0.1` (page 364 and page 373).

In the following is assumed that 1. the computer is attached to a network, 2. the operating-system is installed and working, 3. that the native platform allows communication over the TCP/IP set of protocols — and, 4. of course, has installed a valid Java Runtime System (JRS).

Java: Programming in Networks

Java: Programming in Networks — First consider a collection of statements: The `java.util.Vector`-class works as an extensible array. - The component-model in object-orientation partially changes object-communication-by-method-calls to object-communication-by-message-exchange. - In multithreading context, synchronized blocks of source-code make the blocks of source-code atomic.

In the same way the following part on Java network-programming (on applicational level) presents a *selection* of basic functionalities that are highly extensible or can be used in ways not presented in this text. Especially in network-programming, many applications may take advantage of those supplements for easier programming. Some ideas of concepts for further refinement of network-related programming are described at the end of this chapter.

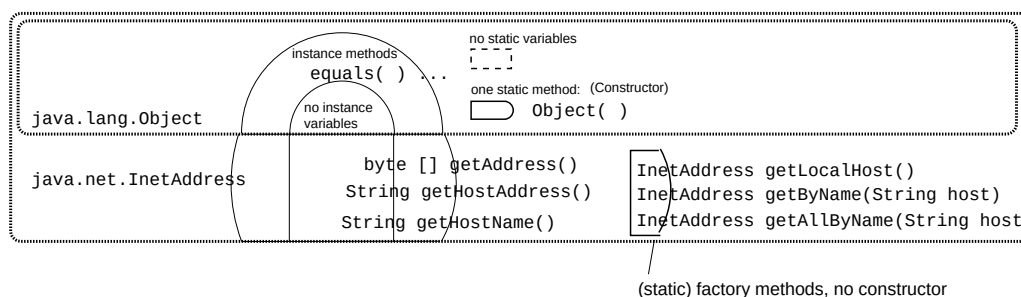
As could be seen, the Java language-design incorporates multithreading concepts. But the Java language-design does NOT incorporate networking-concepts. Java ONLY uses networking-code furnished by packages. For doing basic network-programming, the Java 2 Standard Edition (J2SE) provides readily usable class-libraries:

package java.net	IP/TCP/UDP-network Application Programming Interface (API)
package java.rmi	calling remote methods and exchanging serialized object-information (Java version of Remote Procedure Call (RPC))
package java.applet	classes for using Java's client-based computing infrastructure
package java.servlet	classes for using Java's server-based computing infrastructure (Java version of Common Gateway Interface (CGI))

For full specification of the IP/TCP/UDP-network Application Programming Interface (API) in Java see the  Java Platform API Specification [JPAPIS]. The following sections focus on presenting working prototypes of source-code which demonstrate the *core* functionalities of the Java networking-classes.

Java: Relating IP-Address, Host-Name and Owner-Information

Java: Relating IP-Address, Host-Name and Owner-Information — The package `java.net` furnishes one basic class that, unseen to the application-programmer uses the Domain Name Service (DNS) to match IP-addresses and DNS-host-names: The Java class `java.net.InetAddress` uses factory-methods to instantiate its objects. On a computer with a network-connection, objects of type `java.net.InetAddress` automatically use the network-services to offer the programmer data about relations of IP-addresses and DNS-host-names, as described on page 360. Even information about the computer that runs the current Java Runtime Environment (JRE) becomes accessible when using the (static) factory-method `java.net.InetAddress.getLocalHost()`. The graphic below gives an impression of the class `InetAddress`:



The final class `InetAddress` with its `private` – effectively non-existent – constructor forces the application-programmer into using the factory-methods¹⁵⁸ to instantiate an object. Unlike a constructor the factory-methods can be given more descriptive names.

```
class IPAddressAndDomainName
{ static String domainName = new String( "howstuffworks.com" );
  static String ipAddress = new String( "207.25.71.24" );
  public static void main( String unused[] )
  { java.net.InetAddress localhost, address, addresses[];
    try
    { System.out.println( "The local host is " + java.net.InetAddress.getLocalHost() );
      address = java.net.InetAddress.getByName( ipAddress );
      System.out.println( "IP no. " + ipAddress + " has domainName " + address );
      address = java.net.InetAddress.getByName( domainName );
      System.out.println( "host " + domainName +
                          " has IP no. " + address.getHostAddress() );
      addresses = java.net.InetAddress.getAllByName( "cnn.com" );
      for( int i = 0; i < addresses.length; i++ )
      { System.out.println( "host \"cnn.com\" has IP no. " + addresses[i] );
      } }
    catch( java.net.UnknownHostException e ){ System.out.println( e ); }
    catch( java.io.IOException e ){ e.printStackTrace(); }
  } }

/*
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol > javac IPAddressAndDomainName.java
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol > java IPAddressAndDomainName
The local host is linux.local/192.168.0.1
IP no. 207.25.71.24 has domainName www4.cnn.com/207.25.71.24
host howstuffworks.com has IP no. 216.183.103.152
host "cnn.com" has IP no. cnn.com/207.25.71.25
host "cnn.com" has IP no. cnn.com/207.25.71.26
...
host "cnn.com" has IP no. cnn.com/207.25.71.6
host "cnn.com" has IP no. cnn.com/207.25.71.20
host "cnn.com" has IP no. cnn.com/207.25.71.22
host "cnn.com" has IP no. cnn.com/207.25.71.23
host "cnn.com" has IP no. cnn.com/207.25.71.24
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol >
*/
```

So the `java.net.InetAddress`-class furnishes an interface of the DNS (Domain Name Service) protocol for Java application-programmers. This class encapsulates the access to the DNS protocol. This class makes the use of the DNS protocol transparent (unnoticeable) for the application-programmer. The class `InetAddress` uses the UDP (User Datagram Protocol) that sends single packets over the IP-net. The `java.net`-package also makes the `UDProtocol` explicitly available to the Java application-programmer, for that service made explicit see the next section.

But first remember the list of the TCP-port assignments on page 363; there has been mentioned a WHOIS service of the Internet, an application-protocol that informs about who holds a particular IP-address or a particular domain-name. Some of the knowledge about this protocol has been used in the following program to get the desired information about an Internet host-computer:

Here a `java.net.Socket`, a TCP-socket has to be established explicitly; then there are instantiated two stream-objects on this socket, which channel the output question in the WHOIS-format and process the answer so that it appears on-screen:

```
public class WhoIs
{ final static int WHO_IS_PORT = 43;
  final static String WHO_IS_HOST = "rs.internic.net";
  /* The European region is serviced by whois.ripe.net,
     more of this can be found in www.internic.net.
  */

  static java.net.Socket whoIsTCPSocket;
  static java.io.DataInputStream whoIsInputStream; // <<<<<
  static java.io.DataOutputStream whoIsOutputStream; // >>>>>

  public static void main( String [] potentialDomainName )
  { if( potentialDomainName.length != 1 )
    { System.out.println("enter domainname"); System.exit(-1);}
    try
    { whoIsTCPSocket = new java.net.Socket( WHO_IS_HOST, WHO_IS_PORT );
      whoIsOutputStream =
        new java.io.DataOutputStream( whoIsTCPSocket.getOutputStream() ); // >>>>>
      whoIsInputStream =
```

¹⁵⁸ Factory-methods inside the class of which they instantiate objects, have to be chosen static! Especially if there exists no accessible constructor, at least one factory-method has to be static if the class is to be instantiated. A reminder on factory-methods can be found on page 151.

```

        new java.io.DataInputStream( whoIsTCPSocket.getInputStream() );    // <<<<<
        whoIsOutputStream.writeBytes( potentialDomainName[0] + "\r\n" );
        // the above line sends the question to the WhoIs-server.
        try
        { do
            { byte s = whoIsInputStream.readByte(); // receives answer from WhoIs-server
              System.out.print( (char)s );
            }while( true );
        }catch( java.io.EOFException e )          // an exception ends the WhoIs reply
        { System.out.println(">>>> End of File <<<<<"); }
        whoIsTCPSocket.close();
    }
    catch( java.io.IOException e ){ e.printStackTrace(); }
} }
/* www.dns.net informs on how domain-names and IP-addresses are resolved technically
   by name-servers like DNSAUTH1.SYS.GTEI.NET
*/

/*
(Whois informs only about registered top-level-domains, not subdomains,
 to get a result try below 'berkeley.edu':)
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol > java WhoIs setiathome.ssl.berkeley.edu

Whois Server Version 1.3

Domain names in the .com, .net, and .org domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

No match for "SETIATHOME.SSL.BERKELEY.EDU".

>>> Last update of whois database: Tue, 31 Jul 2001 02:02:11 EDT <<<

The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains and
Registrars.

>>>> End of File <<<<<
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol >
*/

/*
C:\WINDOWS\JavaCodeFiles\CommunicatingComputers\InternetProtocol>java WhoIs berkeley.edu

Whois Server Version 1.3

Domain names in the .com, .net, and .org domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

    Domain Name: BERKELEY.EDU
    Registrar: NETWORK SOLUTIONS, INC.
    Whois Server: whois.networksolutions.com
    Referral URL: http://www.networksolutions.com
    Name Server: NS2.BERKELEY.EDU
    Name Server: NS2.ZOCALO.NET
    Name Server: PHLOEM.UOREGON.EDU
    Name Server: NS1.BERKELEY.EDU
    Name Server: VANGOGH.CS.BERKELEY.EDU
    Name Server: UCSFNS1.UCSF.EDU
    Name Server: DNS2.UCLA.EDU
    Updated Date: 13-jun-2001

>>> Last update of whois database: Mon, 13 Aug 2001 02:05:19 EDT <<<

The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains and
Registrars.

>>>> End of File <<<<<

C:\WINDOWS\JavaCodeFiles\CommunicatingComputers\InternetProtocol>
*/
/* organizational matters:
"Who assigns IP-addresses?" http://www.iana.org, see page 359.
"Who assigns domain-names?" www.icann.org accredits domain-name registrars
                           like www.networksolutions.com
"Who informs about IP-addresses?" see previous program
"Who informs about domain-name assignments?" www.internic.net and www.webmagic.com
                           and any WHOIS-server
*/

```

In the program above, the Domain-Name of the service-providing host is stored as a String-constant WHO_IS_HOST. Since the source-code above initiates the communication with the service-providing host, this piece of code represents a typical client. Java's TCP-socket, given by AN OBJECT OF CLASS-TYPE `java.net.Socket` SUPPLIES JUST THE MEANS OF ACCESSING THE NETWORK¹⁵⁹, these Java objects take

¹⁵⁹Java's TCP-sockets access the IP-network. Sockets control the network-access, streams carry the data. A weak analogy to Java's implementation of multithreading: The Thread-object controls a thread, a `run()`-method contains the code of the thread.

neither an intrinsic client-role nor an intrinsic server-role! For a general description of the notion socket, remember the gray frame on page 273.

Java: UDP Sockets and Datagrams

Java: UDP Sockets and Datagrams — Packet-routing networks¹⁶⁰, like an IP-network (for example the Internet), may be used directly without covering them with a connection-oriented protocol (for example TCP, the Transmission Control Protocol). This enables a programmer to send and to receive individual packets without encapsulating the network under any elaborate protocol-layer (like TCP). A simple protocol on top of a packet-routing IP-network is given by the connectionless (page 350) UDPProtocol (User Datagram Protocol) as mentioned on page 363.

Beside TCP-sockets, derived from the class `java.net.Socket`, which have been introduced in the previous program, and Java server-sockets, derived from the class `java.net.ServerSocket`, which will be introduced on page 385, the package `java.net` also furnishes a UDP-socket facility: An instance of class-type `java.net.DatagramSocket` offers and takes single message-objects of the class-type `java.net.DatagramPacket`. The program below is the only place in this text that actually shows an individual packet of data to be sent over a network or to be received from a network!

```

/* The class UDPSocket has a main()-method.
That method instantiates an UDP-socket-, a sender- and an receiver-object.
The sender sends a single packet out into the network.
The receiver lets a while-loop listen for arriving packets.
This program requires a working counterpart on some other host in the (local) network.
*/
class UDPSocket extends java.net.DatagramSocket
{
    UDPSocket( char udpPort ) throws java.net.SocketException
    { super( udpPort );
      System.out.println("socket opened");
    }
    // No IP-address because UDP is a connectionless protocol, IP-address only in datagram!

    public static void main( String IPAddressOfDestination[] )
    { char udpPort = 1025;
      try
      { UDPSocket udpSocket = new UDPSocket( udpPort );
        new UDPSender(                               // SENDER established
          ( udpSocket, java.net.InetAddress.getByAddress( IPAddressOfDestination[0] ), udpPort );
        new UDPReceiver( udpSocket );// RECEIVER established
      }catch( Exception e ){ e.printStackTrace(); } // ALL PURPOSE CATCH
    } }

    class UDPSender
    { UDPSender( UDPSocket udpSocket, java.net.InetAddress drainIPAddress, char drainUdpPort )
      { String message = new String( "A string message over udp" );
        byte[] byteArray = new byte[ message.length() ];
        byteArray = message.getBytes();
        // here only low order byte of each (two byte) character in String
        // otherwise use the String-method 'getChars()' and a double length array
        java.net.DatagramPacket udpPacket = // THE PACKET
        new java.net.DatagramPacket( byteArray,byteArray.length,drainIPAddress,drainUdpPort );
        try
        { udpSocket.send( udpPacket ); //// the trigger for sending the UDP-packet ----->
          System.out.println( "A packet has been sent." );
        }catch( Exception e ){ e.printStackTrace(); } // ALL PURPOSE CATCH
      } }

    class UDPReceiver
    { UDPReceiver( UDPSocket udpSocket )
      { byte [] buffer = new byte [65535];
        // actually only needed: 2^16 minus IP-Header minus UDPHeader
        java.net.DatagramPacket udpPacket =
          new java.net.DatagramPacket( buffer, buffer.length );
        while(true)
        { try
          { udpSocket.receive( udpPacket );// blocks, stops until new message is received <-
            System.out.println( new String( udpPacket.getData() ) +
              " from " + udpPacket.getAddress() +
              " at port no. " + udpPacket.getPort()

```

¹⁶⁰As introduced on page 345.

```

    });
} catch( Exception e ){ e.printStackTrace(); }
} } }
/* No client, no server because contact is not initiated by anyone,
   therefore sender and receiver denotations have been used.
*/
/* The two computers have to be connected by networking hardware. In both operating-systems,
   the TCP/IP control windows have to be properly configured,
   giving one computer the local IP-address
   Terminal input of computer1 (192.168.0.1):
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] vincent% javac UDPsocket.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] vincent% java UDPsocket 192.168.0.2
socket opened
A packet has been sent. (packet is lost if computer2 has not been started)
A string message over udp from 192.168.0.2/192.168.0.2 at port no. 1025 (received AFTER starting computer2!)

   Terminal input of computer2 (192.168.0.2):
jerome@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol > javac UDPsocket.java
jerome@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol > java UDPsocket 192.168.0.1
socket opened
A packet has been sent.
*/
// Restarting the program on each computer resends the message.
// Sender- and receiver-parts can be threaded to send and receive independently.

```

Exactly this method of communication is used by the Domain Name Service (DNS, page 360) and by Java's Remote Method Invocation (RMI), which will be introduced beginning with page 414.

This UDP-demonstration program uses a clearly symmetric form of contacting its copy on a partner-computer on some IP-network, for example the Internet. This exemplifies a typical peer-to-peer setting (page 352) of communication. All the following examples will take a client-server-form.

Java: TCP-Sockets and Network-Connections

Java: TCP-Sockets and Network-Connections — So far have been introduced elementary classes like `InetAddress`, `Socket`, `DatagramSocket`, `DatagramPacket`. Except for the `DatagramSocket` — an UDP-peer-socket —, each sub-class of the super-class `java.net.Socket` has been used as a TCP-CLIENT-socket (for example see the program `WhoIs`, on page 382). The following two pieces of source-code demonstrate the interplay of Java's TCP-socket with Java's TCP-server-socket, extending the class `java.net.ServerSocket`. Note that the following two programs yield no visible service, they only show the Java-specific overhead which is necessary to establish a client-server-connection on the top of the TCPProtocol (a more practical example follows instantly):

```

class ProtoClient
{ static char serverTCPPortNum = (char)1025;

  public static void main( String unused[] ) throws java.io.IOException
  { java.net.InetAddress serverIPAddress
    = java.net.InetAddress.getByName( "localhost" );
    java.net.Socket cs = new java.net.Socket( serverIPAddress , serverTCPPortNum );
    java.io.InputStream tcpIn = cs.getInputStream();
    java.io.OutputStream tcpOut = cs.getOutputStream();
    cs.close();
  } }
// the server-software is expected to reside on the computer of the client!
// the non-existent server gives a java.net.ConnectionException: Connection refused
/*
(protoServer runs in another terminal-window on the 'localhost')
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadipietro% javac ProtoClient.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadipietro% java ProtoClient
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadipietro%

(protoServer does not run on 'localhost')
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadipietro% java ProtoClient
Exception in thread "main" java.net.ConnectException: Connection refused
    at java.net.PlainSocketImpl.socketConnect(Native Method)
    ...
    at ProtoClient.main(ProtoClient.java:5)
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadipietro%
*/

```

Java
server-sockets
support Java's
standard-sockets.

Having established a client, now consider a server: Here, for the first time occurs Java's TCP-server-socket. The Java server-sockets act rather as a functional support of the standard TCP-sockets than a server-counterpart to a client-socket. Thus, CAUTION: An object of class-type `java.net.Socket` acts NOT automatically as a client-socket, this already has been stressed in the wake of the "WhoIs"-source-code on page 382!

```

class ProtoServer
{ static char tcpPortNum = (char)1025;
  static java.net.ServerSocket ss;
  static java.net.Socket cs; // cs = clientSocket

  public static void main( String unused[] ) throws java.io.IOException
  { ss = new java.net.ServerSocket( tcpPortNum );
    while(true)
    {
      cs = ss.accept(); // the server-socket is only needed to accept the connection!
      new Thread // thread
      ( new Runnable() // thread
        { public void run() // thread
          { try
            {
              java.io.InputStream tcpIn = cs.getInputStream();
              java.io.OutputStream tcpOut = cs.getOutputStream(); // do something
              cs.close();
            } catch( java.io.IOException e ){ e.printStackTrace(); }
          } }
        ).start(); // thread
    } } }
/* accept()-method blocks, until client contacts server,
therefore accept() has been placed inside while-loop and outside thread creation.
Eliminate the lines for threading and the parallel-server becomes a sequential server.
Eliminate the while-loop and the sequential-server becomes an one-time server.
*/

/*
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol > javac ProtoServer.java
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol > java ProtoServer
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol >
*/

```

The source-code above gives the prototype of a parallel-server. If the lines marked by the comment `// thread` are removed and the braces are placed properly, then the parallel-server turns into a sequential-server. If the while-loop is omitted, then the server accepts only one contacting client once.

One-Time Server: A server that accepts only one single connection and then shuts down.

Sequential-Server: A server that accepts multiple connections but only one connection at a time. The current connection has to be finished before another connection can establish itself. In the mean-time other attempts to contact the server yield at best some network-generated kind of server-not-available-message. Assuming an uncongested network, sequential servers respond only after their current client's session has been finished.

Parallel-Server: A server that, for each contacting request, spawns a new thread, thus being available for a new contacting request almost immediately. Parallel servers give the impression as being able to handle multiple communication-requests at the same time, parallel! Assuming an uncongested network, parallel servers are blocked only for about the time-interval of an `accept()`-call and the spawning of a new thread; after that time-period another client-request can be handled. So even parallel servers may be unavailable if too many requests reach them during some time period.

Methods for optimizing a server's performance range from furnishing faster computer-hardware to using network-specific services (DNS one-to-many contact distribution, page 360), down to writing hardware-specific server-programs, not written in Java. — The previous prototype gave the impression of how to write a client and an associated server. A simple HTTP-server can take the following form:

```

/* Processes only requests of the form
"GET http://<host> <optionalPortNumber> <pathWithSlashSeparators> HTTP1.1"
*/
class HTTPServer
{ static char tcpPortNum = (char)1025;
  public static void main( String unused[] ) throws java.io.IOException
  { java.net.ServerSocket ss = new java.net.ServerSocket( tcpPortNum );
    java.net.Socket cs;
    while(true)
    { cs = ss.accept(); // server-socket only needed to accept the connection
      new Thread( new ServerThread( cs ) ).start();
    } } }

class ServerThread implements Runnable
{ java.net.Socket cs;
  ServerThread( java.net.Socket cs ){ this.cs = cs; }
}

```

```

public void finalize() throws java.io.IOException{ cs.close(); }
public void run()
{ try
{ java.io.InputStream tcpIn = cs.getInputStream();
  java.io.OutputStream tcpOut = cs.getOutputStream();
  new HTTPCommunicate( tcpIn, tcpOut );
}catch( java.io.IOException e ){ e.printStackTrace(); }
} }

class HTTPCommunicate
{ HTTPCommunicate( java.io.InputStream tcpIn, java.io.OutputStream tcpOut )
    throws java.io.IOException

{ java.io.BufferedReader stringsIn =
  new java.io.BufferedReader( new java.io.InputStreamReader( tcpIn ) );

  String clientCall = stringsIn.readLine();
  System.out.println( "HTTP request: " + clientCall );
  String uriString = clientCall.substring( 3, clientCall.length() - 8 );
  // cuts out "GET" cuts out " HTTP1.1"
  System.out.println( "uriString: " + uriString );
  java.util.StringTokenizer uriTokens = new java.util.StringTokenizer( uriString, "/" );
  String protocol = uriTokens.nextToken();
  String hostPort = uriTokens.nextToken();
  String nonPath = protocol + "://" + hostPort;
  String path = uriString.substring( nonPath.length() ).trim();
  System.out.println( "path: " + path );

  java.io.FileInputStream fileIn = new java.io.FileInputStream( path );
  byte [] fileBytes = new byte[ fileIn.available() ];
  fileIn.read( fileBytes );

  //java.io.BufferedWriter stringsOut = // NOT NEEDED if file small enough
  //new java.io.BufferedWriter( new java.io.OutputStreamWriter( tcpOut ) );

  tcpOut.write( fileBytes );
  tcpOut.close(); // without close the client may get hooked
} }

/*
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol > javac HTTPServer.java
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol > java HTTPServer
HTTP request: GET http://192.168.0.1:1025/HTTPServer.java HTTP1.1
uriString: http://192.168.0.1:1025/HTTPServer.java
path: HTTPServer.java
Client request: GET http://192.168.0.1:1025/HTTPClient.java HTTP1.1
uriString: http://192.168.0.1:1025/HTTPClient.java
path: HTTPClient.java

marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol >
*/

```

Be reminded that server-programming on a really load-tolerant basis involves more than just proper software-production: The rate at which incoming requests can be processed depends vastly on the maximal capacity of the network-connection and on the choice of the computer-hardware¹⁶¹. — Additionally on the client-side, there is much less programming to be done:

```

/*
input format:
java HTTPClient (calls http://www.w3.org/) OR
java HTTPClient <host> <port> <path>
*/
class HTTPClient
{
  public static void main( String hostPortPath[] ) throws Exception
  {
    String host = null;
    String path = null;
    int port = 80; // default port for http-service
    if( hostPortPath.length != 3 )
    { host = "www.w3.org"; path = "/"; }
    else
    { host = hostPortPath[0]; port = Integer.parseInt( hostPortPath[1] );

```

¹⁶¹Like the programming of efficient databases or the configuration of a proper security-concept, the case of having a performant network-server involves much more than only software-production: In these cases properties and characteristics of the entire system, hardware and software have to be carefully considered. These considerations are just natural extensions of those considerations an average user has to have when managing an overfull hard-disk or a clogged network-connection at a Friday evening.

```

        path = hostPortPath[2];
    }
    java.net.Socket socket = new java.net.Socket( host, port ); // IP-level
    java.io.OutputStream out = socket.getOutputStream(); // TCP-level
    java.io.InputStream in = socket.getInputStream();
    // HTTP-level, command-string below:
    out.write( ( new String( "GET " + "http://" + host + ":" +
        port + path + " HTTP1.1\r\n\r\n" ) ).getBytes() );
    out.flush();
    int intUsedAsByte;
    while( ( intUsedAsByte = in.read() ) != -1 )
    { System.out.write( intUsedAsByte );
    } } }

/*
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadipietro% javac HTTPClient.java
(needs a working Internet-connection otherwise throws a java.net.UnknownHostException: www.w3.org)
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadipietro% java HTTPClient
HTTP/1.1 200 OK
Date: Mon, 09 Jul 2001 14:42:10 GMT
Server: Apache/1.3.6 (Unix) PHP/3.0.15
Content-Location: Overview.html
Opt: "http://www.w3.org/2000/P3Pv1";ns=11
...
</body>
</html>

(needs a working local IP-network otherwise throws a java.net.ConnectException: Operation timed out)
[localhost:~/JavaCodeFiles/CommunicatingComputers/
InternetProtocol] andreadipietro% java HTTPClient 192.168.0.1 1025 /HTTPServer.java
class HTTPServer
{ static char tcpPortNum = (char)1025;
  public static void main( String unused[] ) throws java.io.IOException
  ...
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadipietro%
*/

```

A more general approach to constructing clients presents the `java.net.URL`-class and its affiliates:

Java: URL, Application-Protocol- and Content-Handlers for CLIENTS

Java: URL, Application-Protocol- and Content-Handlers for CLIENTS — Remember the introduction of the Uniform Resource Locator (URL) on page 364 and the URL-description beginning on page 373. Java uses the pre-written class `java.net.URL` to BUNDLE CLIENT-SIDE SERVICES ASSOCIATED WITH A GIVEN UNIFORM RESOURCE LOCATOR. That especially includes the support for standard application-protocols like HTTP or FTP, but that does not exclude the, to be introduced, general use of the class `java.net.URL`! When instantiating an object of type `java.net.URL`, the Uniform Resource Locator usually is given as a `String`-argument of the constructor `URL(String url)`. This `String`-argument takes approximately the following format:

protocol://host:port file

Alternatively the constructor of the pre-written class `java.net.URL` has been overloaded to take the Uniform Resource Locator broken into several strings: `protocol`, `host`, `file` and an optional 4-byte-`int`-value `port`:

```

new URL( String url )
new URL( String protocol, String host,          String file )
new URL( String protocol, String host, int port, String file )

```

With TCP/UDP transport-protocols, *port*-values of “-1” indicate that a default port should be used. The *file*-variable of type `String` represents an absolute path of a directory, given in URL-encoded form (page 375). This means for example, the file-separator always is a slash (/) and the absolute path begins with an initial slash “/...” and in those cases in which the `String`-object(s) use non-ASCII-characters they have to be made platform-independent by URL-encoding them with the `java.net.URLEncoder`-class.

The above gave a first impression of how to instantiate an object of the pre-written class-type `java.net.URL`, now what services does such an object supply?

- An object of class `java.net.URL` stores a **single Uniform Resource Locator**, that generally remains unchanged after the instantiation.
- An object of class `java.net.URL` ALSO ACTIVATES VARIOUS SERVICES around the network-connection and the processing of the exchanged data:

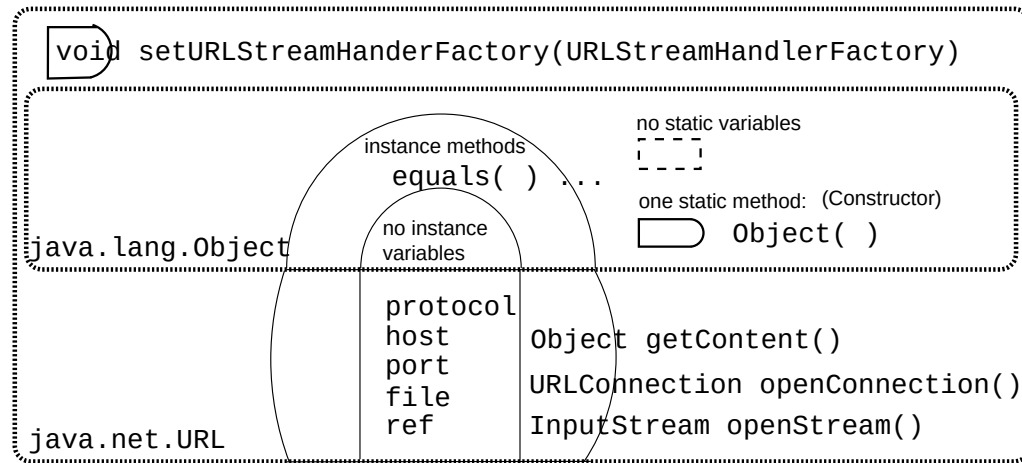
- The overloaded constructor of the class `java.net.URL` **parses, changes, extracts those parts of the string which represent clearly defined pieces of a Uniform Resource Locator** either by taking those values readily (partially) separated:
`URL(String s),`
`URL(String protocol, String host, String file),`
`URL(String protocol, String host, int port, String file),`
given as constructor arguments or by arguments of the
`set(String protocol, String host, int port, String file, String ref)`-method.
And there can be found the methods for retrieval: `getProtocol()`, `getHost()`, `getPort()`,
`getFile()`, `getRef()`, `toString()`.
On instantiation, the protocol-string is extracted and compared to the names of the supported application-protocols; if there exists no match, then a `java.net.MalformedURLException` is thrown. Where can be seen what application-protocols are supported? See next item:
- External files/classes are used to **support different new application-protocols and data-types** without needing to change the `java.net.URL`-class. By default, application-protocols are found in the package `sun.net.www.protocol`. By default, MIME (Multipurpose Internet Mail Extension, page 368)-types are found in the package `sun.net.www.content`. These packages are stored inside the classes.zip-file (MRJClasses.zip in the Mac OS Version 9.x and below) or in Java Runtime Systems of Versions 1.2 and higher in the `rt.jar`-file¹⁶². Standard application-protocol-handlers to be found include one for the HyperText Transfer Protocol (HTTP) under the name `http`, one protocol-handler for the File Transfer Protocol (FTP) under the name `ftp` and a protocol-handler for the Simple Mail Transfer Protocol (SMTP) under the name `mailto`. Standard MIME-types that are recognized by recognizable content-handlers are `text/generic`, `text/plain`, `image/gif` and `image/jpeg`. How to include handlers for own application-protocols and (MIME-) types is shown later.
- Having analyzed the given Uniform Resource Locator and having found the previously furnished and related protocol- and content-handlers, the URL-object is ready for **connecting to the network**, thereby activating the resources the given Uniform Resource Locator points to. This means using the following methods: The `openStream()`-method returns the reference to a pure content-stream (the raw-data) without any type-processing done. The `getContent()`-method may return a range of references: A reference to a `java.awt.image.ImageProducer`-object¹⁶³ for `image/gif`- and `image/jpeg`-data files it received. Or that method may return a reference to a `java.io.InputStream`-object for files of MIME-types `text/generic` and `text/plain`. Thus a `java.net.URL`-object also does much of the **application-related transformation of the received data!**

<code>public final java.io.InputStream</code>	<code>openStream()</code>	throws <code>java.io.IOException</code>
<code>public final java.lang.Object</code>	<code>getContent()</code>	throws <code>java.io.IOException</code>
<code>public java.net.URLConnection</code>	<code>openConnection()</code>	throws <code>java.io.IOException</code>

The graphic below omits the constructors, the `set(...)`- and `get(...)`-methods for entering the String-objects for *protocol*, *host*, *file* and *ref* variables and the int-value for the *port*-variable. As well as the `toExternalForm()`-method which returns a reference to a String-object reproducing the entire given Uniform Resource Locator.

¹⁶² Refer to page 249 for being given finding-strategies when searching the file `rt.jar` (classes.zip or MRJClasses.zip). For example, in Linux consider a directory-path like `/usr/lib/jdk1.3/jre/lib` to find that file.

¹⁶³ The class `java.awt.image.ImageProducer` is introduced around page 529 and beginning with page 606.



The four `java.net.URL`-specific methods in the graphic above, one of them is static, enable a `URL`-object to access the network resources as a client (not as a server). The `java.net.URL`-class is only extended if a kind of Uniform Resource Locator principally cannot be stored in the given format. No much knowledge more is needed to use the default mechanism of the `java.net.URL`-class!

java.net.URL — The Default Mechanism

What does an `URL`-object do?

`java.net.URL` — The Default Mechanism — What does an object of class-type `java.net.URL` essentially do? An `URL`-object takes a Uniform Resource Locator, as a given `String`-object, analyzes it for making it reference proper protocol- and handler-routines, and then uses the given `URL` for establishing a fully-functional (client-)network-connection. Furthermore, the `URL`-object addresses the network-resources and receives the related data, including the service of giving the data a form ready for further platform-specific processing! The default mechanism thus makes it possible to write very short client-programs:

```

class URLClient
{
    public static void main( String url[] ) throws Exception
    {
        java.net.URL u = new java.net.URL( url[0] );
        java.io.BufferedReader stringObject =
            new java.io.BufferedReader( new java.io.InputStreamReader( u.openStream() ) );
        String theLine;
        while( ( theLine = stringObject.readLine() ) != null )
        {
            System.out.println( theLine );
        }
    }
}

```

(a http-server should be running on host-computer with the IP-Address 192.168.0.1)

```

[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadpietro% javac URLClient.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadpietro% java URLClient http://192.168.0.1/
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
...
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] andreadpietro%

```

Here even the application-protocols can be hidden more or less effectively from the programmer. (No visible HTTP, no visible `http-GET` call!) In this way the `java.net.URL`-class hides a lot of complexity as already has been indicated. Looking further into the mechanisms triggered by the `java.net.URL`-class makes it possible to write highly flexible client-applications.

Java-Concept: Uniform Resource Locator in Client-side Connecting to a Network

Java-Concept: Uniform Resource Locator in Client-side Connecting to a Network — Again: An object of class-type `java.net.URL` takes a Uniform Resource Locator, as a given `String`-object, analyzes it to identify references to proper protocol- and handler-routines, and then establishes a fully-functional (client-)network-connection to contact and retrieve the resources, as indicated by the given Uniform Resource Locator!

The default mechanisms of the `java.net.URL`-class use the basic infrastructure of the `URL`-class together with pre-written handler-routines, rather invisibly for the application-programmer. But, if acquainted to this basic infrastructure of the `URL`-class, a Java programmer can customize protocol- and content-handling of the `java.net.URL`-class.

The following paragraphs try to put in words what happens in principle when an object of class-type `java.net.URL` is instantiated:

`String`-
object

protocol : resource First there has to be a string that represents the Uniform Resource Locator, coming from file or memory (user-input or else).

URL-object

protocol : **resource** On instantiation of the URL-object, the constructor isolates the first part to determine the application-protocol.

URLStream-Handler-object

protocol : **resource** With that information, the constructor creates a PROTOCOL-SPECIFIC OBJECT from a given Java class-file, the protocol-handler. This object analyzes the rest of the given Uniform Resource Locator-string and returns the information for completing the URL-object (protocol-handling step1). Furthermore there is furnished a link to a class which can create the actual protocol-specific network-connection:

URL-Connection-object

Input/Output-Streams

Socket Over the link furnished above, the URL-object can create a connection-object that implements the application-protocol-specific connection to the network (protocol handling step2) so that data can be transformed to and from the transport-specific format as needed,

Content-Handler-object

content and, if needed, the connection-object may create another (CONTENT-SPECIFIC)-OBJECT that translates the data-type into a platform-specific form too. That means de-/encrypting, (de-)compressing, rendering the content-data, especially data coded as a MIME-type (content-handler-functionalities).

Thus a `java.net.URL`-object can support new application-protocols and new content-types just by adding protocol-handlers and content-handlers as Java class-files of byte-code. Controlled by the URL-object, the Java Runtime Environment (JRE) dynamically loads these classes as any other class!

Optional: `java.net.URL` — The Potential

String object



Optional: `java.net.URL` — The Potential — Beginning the instantiation of a `java.net.URL`-object means calling the constructor with any kind of String argument(s), this sets off the following procedure: The URL-constructor AUTOMATICALLY

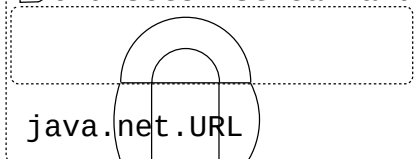
URL-constructor

- **parses** for the name of an application-protocol inside the String-object **or takes** the protocol-string from an overloaded constructor directly
- and **searches** — using the name of the protocol — for the protocol-specific class-extension of the class `java.net.URLStreamHandler`:

extension of the URLStreamHandler

1. The URL-constructor searches in the already loaded classes (cache) within the current Java Runtime Environment (JRE) and then
2. searches using the possibly implemented interface-call
`java.net.URLStreamHandlerFactory.createURLStreamHandler(String protocol).`

void setURLStreamHandlerFactory(URLStreamHandlerFactory)

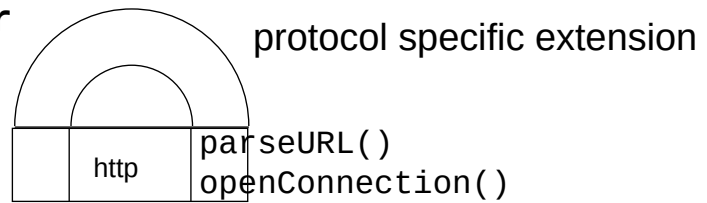


The class `java.net.URL` has a static method which can be called only once during one session of the Java Runtime Environment: `setURLStreamHandlerFactory( URLStreamHandlerFactory shf )`. There the reference `shf` points to an object that implements the interface `URLStreamHandlerFactory`, which means: The object defines the method `createURLStreamHandler(String protocol)` that is supposed to instantiate a `java.net.URLStreamHandler`-object — for more see the next paragraphs.

3. Finally, the URL-constructor searches relative to the `rt.jar` file¹⁶⁴ in the folder/directory `/sun/net/www/protocol/protocolName/` for the package `sun.net.www.protocol.protocolName`

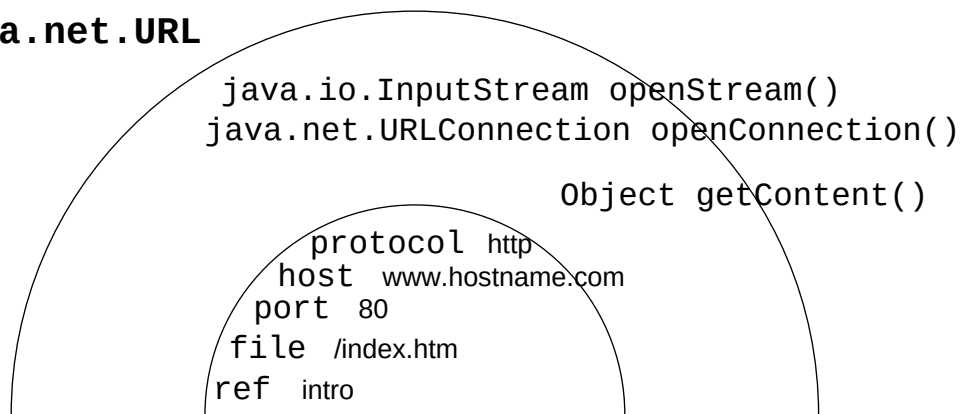
otherwise is thrown a `java.net.MalformedURLException`

java.net.URLStreamHandler



- Then the URL-constructor (automatically) instantiates the found protocol-specific extension of the `java.net.URLStreamHandler`-class,
- calls its `parseURL()`-method that, by default and by convention, has to analyze the non-protocol part of the given string that is supposed to represent a Uniform Resource Locator. The entries into the URL-object are thereby completed and
- the URL-object is instantiated.

java.net.URL

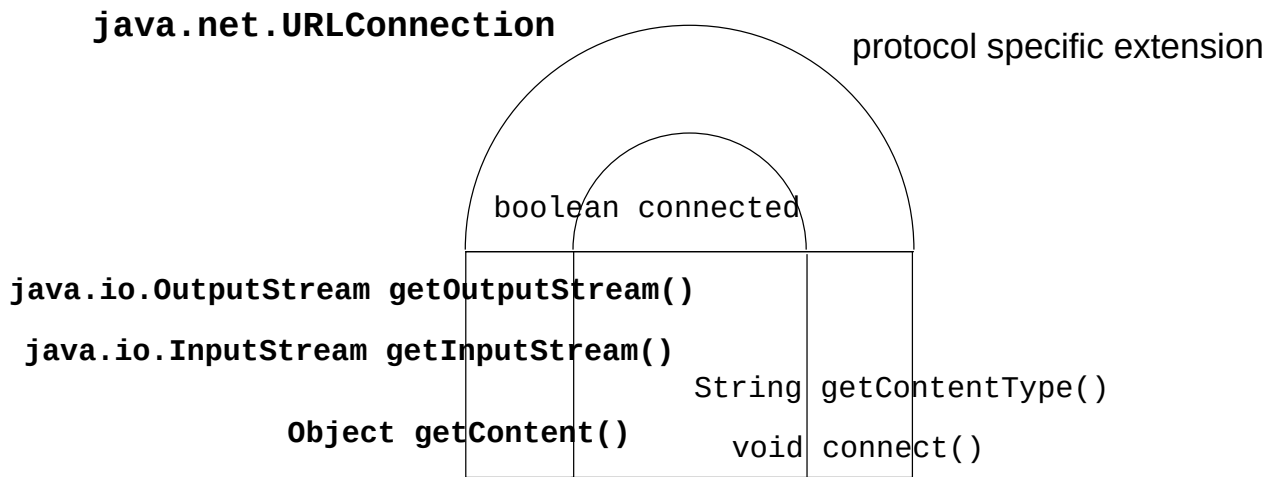


The object of the protocol-specific extension `java.net.URLStreamHandler`-class possesses a factory-method `openConnection()` for linking the URL-object to a newly created object (of type `URLConnection`) that realizes the actual network-connection. — So far a URL-constructor with a `String`-object(s) searched for the name of an (application-)protocol and used a protocol-specific object to fully analyze the “URL-stream” (= `String`-object). This made it possible to instantiate the URL-object. (so far there have been used and created three objects of the following respective types: (`URL`-)String, `URLStreamHandler`, `URLConnection`) Now these preparations make it possible to instantiate the fourth object that creates and manages the protocol-specific network-connection: The protocol-specific extension of the `java.net.URLConnection`-class.

The `java.net.URL`-object has three methods (as previously mentioned) for actually communicating with the server-computer (= `host`) in the network: `openStream()`, `openConnection()`, `getContent()`. All three methods use the factory-method `java.net.URLStreamHandler.openConnection()` which creates a protocol-specific `java.net.URLConnection`-object:

protocol-specific
URLConnection-
object

¹⁶⁴In Java Runtime Environments, of versions below number 1.2, standard and non-standard pre-written packages were given in a file called `classes.zip`.



java.net.URL.openStream()	means	java.net.URLStreamHandler.openConnection().getInputStream()
java.net.URL.openConnection()	means	java.net.URLStreamHandler.openConnection()
java.net.URL.openContent()	means	java.net.URLStreamHandler.openConnection().getContent()

- By default and convention, the instantiation of sockets and streams to connect to the network and their protocol-specific management is done in the `connect()`-method of that object of class-type `java.net.URLConnection`. Therefore methods like `java.net.URLConnection.getInputStream()` and `java.net.URLConnection.getContent()` have to call the `java.net.URLConnection.connect()`-method and set the `connected`-variable to true.
- By default and convention, the methods `getInputStream()` and `getOutputStream()` read and write PURE BYTE-STREAMS OF CONTENT from or to the server (whose name has been given by the *host*-variable of the Uniform Resource Locator). These streams, as well as the `getContent()`-method supply NO PROTOCOL-SPECIFIC OR PROTOCOL-RELATED INFORMATION!
- By default and convention, `java.net.URLConnection.getContentType()` returns a `String`-reference to the (MIME)-type of the retrieved content. This type-information is produced by extracting it from the protocol-management in the `connect()`-method. The `getContentType()`-method is
- AUTOMATICALLY consulted by the method `java.net.URLConnection.getContent()` that uses `URLConnection.getContent()` the type-name (MIME-type *type/subtype*), for searching a type-specific content-handler:
 1. The method searches in already loaded classes (cache) of the current Java Runtime Environment,
 2. it searches by testing the possibly implemented method-call `java.net.ContentHandlerFactory.createContentHandler(String mimetype)`. The Java Runtime System is told where to find this method by a call of `java.net.URLConnection.setContentHandlerFactory(ContentHandlerFactory chf)`, which is possible once in the lifetime of the Java Runtime Environment; such a call registers a reference `chf` to the object that implements the interface `ContentHandlerFactory` indicating a location of the content-handler's code, or

void setContentHandlerFactory(java.net.ContentHandlerFactory)

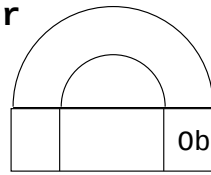
java.net.URLConnection
 3. Finally, the `getContent()`-method searches relative to the `rt.jar` file¹⁶⁵ for a file with the relative path-name `/sun/net/www/content/type/subtype.class` (using the name of the MIME-type *type-subtype* in question where dashes “-” become “_” underscores). This file contains a class with the fully qualified package-name `sun.net.www.content.type.subtype`

¹⁶⁵ In Java Runtime Environments of versions below Number 1.2, standard and non-standard pre-written packages were given in a file called `classes.zip`.

otherwise the method returns null and method `getContent()` throws a `java.io.IOException` (all similarly to the `java.net.URL`-constructor)

Now the control is given back to the `java.net.URL`-constructor:

java.net.ContentHandler



type specific extension

- Then the URL-constructor (AUTOMATICALLY) instantiates the found type-specific extension `java.net.ContentHandler`-class,
- and calls its `java.net.ContentHandler.getContent()`-method (this makes `java.net.URL.getContent()` call the method `java.net.URLConnection.getContent()` which calls the method `java.net.ContentHandler.getContent()`. All in all three different methods of the same name calling each other).
- This method now uses the `java.net.URLConnection.getInputStream()`-method to transform that content-stream according to the type-information into a Java object whose reference is returned.

All these steps are realized for a simple realization of the HTTPProtocol in the following client. What's the advantage? New protocols and types can be added (dynamically) to the client by simply adding files and loading class-files even during the runtime of the client!

```
class URLMechanisms implements java.net.URLStreamHandlerFactory, java.net.ContentHandlerFactory
{ public static void main( String uniformResourceLocator[] ) throws Exception
{ URLMechanisms x = new URLMechanisms();
  java.net.URL.setURLStreamHandlerFactory( x );
  // the method-call above signalizes the system where to search for a create...()-method (***)

  java.net.URLConnection.setContentHandlerFactory( x );
  java.net.URL url = new java.net.URL( uniformResourceLocator[0] );
  java.io.BufferedReader br;
  try
  { br = new java.io.BufferedReader(                                     // (++default:)
    new java.io.InputStreamReader( (java.io.InputStream)url.getContent() ) );
  }catch( java.lang.ClassCastException e )
  { br = (java.io.BufferedReader)url.getContent(); }                      // (++defined below)
/* the try-catch-clause could be avoided
--- by letting the getContent()-method below return an InputStream or
--- by analyzing the returned entity of type Object with the
    java.lang.reflect package and choosing a method of processing.
*/
  String s;
  do
  { s = br.readLine();
    if(s == null) break;                                     // finish if StringBuffer is empty
    if(s.trim().length() == 0) continue; // print no empty lines
    System.out.println(s);
  }while(true);
}
/* This is the end of the regular program;
   this regular program also works with the default settings,
   which become active if the first three lines of the main()-methods are outcommented.
   The code below implements service-modules which have been introduced by calling
   the x-argumented methods above.
*/
// (***) create...()-methods tell the system from where to take the service-modules.

/* Here the service-modules are introduced as being simple class-files,
   given by the source-code below.
*/
// Below define a triggerable method, during URL instantiation
public java.net.URLStreamHandler createURLStreamHandler( String protocol )
{
  //protocol = protocol.toLowerCase();
  // URL returns String-reference to protocol anyway in lower-case!
  try
  { Class urlStreamHandlerName = Class.forName( protocol + "URLStreamHandler" );
    Object o = urlStreamHandlerName.newInstance(); // creation = instantiation
    return ( java.net.URLStreamHandler )o;
  }catch( Exception e ){ e.printStackTrace(); return null; }
}

// Triggerable method, during URL instantiation.
// Called by URLConnection.getContentHandler(String contentType) (default or convention)
```

```

    public java.net.ContentHandler createContentHandler( String contentType )
    { try
      { Class urlStreamHandlerName = Class.forName( contentType + "ContentHandler" );
        Object o = urlStreamHandlerName.newInstance(); // creation = instantiation
        return ( java.net.ContentHandler )o;
      }catch( Exception e ){ e.printStackTrace(); return null; }
    } }

/*
the various service-modules to which the create...()-methods refer
*/
class httpURLStreamHandler extends java.net.URLStreamHandler
// lower-case because URL returns String-reference to protocol in lower-case!
{ protected java.net.URLConnection openConnection( java.net.URL url ) throws java.io.IOException
  { return new HttpURLConnection( url ); } // see below...
}
class HttpURLConnection extends java.net.URLConnection
{ final static char defaultPort = 80; // default port of HTTPProtocol
  java.net.URL url;
  java.io.PrintWriter textOutputStream;
  java.io.InputStream inputStream; // own identifier for use by getInputStream()-method
  java.io.BufferedReader textInputStream, contentIn;
  boolean headerStatesHtml = false;
  public HttpURLConnection( java.net.URL url )
  { super( url ); // necessary because URLConnection has no standard constructor
    this.url = url;
  }
  public void connect() throws java.io.IOException // contains HTTP-mechanisms
  { if( !connected ) // variable taken from java.net.URLConnection
    { char port;
      int potentialPort = url.getPort(); // if -1 then default port
      if( potentialPort > 0 && potentialPort < 65535 ) port = (char)potentialPort;
      else port = defaultPort;
    } // socket and streams
    java.net.Socket socket = new java.net.Socket( url.getHost(), port );
    java.io.OutputStream outputStream = socket.getOutputStream();
    textOutputStream =
      new java.io.PrintWriter( new java.io.OutputStreamWriter( outputStream, "8859_1" ), true );
    inputStream = socket.getInputStream();
    textInputStream =
      new java.io.BufferedReader( new java.io.InputStreamReader( inputStream, "8859_1" ) );
    // get header and analyze header
    textOutputStream.println( "HEAD " + url.getFile() + " HTTP/1.0\r\n\r\n" );
    String s; int headerLength = 2; // last linefeed
    while( ( s = textInputStream.readLine() ).length() != 0 )
    { headerLength = headerLength + s.length() + 2; // 2 for \r\n line endings
      java.util.StringTokenizer st = new java.util.StringTokenizer(s);
      String firstToken = st.nextToken().toLowerCase();
      String secondToken = st.nextToken().toLowerCase();
      if( ( firstToken.startsWith( "content-type" ) && secondToken.startsWith( "text/html" ) ) )
        { headerStatesHtml = true; }
    } // textOutputStream.close(); textInputStream.close(); socket.close(); // no difference
    // 2nd socket needed because of HTTP1.0, this shouldn't be needed in HTTP1.1
    if( headerStatesHtml == true )
    {
      java.net.Socket socket2 = new java.net.Socket( url.getHost(), port );
      textOutputStream = new java.io.PrintWriter
        ( new java.io.OutputStreamWriter( socket2.getOutputStream(), "8859_1" ), true );
      inputStream = socket2.getInputStream();
      textOutputStream.println( "GET " + url.getFile() + " HTTP/1.0\r\n\r\n" );
      for( int i = 0; i < headerLength; i++ ) { inputStream.read(); };
      connected = true;
    }else{ throw new java.io.IOException( "No HTML-file" ); }
  } }
  public java.io.InputStream getInputStream() throws java.io.IOException
  { if( !connected ) connect();
    return inputStream;
  }
  public String getContentType()
  { try
    { if( !connected ) connect();
      if( headerStatesHtml == true ) { return "html"; }
      else { System.out.println( "getContentType returns null" ); return null; }
    }catch( Exception e ){ e.printStackTrace(); return null; }
  } // used by this class's getContent()-method
  // protected void finalize() throws java.io.IOException{ textOutputStream.close(); httpRawIn.close(); }
  // This closes the socket.
}

// ---- here the HTML-tags are extracted for a plain-text version without hyperlinks ----

class htmlContentHandler extends java.net.ContentHandler
{
  // (+++) automatically called by method java.net.URL.getContent()
  public Object getContent( java.net.URLConnection urlConnection )
  { try
    { java.io.InputStream is = urlConnection.getInputStream();
      java.io.InputStreamReader ris = new java.io.InputStreamReader( is );
      HTMLCommandFilter frs = new HTMLCommandFilter( ris );
      java.io.BufferedReader bris = new java.io.BufferedReader( frs );
    }
  }
}

```

```

        return bris;
    } catch( Exception e ){ e.printStackTrace(); return null; } // extract content and transform it
} }

/* How to archive pure relaying action by the getContent( java.net.URLConnection )-method?
Just outcomment everything within its try-clause, except the first line
java.io.InputStream is = urlConnection.getInputStream();
and change the last, then second line to
return is;
To follow the program-run, intersperse a System.out.println();
at the beginning of every method-call.
*/
class HTMLCommandFilter extends java.io.FilterReader
{ HTMLCommandFilter( java.io.Reader in ){ super( in ); }
  boolean transfer = true;
  char character;
  public int read() throws java.io.IOException
  { char c = (char)in.read();
    if( c == '<' ) { transfer = false; return 0; }
    if( c == '>' ) { transfer = true; return 0; }
    if( transfer == true ) return (int)c; else return 0;
  }
  public int read( char[] c ) throws java.io.IOException
  { int len;
    len = in.read(c, 0, c.length);
    return len;
  }
  public int read( char[] c, int off, int len ) throws java.io.IOException
  { len = in.read(c, off, len);
    for( int i = 0; i < len; i++ )
    { character = (char)c[i];
      if( character == '<' ){ transfer = false; c[i] = 0; }
      if( character == '>' ){ transfer = true; c[i] = 0; }
      if( transfer != true ) c[i] = 0;
    }
    return len;
  }
} }

/*
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] marycosway% javac URLMechanisms.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/
InternetProtocol] marycosway% java URLMechanisms http://www.w3.org/
The World Wide Web Consortium
Leading the Web to its Full
Potential...
Activities | Technical&#160;Reports | Site&#160;Index | About&#160;W3C | Contact
The World Wide Web Consortium (W3C) develops
interoperable technologies (specifications, guidelines, software, and
...
Keio), All Rights Reserved. W3C liability, trademark, document
use and software licensing rules
apply. Your interactions with this site are in accordance with our public and Member privacy
statements.
[localhost:~/JavaCodeFiles/CommunicatingComputers/InternetProtocol] marycosway%
*/

```

Next to the `java.net.URL`-class are two other URL (Uniform Resource Locator)-based communication-mechanisms on application-level: The applet- and the servlet-mechanisms:

Java: Applets as Programs in a Client-Side-Controllable Environment

Java: Applets as Programs in a Client-Side-Controllable Environment — Recall that Java-applications are programs that run in the Java Runtime Environment (graphic on page 51). A Java-application is given by a class with a `main()`-method as a member; this `main()`-method, after compilation into a class-file, can be invoked externally by issuing the `java`-command in a terminal-window of the computer's operating-system. — Contrastingly, remember from the conceptional introduction of the notion “applet” on page 67, that an applet is given by a class which extends the pre-written class `java.applet.Applet` (or the class `javax.swing.JApplet`). After compilation into byte-code, such an applet is run in a special applet runtime-environment¹⁶⁶ within the Java Runtime Environment (JRE). To start the applet, end-users enter a Uniform Resource Locator (URL) into the input-line of some browser-software or enter the `appletviewer`-command into the terminal-window of the operating-system. URL or the `appletviewer`-command trigger the successive run of the applet's `init()`-, `start()`-, `stop()`- and `destroy()`-methods.

Ideally, the end-users, but at least the producers of an Applet Runtime Environment have full control over the security aspects of the running applet. The programmers of applets are supposed to cope with these security restrictions, as given by the end-user or the producer of the Applet Runtime Environment. (The Java security facilities introduced with Java Version 1.2 give the end-user similar means of control over regular Java-applications.)

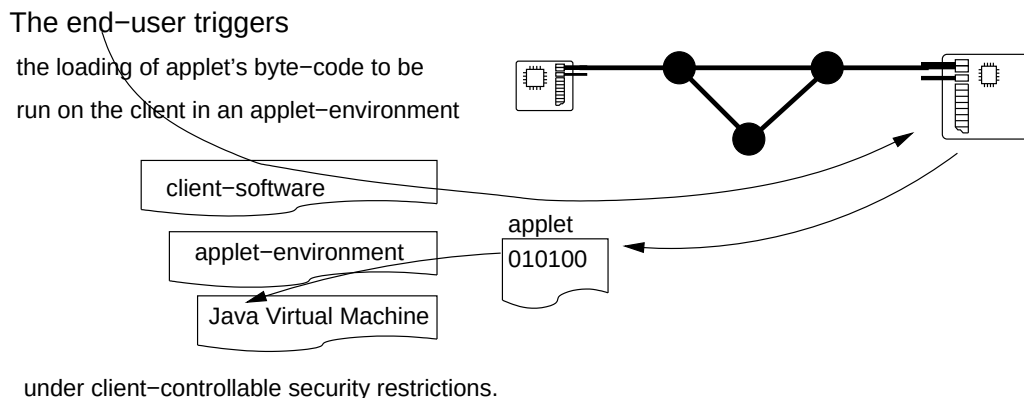
¹⁶⁶Also called “sandbox” running within a Java-enabled Internet-browser, `appletviewer` or the explicit program on page 619.

This section presents only those aspects of applet programming that concern the network and the applet runtime-environment, whereas security and display-related aspects are considered later (beginning with pages 475 and 613, respectively).

Applet-Concept

Applet-Concept — So keeping in mind what has been said about applets around page 67, consider the compilation below:

- APPLETS ARE PROGRAMS but
- APPLETS RUN IN AN END-USER-CONTROLLABLE ENVIRONMENT (inside the Java Runtime Environment), like appletviewers or Internet-browsers, so that the end-user ideally can restrict the applet's access to the platform. This makes applets different from any other downloadable program. Usually the byte-code of applets is loaded by a client from a server.



- References to the applet's byte-code are EMBEDDED IN HTML-DOCUMENTS, by using the `<applet> ... </applet>`-tag pair; in those documents. Applets are loaded and run during the run-time of the Java-enabled HTML-browser, therefore
- APPLETS ARE USUALLY INVOKED BY CALLING AN HTML-DOCUMENT WITH A UNIFORM RESOURCE LOCATOR (URL), that means applets are URL-invoked programs.
- Applets produce a little GRAPHICAL USER INTERFACE (GUI) of their own, when being made non-trivially visible by implementing their `paint()`-method.

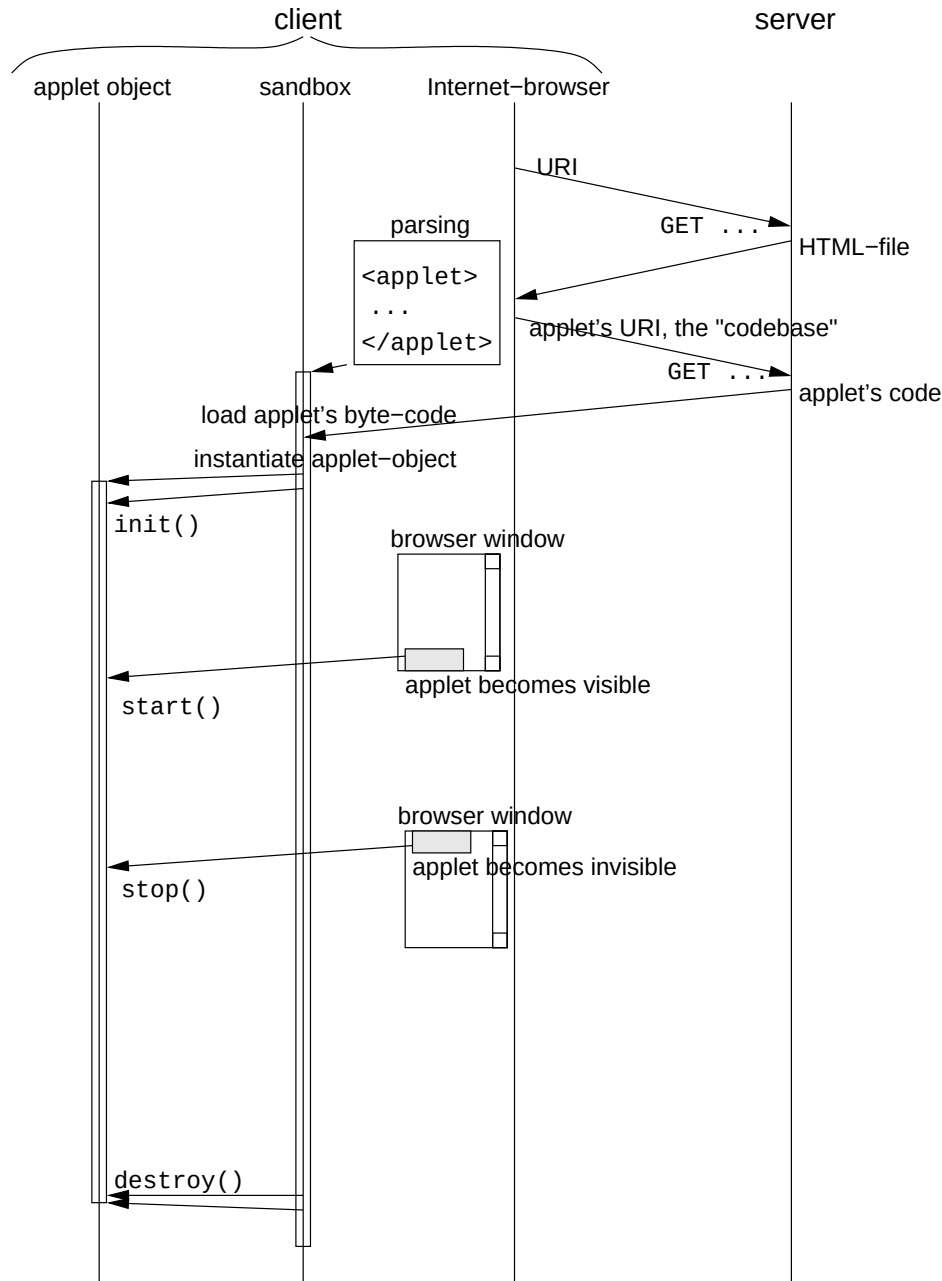
Applet-Infrastructure

- JDK relation: The Standard Java Development Kit (JDK) furnishes the necessary pre-written classes (base-classes) `java.applet.Applet` and `javax.swing.JApplet`; as well as the command-implementation for test-running an applet, the `appletviewer`-command.
- THE DENOTATION "APPLET" IS USED FOR either
 1. the SOURCE-CODE of a class derived from the base-class `java.applet.Applet` or another base-class `javax.swing.JApplet` (which furnishes a reduced Swing-support),
 2. the BYTE-CODE generated with the Java compiler (`javac`) or
 3. an OBJECT, an implementation of such an (applet-)class.
- Security control on the side of an end-user should be included in the browser-software, see page 460 and page 475.
- The Java programmer's code can be found in several triggerable-methods (the equivalent to the application's `main()`-method), which all bear the modifiers **public** and **void**
 - `paint()`, called by the Java Runtime Environment (JRE) when the display is changed.

- `init()`, called by the applet-container¹⁶⁷ after the applet has been instantiated,
- `start()` called by the applet-container when the applet becomes visible on the screen,
- `stop()` called by the applet-container when the applet becomes INvisible on the screen and
- `destroy()` called by the applet-container before the applet is deinstantiated (an equivalent to the `finalize()`-method of any instantiation of type `Object`, see page 152).

The graphic below shows an applet's run and the associated flow of interactions between the different computational entities: The end-user triggers the dispatch of a Uniform Resource Identifier (URI¹⁶⁸) to a server. In this example, the sandbox or applet runtime-environment is given by a Java enabled Internet-browser. Java enabled Internet-browsers recognize `<applet>...</applet>`-tag pairs, and thereupon do start a Java Runtime Environment (JRE) with its applet-running software.

APPLET



Observe: After the initial loading-process, the client needs no additional contact with the server. The `init()`-method is only called once during the time-interval of existence of the applet. The `start()`-

¹⁶⁷Alternative name for the applet runtime-environment or the applet-sandbox.

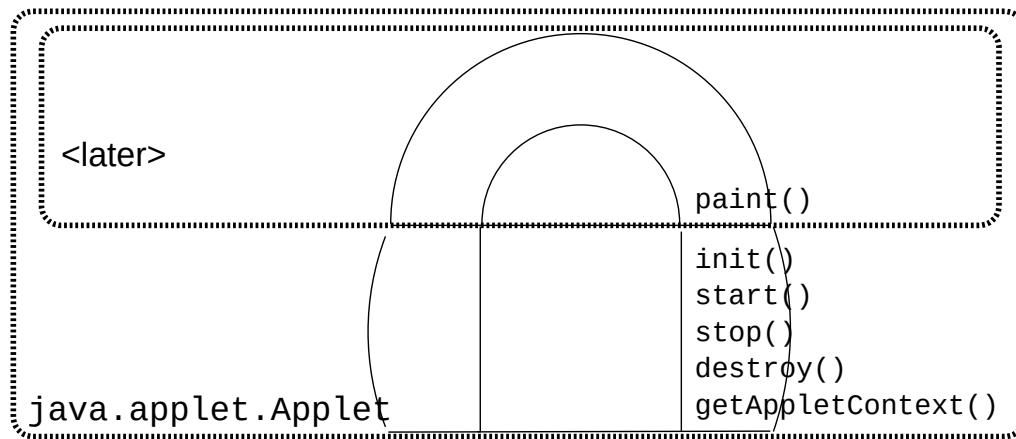
¹⁶⁸A Uniform Resource Identifier (URI) is a generalization of the Uniform Resource Locator (URL). Short and unprecise: A URI can remain unchanged even if the URL changes due to relocations inside the server's file-system or due to a renaming of the entire server. For an example see page 411. Thus a URI is somewhat more general than a Uniform Resource Locator URL, see also the gray frame on page 365.

and `stop()`-methods may be called arbitrarily often, always automatically, when the end-user makes the applet visible or invisible, for example by scrolling the HTML-page into which the applet usually has been embedded.

The Java programmer overrides these methods and uses them to furnish custom-written source-code.

A Simple Applet

Applets are predestined to be used in a Graphical User Interface (GUI), therefore they extend the class (`java.awt.Panel`) of a screen-component. The associated class-hierarchy as well as functionalities of applets, related to the Graphical User Interface, are considered beginning with page 615.



For an extensive description of the class `java.applet.Applet` and its underlying class-structure, see page 615 and for the specification of all related methods see the [Java Platform API Specification \[JPAPIS\]](#).

Consider the following little file of an HTML-document; an `<applet> ... </applet>`-tag-pair, signals that a file of Java byte-code (`HelloWebApplet.class` within the same directory of that HTML-file) is referenced by that HTML-document. On having loaded this HTML-file, the client's Java enabled browser, thus knows it has to load the applet's byte-code too. But the client's Java enabled browser also has to start a Java Runtime Environment (JRE) with the applet-running extension for starting the applet within the on-screen presentation of the HTML-document:

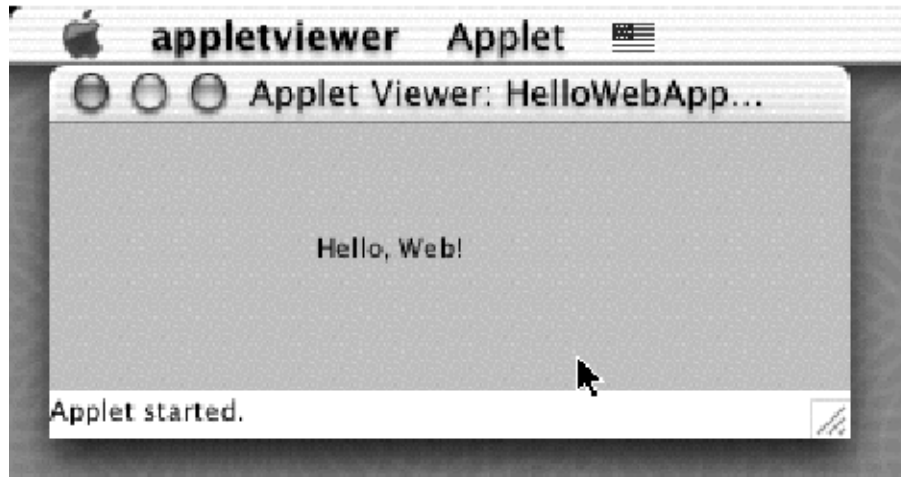
```
<!-- This is a comment in HTML (comments in Java use '/*' or '/*' ... '*/')
      (comments in Unix scripts begin with '#')
-->

<!-- Name of file: HelloWeb.html -->
<html>
  <head> <h3> Hello Web Applet </h3> </head>
  <body> Standard HTML text before the embedded applet-window.
    <applet code=HelloWebApplet width=300 height=100 >
    </applet>
    Standard HTML text past the embedded applet-window.
  </body>
</html>

<!--
[localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway% javac HelloWebApplet.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway% appletviewer HelloWeb.html
^C ( <--- type Ctrl-C, to stop appletviewer)
[localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway%
-->

<!--
(Or view a HTML-Documnt with Java-enabled Internet-browser.
 Or use AppletLauncher applications like the one provided as a utility with the Mac OS X.)
-->
```

Thus, when made visible on-screen, this HTML-document triggers the loading of the applet's byte-code from the server.



The applet's byte-code has been compiled before from the following source-code. This applet should greet the user with a message written onto a window on the desktop:

```
public class HelloWebApplet extends java.applet.Applet
{
    public void init(){ this.setBackground( java.awt.Color.lightGray ); }
    public void paint( java.awt.Graphics g )
    { g.drawString( "Hello, Web!", 100, 50 );
    } }
}
```

The applet's triggerable `paint()`-method makes it possible to put information directly onto the screen. The Java Runtime Environment (JRE) furnishes a toolbox of basic graphical methods for drawing on-screen inside the applet's window: The reference of *abstract* type `java.awt.Graphics`, is fully valid, as has been described on page 184. This reference allows to access native code for drawing on-screen.

But applets can also work without a graphical functionality, nevertheless their existence ideally is always documented by a visible, applet-identifying, frame on the desktop or inside a desktop-window! This forced visibility and identification supports security: An applet should not make itself invisible and consume processor-time. Or the applet should not pretend to be a window of the operating-system and request any sensitive data from the unsuspecting end-user.

Since applets have no `main()`-method, control structures and other sequentially ordered statements occur either in the applet's `init()`-, `start()`-, `stop()`-, `destroy()`- or `paint()`-methods OR in a method called by one of those methods.

Organize: Browser with Java Runtime Environment: To Do

Organize: Browser with Java Runtime Environment: To Do — Having the graphic on page 67 in mind, together with the previously presented graphic; write your applet's source-code, store it as a text-file with the name `className.java`, compile it into a `className.class`-file and possibly other class-files; now add a HTML-file that references the applet's class-file (name: *appletReferencing.html*); for this time keep all files in the same directory or folder (its absolute path in the Hierarchical File System (HFS) given by *absolutePath*). Start a Java-enabled browser and enter `file:// absolutePath-SlashSeparated/appletReferencing.html`; then, if all went well, the applet will be displayed inside the browser-window.

If, in the case of a client-server-setting, the applet's path of the file-system is remotely inaccessible, then put the HTML-file *and* the applet's class-file into the folder that contains the served files (For example, the Apache-server under Linux may use the absolute path `/usr/local/httpd/htdocs`).

Or, within the environment of the operating-system, use the terminal-command "`appletviewer uri`" from some command-line or terminal-window. If the terminal-window runs with a proper chosen current directory, then a new window will open, that contains the "Hello Web!"-message. If the terminal-command above returns a "file not readable error", then check whether the HTML-file exists in the present working directory, if not, change the directory to the directory that contains the HTML-file and the applet's byte-code.

The main entities to remember for running an applet are listed again below:

A text-file, post-qualified by `.java`, containing the declaration of a public class, which extends the class `java.applet.Applet`. The text-file (`c className.java`) and the class-declaration, both have the same name `c className`.

The terminal-command: “`javac c className.java`” compiles that source-code, producing the applet’s byte-code given inside a file named `c className.class`.

An HTML-file named `appletReferencing.html` has to reference the class `c className` within an HTML-tag `<applet>...</applet>`.

The terminal-command: “`appletviewer appletReferencing.html`” starts an applet runtime-environment together with the applet and produces an applet window on the screen.

(Remember that `c className` denotes the applet’s class-name as declared in the source-code. The byte-code of the applet is to be found in the file named “`c className.class`” and the source-code in the text-file named “`c className.java`”).

Applet Threading

Applet Threading — Methods of thread-instantiation have been presented beginning with page 206. A short and rather coarse but pointed way to give an applet a `run()`-method of its own can be seen here: (Thread-related keywords and -commands are written in *italics*!)

```
public class ThreadedMiniApplet extends java.applet.Applet implements java.lang.Runnable
{ public void init()
  { this.setBackground( java.awt.Color.lightGray ); new Thread(this).start(); }
  public void run(){ /* something to be done like in a main()-method */ }
}
```

Another threaded applet is given in the following source-code. – Observe especially the different uses of the applet’s and the thread’s `start()`-method. – The public and void method `showStatus( String message )` is one of the applet-specific methods. This method permits the programmer to place a string-value into the so-called status-bar of the on-screen-window of the current Java-enabled Internet-browser. If the applet runtime-environment furnishes no status-bar, then the call of the method `showStatus()` is ignored.

```
public class ThreadedApplet extends java.applet.Applet implements Runnable
{ private Thread t;
  public void paint( java.awt.Graphics g )
  { g.drawString( "Threaded Applet: Watch the status-bar!", 20, 20); }
  public void run()
  { int i = 0;
    while( t == Thread.currentThread() )
      // Above, why not simply true? To stop run() when t == null;
    { try{ Thread.sleep(1000); }
      catch( Exception e ){ e.printStackTrace(); } // ALL PURPOSE CATCH
      i = (byte)(++i % 4);
      switch(i)
      { case 0: showStatus("\t -"); break;
        case 1: showStatus("\t \"); break;
        case 2: showStatus("\t |"); break;
        case 3: showStatus("\t /"); break;
      }
    }
  }
  public void init()
  { this.setBackground( java.awt.Color.lightGray ); t = new Thread(this); }
  public void start(){ t.start(); }
  public void stop(){ t = null; }
  public void destroy(){ t = null; }
}
```

There have been used no new programming constructs, this is only a mixture of applet- and thread-related notions. In a standard appletviewer, the priority of threads inside applets can only range from `MIN_PRIORITY`, equal to one, to `NORM_PRIORITY + 1`, equal to six. The text-file in HTML-format, containing the triggers for running those threading-applets, is given below:

```
<! This is a comment in HTML (comments in Java use '/*' or '/*' ... '*/')
  (comments in Unix scripts begin with '#')
>

<! Name of file: ThreadedApplets.html >
<html><head><h3>Threaded Applets</h3></head><body>
```

```

Here two applets with threads are embedded into standard HTML text:
<applet code=ThreadedMiniApplet width=0 height=0 >
</applet>
Standard HTML text
<applet code=ThreadedApplet width=400 height=50 >
</applet>
Standard HTML text</body></html>

<! As can be seen above, invisible applets are not easily to produce:
... width=0 height=0 ... still produce windows within appletviewers,
some browsers may allow invisible applets.
>
<! Threads are stoppable by setting all its references 'null'
and letting the garbage-collector dispose of it.
>

<!
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/Applets > javac ThreadedApplet.java
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/Applets > javac ThreadedMiniApplet.java
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/Applets > appletviewer ThreadedApplets.html
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/CommunicatingComputers/Applets >
>

```



Applet Introspection

Applet Introspection — An applet (or associated code) can use various sources of information to get information about itself or about its context:

- Information may be furnished by the applet HTML-tags of the calling HTML-code. (The parameters name, codebase, archive, code, object, <param>-subtags are accessible from the applet's source-code by using methods like `getDocumentBase()`, `getCodeBase()`, `getParameter( String parameterName )`) Refer also to the HTML-file below and the one on page 614.
- Information can be furnished explicitly by the applet's programmer as a String-object. (`getAppletInfo(){ return "theInformationAsString"; }`)
- Information about the applet's overall runtime-environment can be got from the System- or Runtime-classes (around page 258) or from Locale-objects (on page 259); but only if security-manager, policy-files and protection-domains permit the access of related information.

The following HTML-text and Java source-code demonstrate the use of these informational resources:

```

<! This is a comment in HTML (comments in Java use '/*' or '/*' ... '*/')
  (comments in Unix scripts begin with '#')
>

<! Name of file: Introspection.html >
<html>
  <head><h3>Hello Introspective Applet</h3></head>
  <body>The following applet tells something about itself
  <! width and height in pixels of the on-screen-representation of applet >
  <! code="ClassName", means within current directory file-name: "ClassName".class >
    <applet width=500
      height=200

```

```

code=IntrospectiveApplet
name="This Applet-instance's name"
> <! code = has to take the name of the class that implements the applet
   (for global identity use names like "COM.abc.TrivialApplet.class") >

<! If necessary substitute "code" above by:
codebase="http://www.abc.com/folder1" >
<! This is the URL (Uniform Resource Locator) of the class-file.
   This tells the browser from where to retrieve the class-file.
   The default value is the directory of the HTML-file
   with the applet HTML-tag. >

<! Or, if necessary substitute "code" above by:
archive="someArchiveName.jar" >
<! Only if JAR- or ZIP-files contain compressed class-files or
   other resources. >

<! Or, if necessary insert above: object="serializedApplet.ser" >
<! This is the name of file containing a serialized version of applet
   also used instead of the "code=" attribute. >

<! If wanted insert above: name="This Applet-instance's name">
<! Another applet in same HTML-document can get the reference
   to this instance. >

<! Insert above: alt="Text to display if the applet-viewing environment
   recognizes the applet-HTML-tag, but cannot run the applet!" >

<param name="someStringParameterName" value="theParameterValueAsString" >
</param>
</applet> Standard HTML text
</body>
</html>

<!--
[localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway% javac IntrospectiveApplet.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway% appletviewer Introspective.html
^C
[localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway%
-->

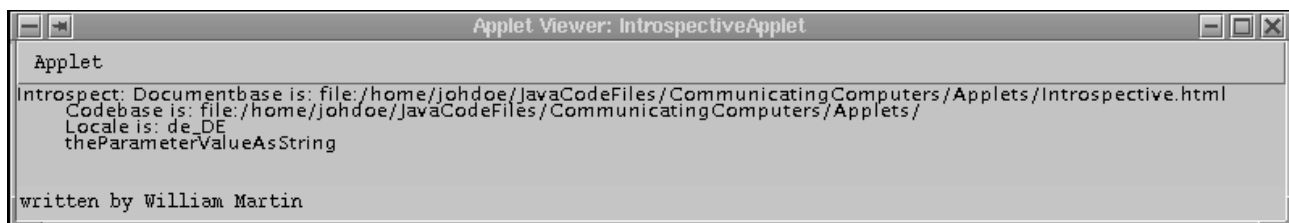
```

Here the applet's source-code for retrieving information about itself and its surroundings from a small set of functions:

```

public class IntrospectiveApplet extends java.applet.Applet
{ public String getAppletInfo(){ return "written by William Martin"; }
  public void init(){ this.setBackground( java.awt.Color.lightGray ); }
  public void paint( java.awt.Graphics g )
  { g.drawString("Introspect: Documentbase is: " + getDocumentBase().toString(), 0, 10 );
    g.drawString("Codebase is: " + getCodeBase().toString(), 30, 20 );
    g.drawString("Locale is: " + getLocale().toString(), 30, 30 );
    g.drawString( getParameter( "someStringParameterName" ), 30, 40 );
    showStatus( getAppletInfo() );
  } }

```



Applet Communication

Applet Communication — To a certain extent an applet can communicate with the server it comes from and with other applets that have been delivered by the same server. Use an HTML-document (named "Communication.html") of the following kind to see one applet stopping the thread of the other applet:

```

<!-- This is a comment in HTML (comments in Java use '/*' or '/*' ... '*/')
   (comments in Unix scripts begin with '#')
-->

```

```

<! Name of file: Communication.html >
<html><head><h3>Communicating Applets</h3></head><body>
Here two applets within one standard HTML text do communicate:
<applet code=ComReceiverApplet name="interAppletReference" width=400 height=100 >
</applet>
Standard HTML text
<applet code=ComSenderApplet name="anotherReference" width=450 height=100 >
</applet>
Standard HTML text</body></html>

<! The first applet has to reference the second applet by other means than by a given name
like name="anotherReference", because the second applet is instantiated after the
first applet, the first applet cannot easily get a reference of the second
applet of the applets.
If trying to reference the second applet before it has been instantiated,
this likely produces a java.lang.NullPointerException for the forward-reference.
When incorporating threading and a time-delay for forward-referencing, only then a
java.lang.NullPointerException MAY be avoidable!
>

<| [localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway% javac ComReceiverApplet.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway% javac ComSenderApplet.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway% appletviewer Communication.html
ComReceiverApplet[pane10,0,0,400x100,hidden,layout=java.awt.FlowLayout]
null
ComReceiverApplet[pane10,0,0,400x100,layout=java.awt.FlowLayout]
ComSenderApplet[pane11,0,0,450x100,hidden,layout=java.awt.FlowLayout]
^C (Ctrl-C)
[localhost:~/JavaCodeFiles/CommunicatingComputers/Applets] marycosway%
>

```

The source-code below uses an object described by the interface `java.applet.AppletContext` to get information about other applets that run in the same applet runtime-environment:

```

interface Receiver{ public void processString( Object sender, String s ); }

public class ComSenderApplet extends java.applet.Applet implements Receiver
{
    java.applet.Applet otherApplet, thisApplet;
    Receiver receiver;
    java.util.Enumeration e;
    public void init()
    { this.setBackground( java.awt.Color.lightGray );
      e = getAppletContext().getApplets();
      while( e.hasMoreElements() ){ System.out.println( e.nextElement() ); }
      thisApplet = this;
      otherApplet = getAppletContext().getApplet( "interAppletReference" );
      receiver = (Receiver)otherApplet; /////////////// CONTACTING RECEIVER
    }
    public void start() // before painted
    { new Thread
      ( new Runnable()
        { public void run()
          { for( int j = 0; j < 5; j++ )
            { try{ Thread.sleep(1000); }catch(Exception e){ System.out.println( e ); }
              receiver.processString( thisApplet, new String("The message no. " + j ) );
              // thisApplet not this (Thread)
              /////////////// CALLING RECEIVER
            } } }
        ).start();
    }
    int k = 0;
    public void paint( java.awt.Graphics g )
    { g.drawString( "This is the sender applet", 30, 10 );
      for( int l = 0; l < k; l++ )
        g.drawString( "SenderApplet has got a receipt for message " + l, 40, 40+10*l );
    }
    public void processString( Object sender, String s )
    { k++;
      repaint();
      this.showStatus(" SenderApplet received: " + s );
    } }
/*
The Receiver interface introduces a common type between two independent Java class-files.
Again the applets known to the current Runtime Environment are listed.
One of those applets, known to implement the interface Receiver is type-cast
into being of Receiver type.
The sending of the messages is threaded because otherwise the paint()-method
had to wait until all the sending was done, before painting the applet on-screen.
*/

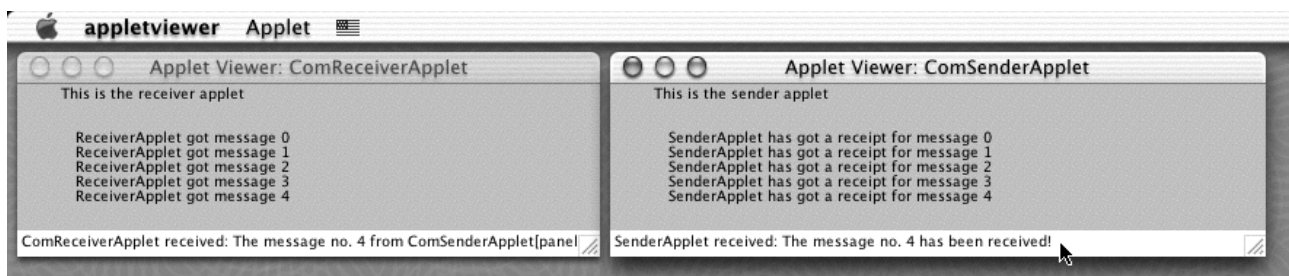
```


Observe that the “Receiver”-interface is shared by both applets. This Java-interface introduces a common ground of communication for the otherwise unrelated pieces of source-code. Such constellation already has been conceptually introduced on page 269. The other piece of source-code describes the receiver-applet:

```
interface Receiver{ public void processString( Object sender, String s ); }

public class ComReceiverApplet extends java.applet.Applet implements Receiver
{
    public void init()
    { this.setBackground( java.awt.Color.lightGray );
      java.util.Enumeration e = getAppletContext().getApplets();
      for( ; e.hasMoreElements(); ){ System.out.println( e.nextElement() ); }
    }
    int k = 0;
    public void paint( java.awt.Graphics g )
    { g.drawString( "This is the receiver applet", 30, 10 );
      for( int l = 0; l < k; l++ )
        g.drawString( "ReceiverApplet got message " + l, 40, 40+10*l );
    }
    public void processString( Object sender, String s )
    { k++;
      repaint();
      this.showStatus( " ComReceiverApplet received: " + s + " from " + sender );
      Receiver t = (Receiver)sender;
      t.processString( this, s + " has been received!" ); //////////// CALLBACK TO SENDER
    }
}
/*
The Receiver interface introduces a common type between two independent Java class-files.
The applets known to the current Runtime Environment are listed.
Then the paint()-method lists the number of messages received so far.
When another object calls the processString()-method,
its sends its reference and the source-code above assumes
that any sender is castable into being also a Receiver,
thus being able to send a receipt.
The applet's screen impression has to be controlled from out of the paint()-method.
Changing the applet's screen impression requires changing the paint()-method
and calling the repaint()-method to update the on-screen representation.
(This is the general rule for awt-screen-components in Java.)
*/
```

Applets can only communicate with the server they originated from. They may do this using an URL-object or by instantiating an appropriate Socket-object and using its factory-methods for producing input- and output-streams.



For an in-depth description of the `java.applet.Applet` class, see the chapter beginning on page 613.

Java: Servlets / (CGI)

Java: Servlets / (CGI) — The class `java.net.URL` introduced a mechanism to incorporate new application-protocol-handlers and new content-handlers during run-time. This mechanism make it *relatively* simple to include different application-protocols and different content-types on the client's side. Servlets can do the same for servers. — Applets empower servers to use computing power of their client, servlets empower clients to use computing power of servers¹⁶⁹. Client-applications using the `java.net.URL`-class and client-applications given in form of applets differ in some fundamental aspect: Applets are downloaded when needed to run, and therefore applets are usually given a tighter security-check.

¹⁶⁹In Unix and C/C++ context, some of the services Java-servlets provide, are offered by a software-infrastructure called the Common Gateway Interface (CGI).

Servlets and applets are presented using the following scheme: Concept, Infrastructure, Organization, Threading, Introspection, Communication. This makes it easier to compare concepts and their implementation.

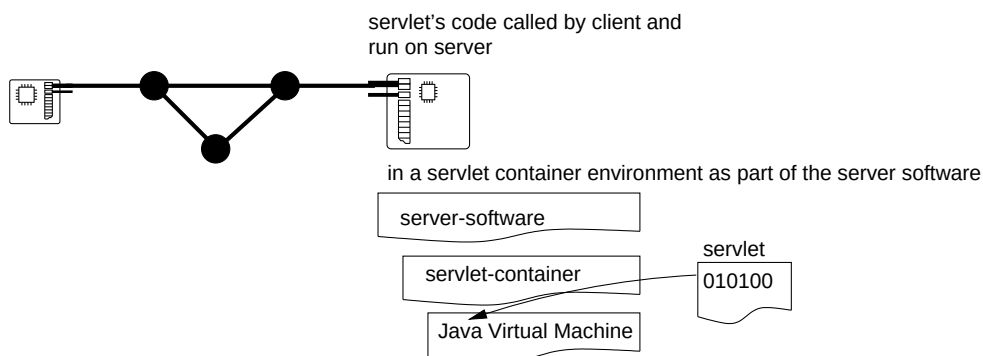
Servlet-Concept

- Servlets are PROGRAMS;
- to be RUN ON HTTP-SERVERS in an own container-environment on top of a Java Runtime Environment (JRE) and
- are CLIENT-INVOKED. (Applets are client-invoked programs, run on the client in an applet runtime-environment within (or on top of) a Java Runtime Environment.)
- A servlet-container is a part of the server-software (like the applet runtime-environment (or sandbox) is part of the HTML-browser software) and the dynamically loadable SERVLETS ACT LIKE MODULES FOR EXTENDING THE SERVER'S SERVICES.
- Servlets are RUN BY A DIRECT URL-REQUEST from the client or indirectly:
- Servlets are RUN WHEN THE CLIENT FETCHES A HTML-DOCUMENT WHICH INCLUDES `<SERVLET> ... </SERVLET>`-HTML-TAG PAIRS. Those servlet HTML-tag pairs are discovered by the server when parsing the to-be-served HTML-pages and cause the server-sided invoking of the indicated servlets.
- Servlets have no built-in screen display (contrary to applets with their `paint()`-method)

Servlet: Servlets are server-based programs, invoked by a single client and usually working exclusively for this single client. The client triggers the run of such a program by entering an Uniform Resource Identifier (URI) into an Internet-browser; directly or by having the server parse the HTML-pages for a `<servlet> ... </servlet>`-HTML-tag pair. Thus the client is serviced by a Java program running on a network-server. This is also called server-side computing.

Contrary to that, the server can send a Java program, an applet to the client which there, on the client's computer, runs in a safe environment and can communicate with the server. Applets are one implementation of the concept of client-side computing.

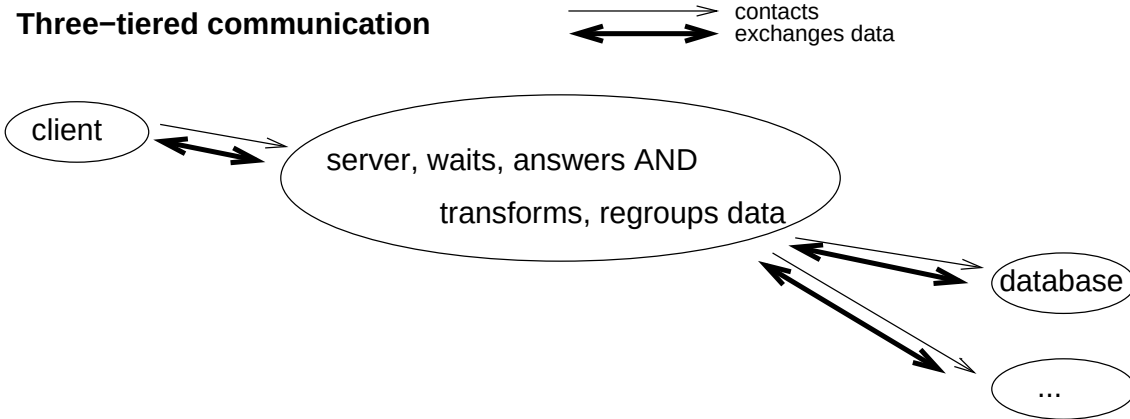
Web-applications are the collections of servlets and other files that furnish a complete server-side service to a client. That service may consist of the pure servlet-functionality, invoked by an Uniform Resource Identifier (URI), or the server offers HTML-pages with an embedded servlet.



Web-Application: 1. Any resource that is accessed using a Uniform Resource Identifier (URI). The resource may be found on the local computer-platform or on the some remote one in the network to which the local platform is connected.

2. In the context of (Java-)servlets: A collection of servlets and related files. The servlets share the same servlet-context and the entire resource can put to work by calling a single Uniform Resource Identifier (URI). The collection may be packed into an archive or be a collection of files referencable in a standard directory-structure.

Contrary to applets running on the client-computer, servlets run on the server-computer. This makes it possible for servlets to realize so-called three- or more-tiered information infrastructures:



In these cases the server does not only wait for requests to answer, in these cases the server-computer also becomes a resource-management system. The server-computer may consult databases and may construct its answer by consulting other (network-)resources. This three-tiered communication concept constitutes an integral part of the Java 2 Enterprise Edition (J2EE, not treated in this text).

Servlet-Infrastructure

- Servlet-software and its relation to the Standard Java Development Kit (JDK): The servlet-related software is NOT part of the Standard Java Development Kit (JDK). — The Tomcat servlet-extension of the Apache web-server, or for a minimalistic approach the Java Servlet Web Development Kit (JSWDK) containing a little server itself, may be used as a source of a servlet-container for developing servlets. Below has been indicated the installation and use of the JSWDK; that separate installation includes:

- The necessary pre-written (base-)classes `javax.servlet` and other Java-packages,
- a little platform-specific HTTP-server with an integrated servlet-container for test-running a servlet. This little server may be controllable from the native operating-system by issuing the “startserver”- and “stopserver”-commands in some terminal-window or command-line environment.
- Some example code.

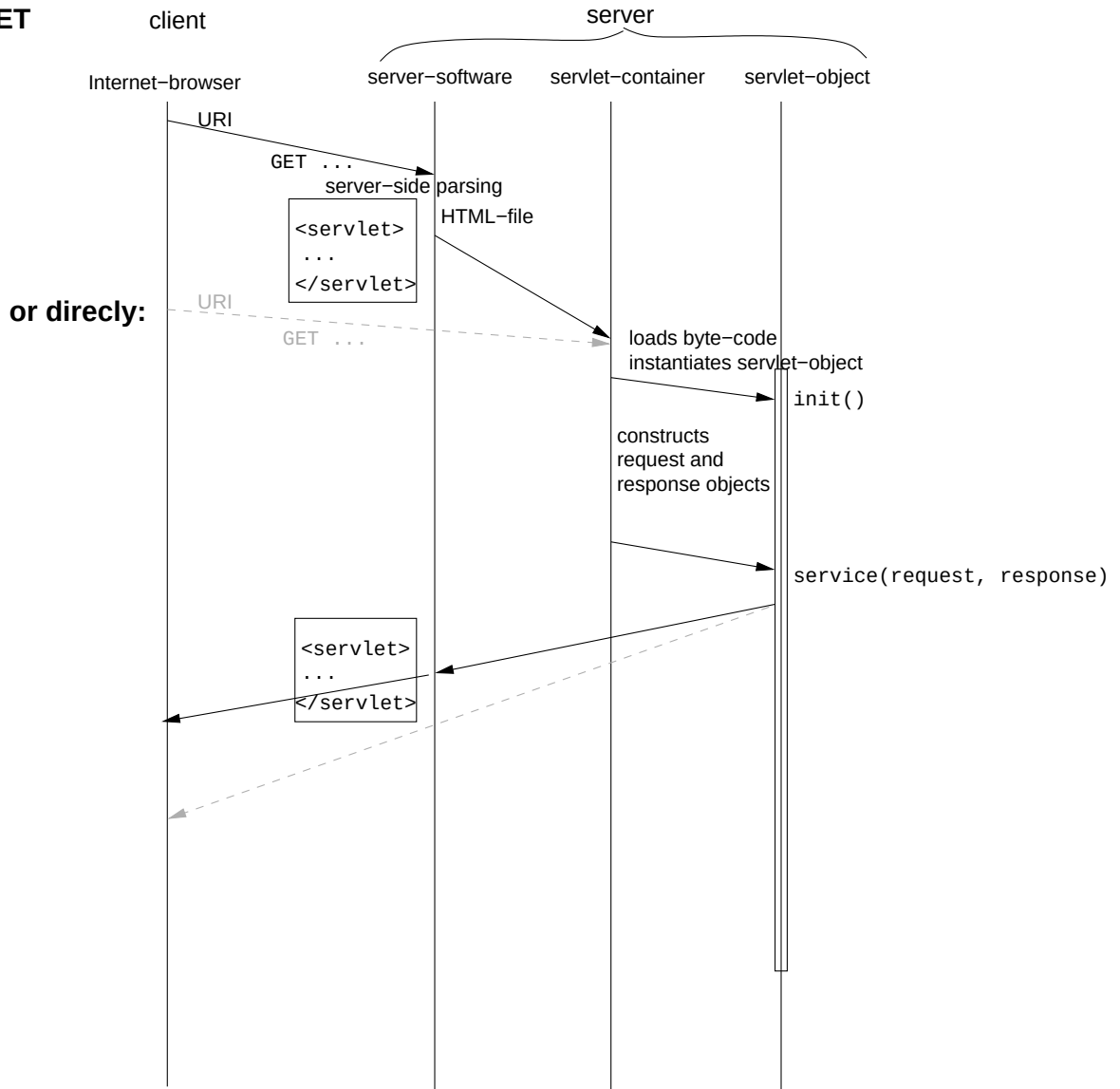
- THE DENOTATION “SERVLET” IS USED FOR either

servlet

1. the SOURCE-CODE of a class derived from the base-class `javax.servlet.GenericServlet`, or `javax.servlet.http.HttpServlet`,
2. the BYTE-CODE generated with the Java compiler (`javac`) or
3. an OBJECT, an instantiation of such a (servlet-)class.

- Security control on server-side should be included in the server-software.
- A servlet’s triggerable `service()`-method contains the primary application-programmer’s code¹⁷⁰. When receiving a client-request which also activates a servlet, then the servlet-container instantiates (the servlet-object unless already existent and) a `ServletRequest`- and a `ServletResponse`-object (a kind of dashboard-objects for outside communication, much like the `Thread`-objects for thread-code inside the `run()`-method) and finally calls the `service()`-method with these two object’s references as parameters.

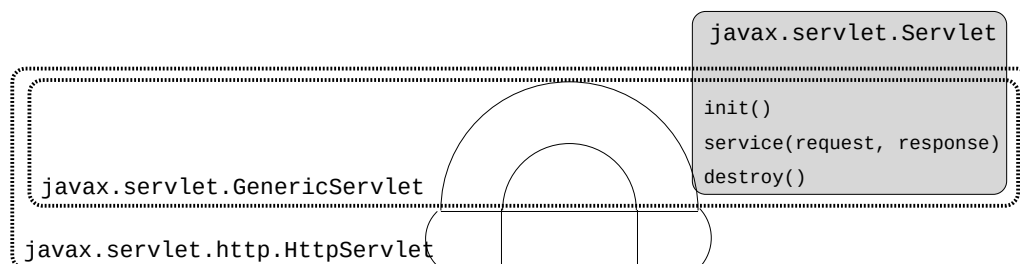
¹⁷⁰Application-programmer’s code is also contained in `main()`-methods of applications, `run()`-methods of objects that implement the `Runnable`-interface or are instances of the `Thread`-class, and the `init()`, `start()` or `stop()`-methods of applets.

SERVLET

A servlet-object can be used for multiple, different client-requests. Interferences of different client-request have to be handled by machine-synchronizing or atomizing the sensitive parts in the `service()`-method.

A Simple Servlet

A Simple Servlet — A servlet-class implements the `javax.servlet.Servlet`-interface. Base-classes of that interface-type are the `javax.servlet.GenericServlet` and the `javax.servlet.http.HttpServlet`-classes from which a servlet-container can derive servlet-objects.



A typical servlet may look like the one below.

```

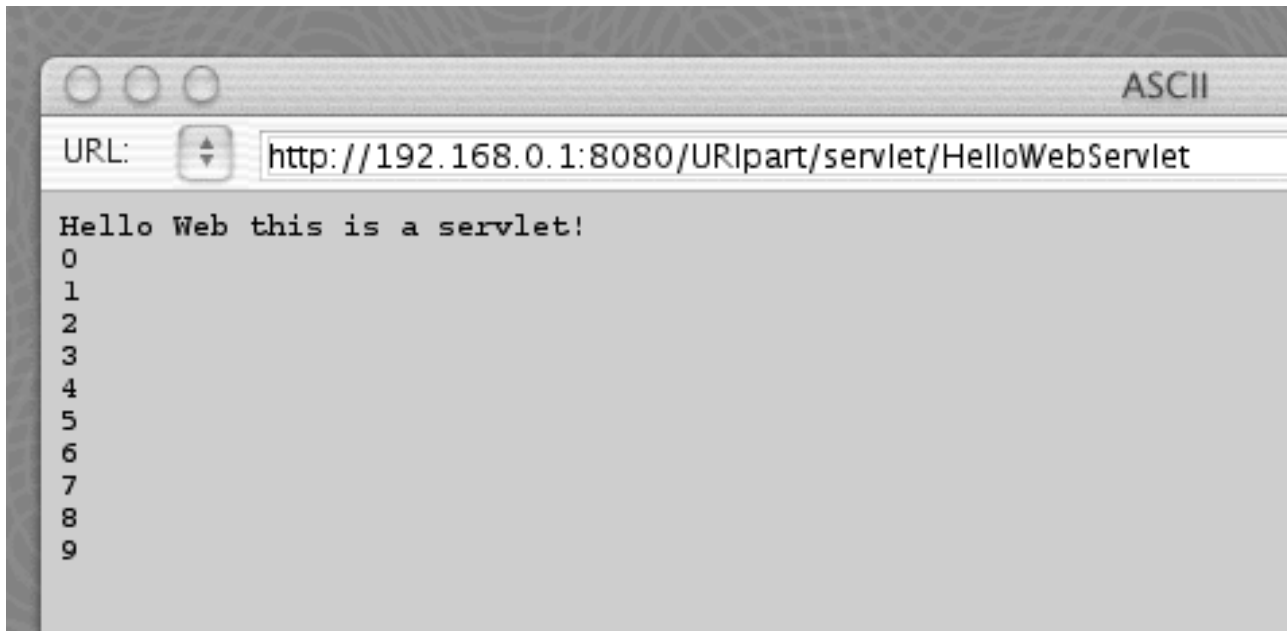
public class HelloWorldServlet extends javax.servlet.GenericServlet
{
    public void service(
        javax.servlet.ServletRequest request,

```

```

        javax.servlet.ServletResponse response
    ) throws java.io.IOException
{
    response.setContentType( "text/plain" );
    javax.servlet.ServletOutputStream out = response.getOutputStream();
    out.println( "Hello Web this is a servlet!" );
    for( int i = 0; i < 10; i++ )
    { out.println(i); out.flush();
      try{ Thread.sleep(1000); }
      catch(java.lang.InterruptedException e){ e.printStackTrace(); }
      finally{ out.close(); }
    } } }
/*
sue@linux:~/JavaCodeFiles/CommunicatingComputers/Servlets/WEB-INF/servlets > javac HelloWebServlet.java
sue@linux:~/JavaCodeFiles/CommunicatingComputers/Servlets/WEB-INF/servlets >
*/

```



The location of the servlet's class-file can be inferred from the directory-path displayed in the last comment of the source-code above. The Uniform Resource Identifier (URI), that triggers the run of the servlet can be seen in the image above, this URI has been declared as an alias for the directory-path in some configuration text-file ("URIpart" instead of a directory-path). So, at this point can be seen the difference between a URL that would need the directory-path to locate the resource and a URI that uses a simple name to locate the resource! Details of the related configuration process will be shown in the next sections.

Organize: Install Server with Servlet-Container: To Do



Organize: Install Server with Servlet-Container: To Do —The following gives an example of a standard installation for elementary development needs. Like any installation-hints in this text, such listings are meant to give an idea of the secondary work behind the pure programming task. Due to the diversity of possible contexts (hardware, operating-system, server-software and servlet-container software) and procedures (installation, configuration) a comprehensive description may fast become voluminous and outdated.

Development Kit Get the "Java Servlet Web Development Kit" (JSWDK) usually by downloading it from the Internet (<http://java.sun.com>) using an Internet-browser. The files will arrive as one file compressed into ZIP- or tar-format.

Dearchiving Use an archiving program (ack or archiver in Linux) to extract the contents into a pre-chosen destination-folder ("*path/DestinationFolder*").

Command-activation For command-activation in Linux see page 62. In Linux, this means essentially extending the PATH-variable with the path that points to the command-files:
 # PATH=\$PATH:*path/DestinationFolder*.

Command-Usage Or for using the commands in terminal-mode, change the directory to the folder that contains the commands:

```
# cd path/DestinationFolder
path/DestinationFolder# startserver
```

Class Incorporation Make the `javax.servlet`-package and additional pre-written classes part of the Java Runtime System (JRS) on the server-side by either adding the appropriate path of the Hierarchical File System to the system's `CLASSPATH`-variable or, if a package is given as a JAR-file, by putting it into the Java Runtime System's `ext`-folder.

View Furnished Examples Use the existent TCP/IP-network-connection or the "localhost"-mechanism on Unix-type operating-systems. Load a web-browser and invoke the examples by simply entering into the browser's command-line:

```
http://localhost:8080/
```

The TCP-port 8080 is used by the server of the Java Servlet Web Development Kit (JSWDK). The domain-name "localhost" or any other valid IP-address or DNS-domain-name has to identify a server with a Java servlet-engine.

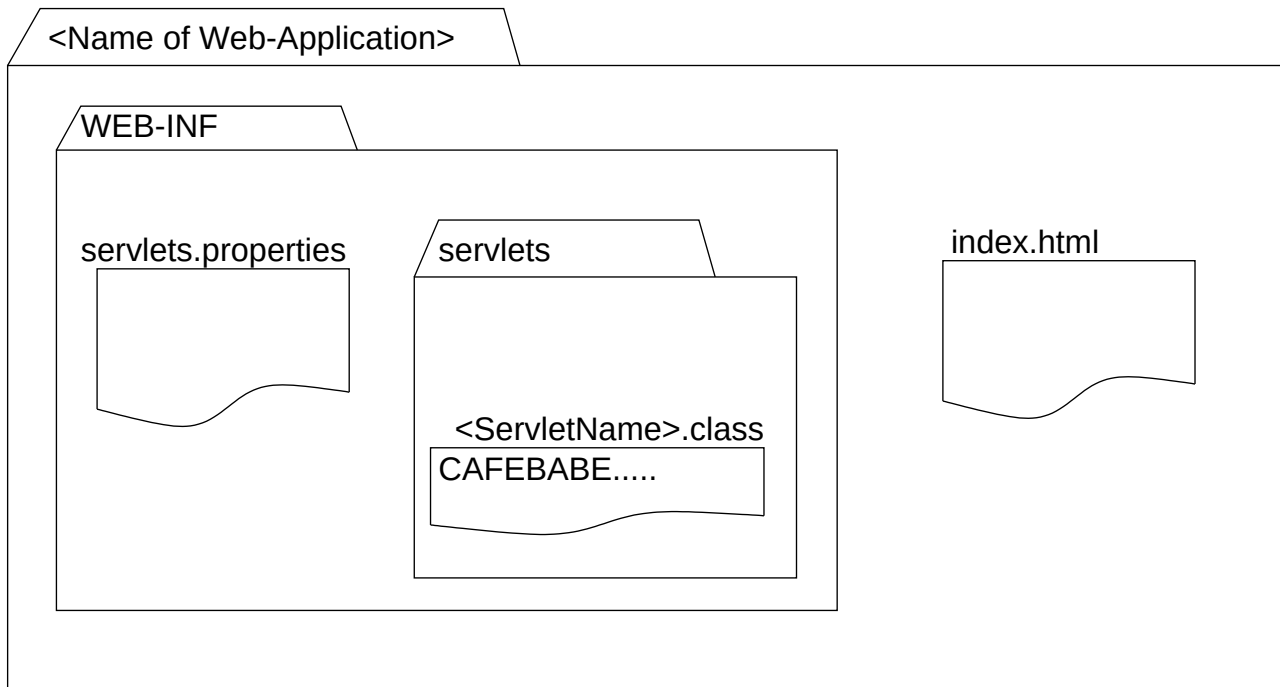
Another option means of course to use a regular HTTP-server together with a servlet-engine, for example the Apache-server with a Tomcat servlet extension (<http://jacarta.apache.org/tomcat/>).

Organize: Run a Servlet

- Imitating the examples, which are shipped with the Java Servlet Web Development Kit (JSWDK), identify or create the appropriate structure of folders for a web-application.
- Register the servlet with the servlet-container and determine whether the individual servlet is to be started together with the server or whether a servlet is loaded upon the client's request. In Linux and JSWDK that means entering location and further identificational data into text-files named `webserver.xml` or `servlets.properties`.
 - `webserver.xml` contains the JSWDK-server configuration and a list of the web-applications that are supposed to be known to the server. Here the application-programmer has to enter the directory-paths ("docbase") pointing to the folders that contain the web-applications. There can be added a sort of virtual path ("mapping" or "path") used later in the Uniform Resource Identifier as *uripath*-part of the URI.
 - `servlets.properties`¹⁷¹ contains the information about the servlets that are part of a web-application. The application-programmer can tag the servlet with a name and enter the name of the servlet's byte-code as well as a list of parameters and their values for the servlet:


```
somename.code = servlet's_name
somename.initparams=param1=stringValue1, param2=stringValue2 ...
```
- Put the servlet and associated files into the appropriate folders (The names of all related folders in the graphic below are crucial to a successful launch of a servlet):

¹⁷¹Sometimes also `servlet.properties` (without "s")!



- Activate a *server*-software together with its servlet-container.
In Linux with JSWDK choose the terminal to be in the directory that contains the relevant server-configuration-file `webserver.xml`. Then enter the “startserver”-command.
- Activate the *servlet* by starting a client (some Internet-browser) and entering a Uniform Resource Identifier (URI). That may be a standard URL of some HTML-page which calls the servlet OR a URI directly calling a servlet:
`http://host:port/uri-part/servlet/servletName`
OBSERVE, that the above represents a true URI! Why? Because the part *uri-part/servlet* does URI-vindication NOT represent a path inside the Hierarchical File System; this string constitutes nothing else but a simple arbitrarily chosen name! This allows the servlet to be relocated without a noticeable change of service to the end-user. An example showing that difference has been given on page 408.

And OBSERVE that unless the servlet-object is deinstantiated or the server is restarted, recompilation of the servlet’s source-code does not change the running servlet-object. That means the development of a servlet becomes a two-step-process: 1. Compilation and then 2. Restarting the server (if the development isn’t done in an appropriate Integrated Development Environment (IDE)).

Servlet Threading

Servlet Threading — A client’s call blocks the servlet, until the `service()`-method or its equivalent has finished. Threading makes it possible to keep the `service()`-method available for incoming calls. A practical example has been omitted.

Servlet Introspection

Servlet Introspection — What information can get the servlet-object about itself and the server it is running on? The classes `ServletRequest`, `GenericServlet`, `ServletConfig` and the interface `Servlet-Context` contain related methods. Some, like the method `GenericServlet.getServletInfo()` have to be implemented by the Java programmer, most others are pre-programmed:

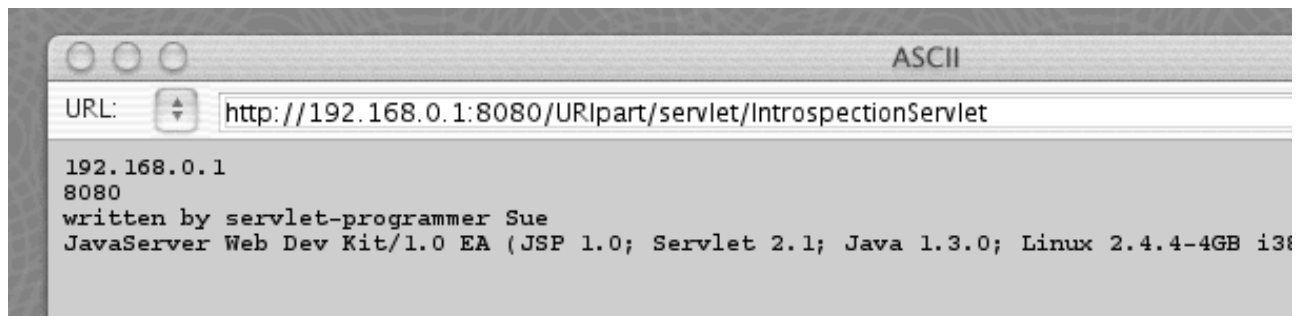
```

public class IntrospectionServlet extends javax.servlet.GenericServlet
{
    public String getServletInfo(){ return "written by servlet-programmer Sue"; }
    javax.servlet.ServletConfig sc;
    String s;
    public void init( javax.servlet.ServletConfig sConf )
        throws javax.servlet.ServletException
  
```

```

{ super.init( sConf );
  sc = sConf;
}
public void service(
    javax.servlet.ServletRequest request,
    javax.servlet.ServletResponse response
    ) throws java.io.IOException
{
    response.setContentType("text/plain");
    javax.servlet.ServletOutputStream out = response.getOutputStream();
    out.println( request.getServerName() );
    out.println( request.getServerPort() );
    out.println( getServletInfo() );
    out.println( sc.getServletContext().getServerInfo() );
    out.close();
} }
/*
sue@linux:~/JavaCodeFiles/CommunicatingComputers/Servlets/WEB-INF/servlets > javac IntrospectionServlet.java
sue@linux:~/JavaCodeFiles/CommunicatingComputers/Servlets/WEB-INF/servlets >
*/

```



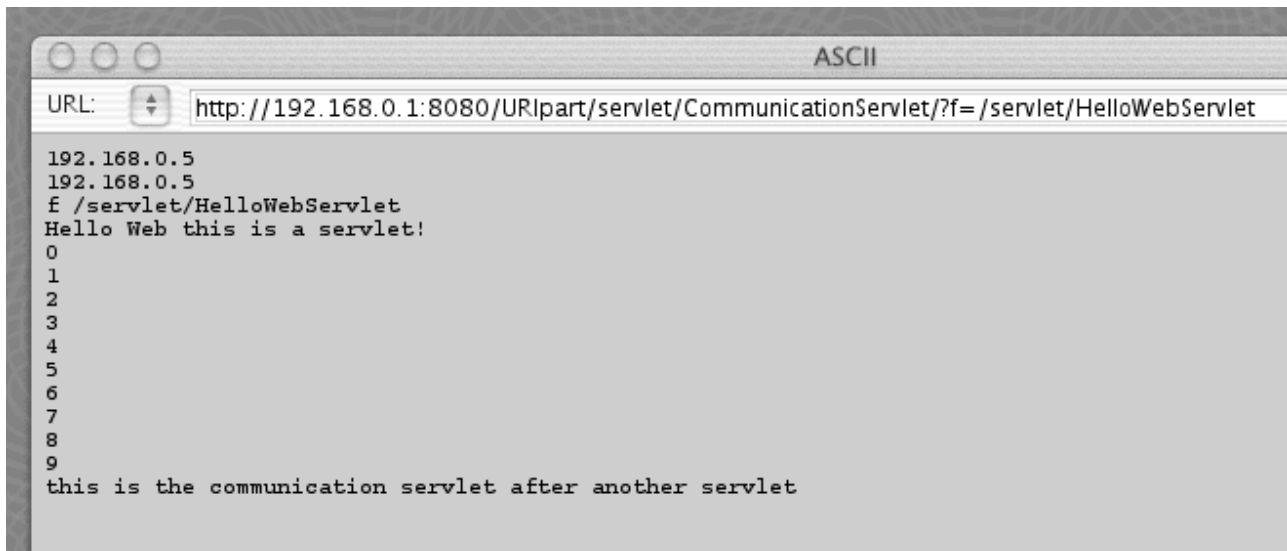
Servlet Communication

Servlet Communication — How does the servlet communicate with the outside world and especially with other servlets? The piece of source-code below demonstrates some of the possibilities: Using response- and request-associated streams enables basic communication with the client. This includes acquiring the network-address of the client. Inter-servlet communication becomes possible by using the `include()`-method.

```

public class CommunicationServlet extends javax.servlet.GenericServlet
{
    public void service( javax.servlet.ServletRequest request,
        javax.servlet.ServletResponse response) throws java.io.IOException
    {
        response.setContentType( "text/plain" );
        javax.servlet.ServletOutputStream out = response.getOutputStream();
        javax.servlet.ServletInputStream in = request.getInputStream();
        out.println( request.getRemoteAddr() );
        out.println( request.getRemoteHost() );
        java.util.Enumeration p = request.getParameterNames(); // analyzes last part of URI
        while( p.hasMoreElements() )
        { String key = (String)(p.nextElement());
          out.println( key + " " + request.getParameter( key ) );
        }
        String fileName = request.getParameter( "f" );
        // ... /servlet/CommunicationServlet?f=/servlet/HelloWebServlet
        try
        { getServletContext().getRequestDispatcher( fileName ).include( request, response );
        } catch( javax.servlet.ServletException x ){ x.printStackTrace(); }
        out.println( "this is the communication servlet after another servlet" );
        out.close();
    } }
/*
sue@linux:~/JavaCodeFiles/CommunicatingComputers/Servlets/WEB-INF/servlets > javac CommunicationServlet.java
sue@linux:~/JavaCodeFiles/CommunicatingComputers/Servlets/WEB-INF/servlets >
*/

```

HTTP-Servlet Extension

HTTP-Servlet Extension — HTTP-Servlets implement basic HTTP (Hypertext Transfer Protocol)-commands:

```
public class HelloHTTPServlet extends javax.servlet.http.HttpServlet
{
    public void doGet( javax.servlet.http.HttpServletRequest request,
                      javax.servlet.http.HttpServletResponse response
                      ) throws java.io.IOException
    {
        response.setContentType( "text/html" );
        java.io.PrintWriter out = response.getWriter();
        out.print( "<h1>Hello, Web this is a HTTP-Servlet!</h1>" );
        for( int i = 0; i < 10; i++ )
        { out.println(i);
          out.flush();
          try{ Thread.sleep(1000); }
          catch( java.lang.InterruptedException e ){ e.printStackTrace(); }
          finally{ out.close(); };
        }
    }
}
```

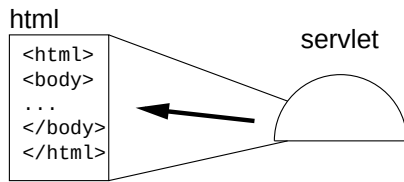
sue@linux:~/JavaCodeFiles/CommunicatingComputers/Servlets/WEB-INF/servlets > javac HelloHTTPServlet.java
sue@linux:~/JavaCodeFiles/CommunicatingComputers/Servlets/WEB-INF/servlets > */



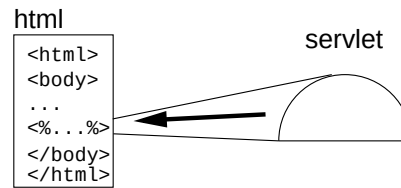
OUTLOOK: Though the HTTPProtocol is a connectionless protocol, HTTP-servlets have a built-in session-management by using Cookies or URL-rewriting. Sessions with Cookies means that data is stored by the server on the client's computer and URL-Rewriting means getting the IP-address of the client and use it as session- and temporary destination-identifier. Both methods are not covered in this text.

OUTLOOK: (Generating Web Pages Dynamically) As has been seen indicated in the last piece of Java source-code, servlets can generate HTML-files dynamically. In a practical context this proves not very useful: HTML-designers would have to know Java to develop page templates. And thus page-development would need Java compilation-processes, instead of using the faster and easier interpreted HTML.

JSP (Java Server Pages)



servlet generates entire HTML-document



servlets generates parts of a given HTML-document

So servlets are supplemented by another approach toward generating web-pages dynamically, HTML-files use additional tagging to include servlet generated dynamics:

- `<Servlet></Servlet>` HTML-tags, parsed for and recognized by the server, activate a servlet when a client downloads that HTML-document.
- `<% pieces of Java source-code in here %>` denote “Java Server Pages” (JSP) those HTML-pages with Java source-code which is automatically translated into servlets.

text-based
document-
format

(XML, XHTML) HTML (HyperText Markup Language) is a text-based document-format. That means HTML-format-commands (= HTML-tags) are written as plain-text-characters and positioned into the original plain text, with a simple text-editor. A browser knows about the HTML-tags and their meaning and therefore can convert the HTML-file into a formatted text. Files that represent a HTML-document are identified by their name having a suffix `.html` or `.htm`.

The eXtensible Markup Language XML is also a text-based document-format. But XML makes it possible to custom-define document-specific XML-tags.

Adobes Portable Document Format (PDF) is a non-text-based format. PDF-files can be identified by their suffix `.pdf` or the plain text beginning “%PDF-1.3”.

Servlets can be made part of a so-called web-application that resides as a collection of files on the server: These files may include static HTML or XHTML documents, dynamic request-dependent HTML or XHTML documents, so-called Java Server Pages (JSP) (a methodology that makes it possible to create HTML-pages dynamically, for example to show the contents of the specific client’s shopping basket in an Internet-store), Java-Beans, image-files and other data-files.

The servlet introduction above gave a faint hint what to expect from the programming context of the Java 2 Enterprise Edition (J2EE) (servlets, beans, three-tiered communication). For further illustrations see the online tutorial at <http://java.sun.com>.

Java: Remote Method Invocation (RMI) / (RPC)

Java: Remote Method Invocation (RMI) / (RPC) — Java’s datagram related classes, including the datagram socket, make it possible to send and receive bytes in individual IP-packets. Other sockets with their associated streams enable the programmer to communicate over application protocols like HTTP (Hypertext Transfer Protocol). This protocol together with the HTML-text format are also used to distribute Java applets and to offer servlet-functionalities network-wide.

Java’s Remote Method Invocation tries to answer the question: Is it possible to let Java objects communicate directly with each other over some network? Can Java language constructs, like objects, interact with each other if they are found in different Java Runtime Environments? So, what is the essence of Java’s Remote Method Invocation (RMI)?

A method of an object in one Java Runtime Environment can be invoked from another Java Runtime Environment¹⁷².

Java objects can be exchanged only as arguments of such a method-call, in serialized form, which effectively amounts to an object-copying process between Java Runtime Environments (JREs); the associated classes have to be loaded or loadable in advance. Generally the different Java Runtime Environments reside on different computers in a network. In Java, remote means within a different Java Runtime Environment (JRE); this may mean another JRE on the same platform or some JRE on another computer within the same network.

¹⁷²In Unix and C/C++ context the same mechanism is called Remote Procedure Call (RPC).

Due to the still developing status of RMI, the following paragraphs consider only the Version 1.1 of RMI. The Remote Method Invocation (RMI) is organized as a repository-based communication-scheme (see page 337) between computers in a TCP/IP-network. Communication is done unseen to the application-programmer over the User Datagram Protocol (UDP). Java's RMI infrastructure represents another approach towards *distributed computing*. Compared to the applet- and server-approaches, this one is closer to the level of the programming language. Nevertheless RMI does not realize the concept of *distributed programming* (page 423) as can be seen in the list below.

Type of object The type of a local object is given by class- or interface-name.

The type of a remote object is given by an interface which extends the `java.rmi.Remote`-interface, thus introducing a type without having to load an entire class or to define one!

Implementation or definition of type A type defined locally in a Java Runtime Environment (JRE) is given by a regular class- or interface-definition.

A type defined remotely is given on some RMI-server by a regular class- or interface-definition implementing the `java.rmi.Remote`-interface.

Creation/Instantiation of an object An object is created locally (within a JRE) by letting the `new`-keyword prefix a constructor-call or by calling a factory-method.

An RMI-client cannot trigger the creation of an object remotely. (The new Java 2 Remote Method Activation is supposed to make that possible.)

Access/Reference Locally, within a Java Runtime Environment (JRE) an identifier, also called reference, points to an object of class-type. If necessary, that reference may be qualified by a class- or package-name. If no references point to that object anymore, then the object is erased from the Random Access Memory (RAM) by the local JRE's garbage-collector.

Accessing an object remotely from another JRE, means using a pointer of interface-type which extends the `java.rmi.Remote`-interface. Unseen to the Java programmer a so-called stub-object organizes the access to the network and the communication with the remote object. Distributed garbage-collection becomes active if, within some time-period, there exist no references on stub-objects, then the garbage-collection is done by the local garbage-collectors.

Finalizing code Locally, pieces of code to be run before the object is erased from RAM, can be put into the `finalize()`-method for being run in a separate thread on garbage-collection. Remotely, the implemented `java.rmi.server.Unreferenced`-interface defines the `unreferenced()`-method to be run upon distributed garbage-collection.

Exceptions Locally any object of type `java.lang.Throwable` is handled by try-catch-clauses. Problems within remote objects are relayed back using message-objects of type `java.rmi.RemoteException`.

Organize: JRE-Extension: What is needed for a RMI-Connection?



Organize: JRE-Extension: What is needed for a RMI-Connection? — What resources are needed in advance to establish a functioning communication, by using the Remote Method Invocation (RMI) of Java? Generally the following resources are an integral part of any Java Development Kit or native platform and need no extra installation procedures:

(TCP-connection) A working TCP/IP-connection between two computers or an operating-system, being capable of establishing a TCP/IP-connection with itself¹⁷³.

(RMI-infrastructure) An RMI-compiler “`rmic`” and the RMI-registry “`rmiregistry`”.

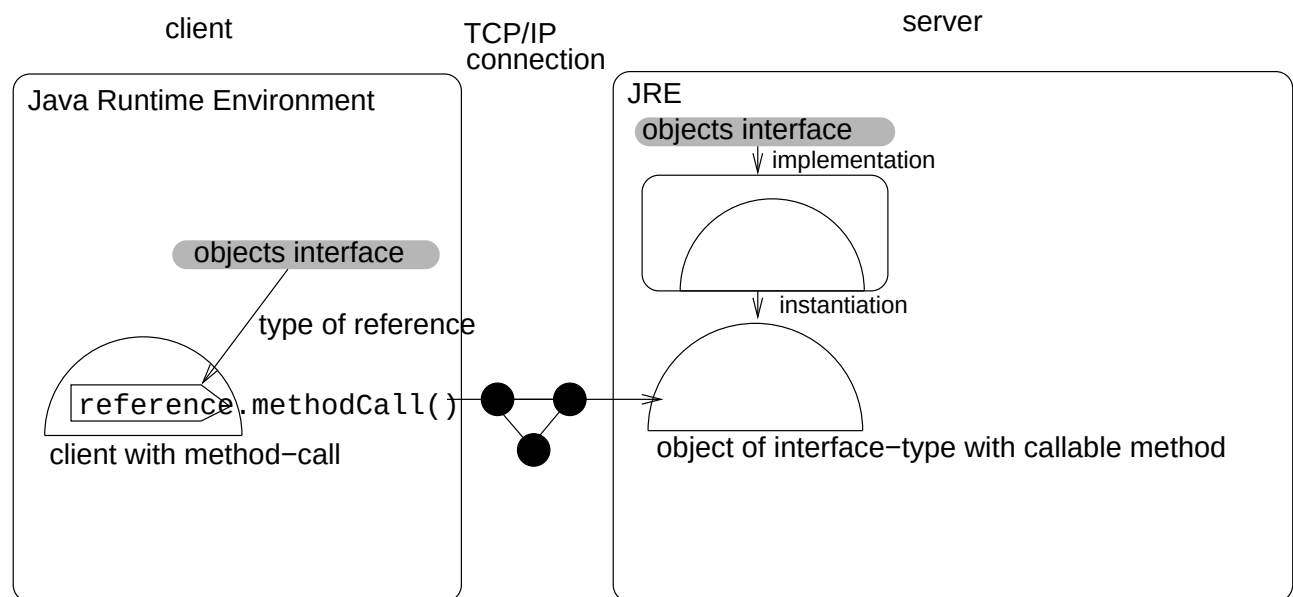
(Java Runtime Environment) A working Java Runtime Environment (JRE) is needed on both computers too.

(JDK) For writing and compiling the source-code a Java Development Kit (JDK) has to be found on one of the computers.

¹⁷³Some operating-systems can address themselves by using the IP-address `0.0.0.0` or the domain-name `localhost`.

A RMI-Connection

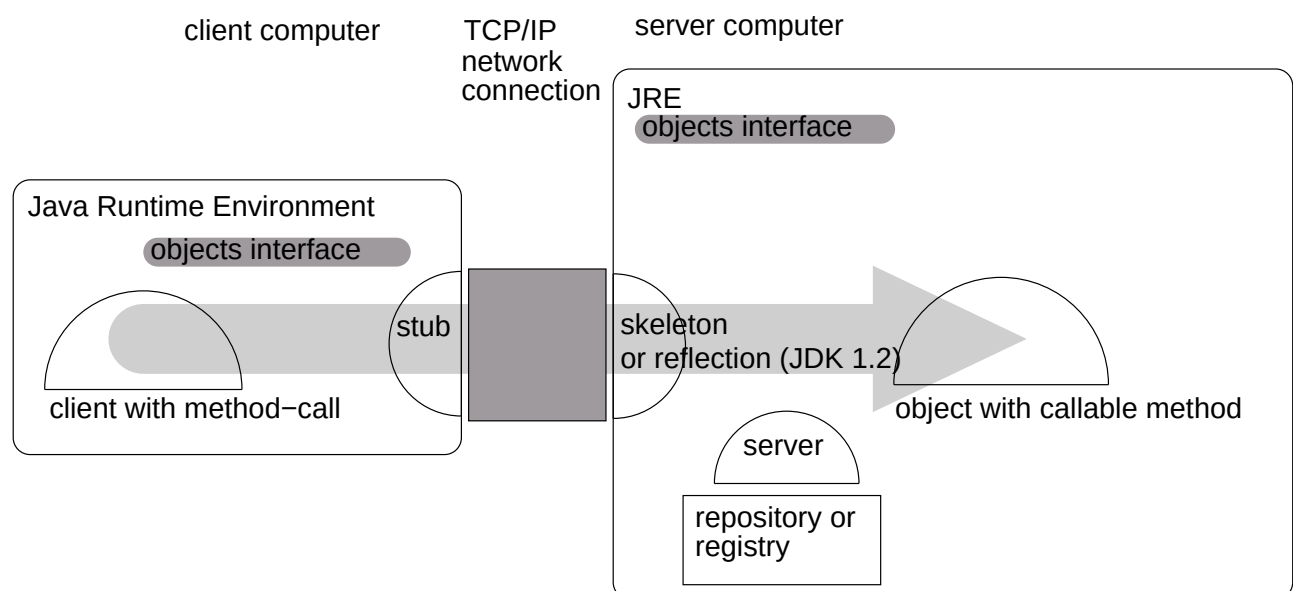
A RMI-Connection — Remote Method Invocation means that one Java object can call the method of another Java-object which exists in another Java Runtime Environment (JRE). The knowledge about the remote method is communicated by a Java-interface: This interface bears the signature of the method to be called and has to exist on both sides of the communication-path¹⁷⁴.



What has to be existent on the RMI-server-side and what has to be found on the RMI-client's side?

Server side: The **object** with the method to be remotely invoked; the **interface-class** containing the signature of the remotely callable method, implemented by the remote object's class, also a **skeleton-class** compiled from the object's class with the RMI compiler (rmic). And finally of course the **RMI-server-class**.

Client side: **RMI-client-object**, calling the remote-method, the same **interface-class** and the **stub-class** also generated with the RMI compiler (rmic).



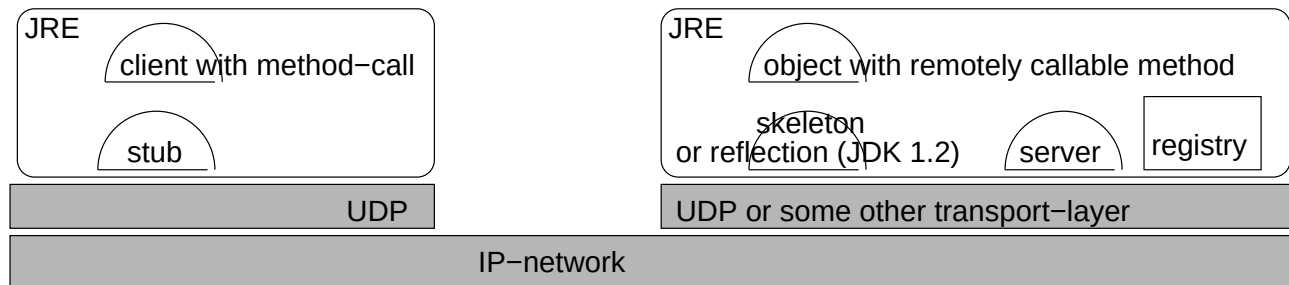
The mechanics¹⁷⁵ indicated by the graphic above and below should be unseen by the Java programmer! Only when considering the class-files of byte-code there do occur additional class-files with the names

¹⁷⁴Remember the use of an interface for communication in the applet's source-code beginning with page 404. And remember the introduction of this means of communication on page 269.

¹⁷⁵These mechanics follow a so-called proxy-design-pattern. There, a so-called proxy (stub in RMI) relays commands or data between different contexts.

including stub- (and skeleton-) denotations. Consider also the layer-diagram using stub, skeleton, UDP, IP, application, server and registry:

Remote Method Invocation (RMI) Layer Model



Both graphics above show how stub- and skeleton-pairs encapsulate the networking tasks and thus make the Java source-code independent of any concrete network.

Now, first the files with the source-code are shown below. There should be four basic files of source-code:

- The client-code, containing the call of the remote method, and
- the server-code,
- the code of the object to be accessed on the server, with the remotely callable method, and
- the interface with the signature of the remotely callable method.

Begin with the interface specifying the signature of the method(s) to become remotely callable:

```
public interface RMIClassInterface extends java.rmi.Remote
{
    public int method(int i, String s) throws java.rmi.RemoteException;
    // REMOTELY CALLABLE
}
```

Below, consider the code from with the actual object with the remotely callable method is to be derived:

```
public
class RMIClass extends java.rmi.server.UnicastRemoteObject implements RMIClassInterface
{
    // explicit constructor to declare exception
    public RMIClass() throws java.rmi.RemoteException
    { super(); }
    /////////////////////////////////////////////////// INTERFACE implementation
    public int method( int i, String s ) // REMOTELY CALLABLE
    { System.out.println( "Got String " + s ); // prints out a string on server
      return i*i; // returns a squared number
    }
}
```

The RMI-server instantiates the object which declares the remotely callable method. And registers the object with a name of its own in the RMI-registry using class-methods of the class `java.rmi.Naming` (final class, private constructor: Only static methods because the class is non-instantiable and non-extendable):

```
class RMIServer
{ public static void main( String unused[] )
{
    System.out.println(" RMIServer running ...");
    try
    { RMIClass theRMIObject = new RMIClass();
      System.out.println(" RMI-object instantiated ...");
      java.rmi.Naming.rebind( "theObject", theRMIObject );
      // Naming.rebind(): registers object with registry for remote use!
      System.out.println(" RMI-object rebound ...");
      //String [] entries = java.rmi.Naming.list( linux.local );
    }catch(java.lang.Exception e){ System.out.println(e); }
    System.out.println(" RMIServer left ");
  } // java.rmi.RemoteException, java.net.MalformedURLException
}
```

/* Input in FIRST TERMINAL-WINDOW:

```
[localhost:~/JavaCodeFiles/CommunicatingComputers/RMI] vincent% javac RMIClass.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/RMI] vincent% rmic RMIClass
[localhost:~/JavaCodeFiles/CommunicatingComputers/RMI] vincent% javac RMIServer.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/RMI] vincent% javac RMIClient.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/RMI] vincent% rmiregistry &
[1] 521
[localhost:~/JavaCodeFiles/CommunicatingComputers/RMI] vincent% java RMIServer
RMIServer running ...
RMI-object instantiated ...
RMI-object rebound ...
RMIServer left
Got String string sent away, Hello RMI
*/
```

And finally consider the RMI-client that can access the RMI-registry by using the class-methods of the class `java.rmi.Naming`. The interface is needed for casting the remote reference into a form which allows the remote method-call:

```
class RMIClient // needs declaration of RMIClass
{ public static void main( String unused[] )
{ try
{ System.out.println("Client running ...");
String [] remoteObjects = java.rmi.Naming.list( "linux.local" );
RMIClassInterface remoteReference = ////////// INTERFACE cast for communication
(RMIClassInterface)java.rmi.Naming.lookup( "rmi://localhost/theObject" );
// localhost or here 192.168.0.1
//// Naming.lookup(): Accessing the server's registry.
System.out.println(
remoteReference.method( 3, new String("string sent away, Hello RMI") ) );
}catch(java.lang.Exception e){ e.printStackTrace(); }
} }
/* Input in S E C O N D TERMINAL-WINDOW:
[localhost:~/JavaCodeFiles/CommunicatingComputers/RMI] jerome% javac RMIClient.java
[localhost:~/JavaCodeFiles/CommunicatingComputers/RMI] jerome% java RMIClient
Client running ...
9
[localhost:~/JavaCodeFiles/CommunicatingComputers/RMI] jerome%
*/
```

All these classes have to be compiled (`javac`) regularly, AND the `RMIClass` has to be additionally compiled with the RMI compiler (`rmic`); this gives four (`javac`-produced) plus two (`rmic`-produced) class-files, including source-code, ten files.

	RMIClassInterface.java		
javac	RMIClass.java	→	RMIClass.class
		→	RMIClassInterface.class
rmic	<i>RMIClass</i>	→	RMIClass_Skel.class
		→	RMIClass_Stub.class
javac	RMIServer.java	→	RMIServer.class
javac	RMIClient.java	→	RMIClient.class

Then the files may be distributed properly onto the client- and server-computers, using for example an FTP-connection; or tested first on the same computer if an addressing as a “localhost” (Unix, Linux) is possible. As indicated in the graphic above, the client-side always needs the interface-class of the object, its stub, the repository-name of the object and the server-address.

How is the Remote Method Invocation Service activated?

Both computers Establish a TCP/IP connection (TCP/IP control-panel on Mac OS Version 9.x), if not automatically configured by the platform (Linux, Mac OS X).

Change the current directory of the terminal-window to a directory where the presented RMI-classes are to be found. For example use a command like “`cd JavaCodeFiles/CommunicatingComputers/rmi`” within a terminal-window; or within Apple’s JBindery-tool choose the appropriate folder.

Server: First run the RMI-repository , a thread that takes RMI-protocol requests, by entering “`rmiregistry &`” and then enter “`java RMIServer`”, to start the server.

Client: Finally run the RMI-client on the client-computer by entering “`java RMIClient`”.

Then the server-output should read “Hello RMI” and the client-output should read “9”.

What more?

What more? — As can be seen above, arguments of primitive-type are passed by-value, and an argument of the RMI method-call can be a reference to a (serializable) object. Assuming that, like the Java-interface above, the class of the argument is also available and loaded in the server's Java Runtime Environment, this makes it possible to have a sort of object-migration. A *sort* of object-migration because it is *actually an act copying of an object* — the identity (see the footnote on page 131) of the object is LOST. And the migration-emulation of an object derived from a non-standard class has to be accompanied by a class-loading-process.

Method-call arguments Locally, within a Java Runtime Environment, arguments of primitive-type are passed by-value, this means: a copy of the value is made. Arguments of reference-type are passed by-reference, this means: Only a new pointer to the original object is declared!

Remotely, arguments of primitive-type are treated the same way as done locally, they are passed by-value. An argument of reference-type causes the referenced object to be serialized and transferred to the remote Java Runtime Environment. This gives an equivalent of a call by-value with some added complexities: First, the argument has to be a serializable object, or rather a serializable object-graph (the transfer over the network may become resource-consuming!). Second, the classes of that object-graph have to be readily loadable on the destination computer. Third, the reinstantiated object-graph on the destination computer represents a COPY of the original object, having a different identity (`java.lang.Object.hashCode()`). The same considerations have to be given any object-graphs returned by a remote method.

Already from this can be seen that RMI is a very rudimentary tool for getting a *very* first impression of what is called distributed programming. This RMI-mechanism works somewhat on the same level as the general event-mechanism in Java (page 286). To get a really application-oriented functionality, the Java programmer still has to do much work. In some generality that work is being done in software-products like “Voyager” and “Dejay” (<http://www.dejay.org>).

By now, the new Version 1.2 of the “Java Remote Method Protocol” (JRMP) may have emerged and may have been superseded by the Internet Inter Object Management Group (IIOP) for communication. There, the skeleton-class-file is to be substituted by Java's reflection mechanism; so-called “dormant objects” may be remotely instantiated; and method-call-multicasting means accepting the first method-call-answer, supposedly speeding up the remote reaction.

Outlook: Java and Networking

Outlook: Java and Networking — Multicasting is implemented in the `java.net.MulticastSocket` and makes it possible to send one IP-packet to multiple IP-addresses.

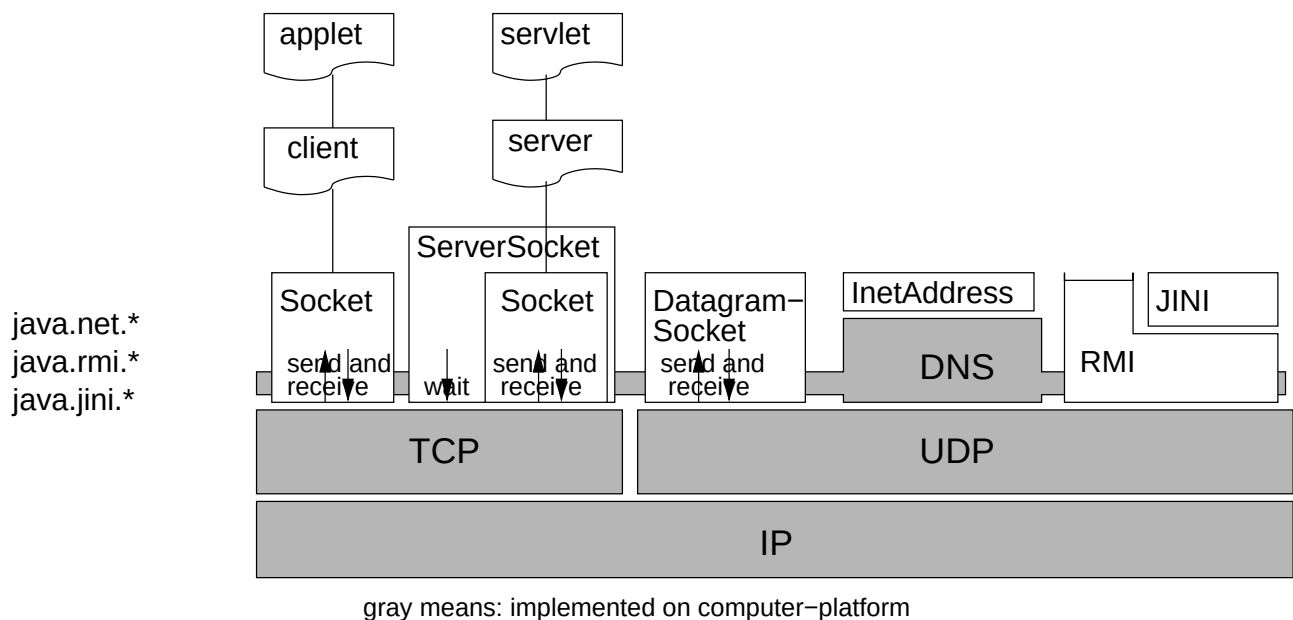
Java Network Naming Interface (JNDI)  gives the Java Runtime Environment (JRE) access to file-systems that need not be accessible by the underlying, native operating-system.

What is Jini?  A blackboard for application-services (printer etc.). How is that blackboard, also called Lookup Service, found? This is found by sending a datagram to a registry given by an IP-address and waiting for an answer. Since IP-multicast-packets are rejected by routers, Jini works only in the boundaries of routers. Jini can be seen as a registry for network-resources on the level of application-programming.

The last two items clearly work with new kinds of name-spaces!

Summary: Java Communication

Summary: Java Communication — This section essentially comments the graphic below: The standard networking Java packages (`java.net`, ...) assume to be working with an underlying IP-network and communicate with that network over the transport-protocols TCP and UDP. A direct low-level access to individual packets of the IP-network give instances of the class-type `java.net.DatagramSocket`: Objects of this class make it possible to inject single IP-packets into the IP-network and to extract them likewise! The Java class `java.net.InetAddress` resolves IP-addresses by using an existent Domain Name Service of the IP-network. Java's RMI-infrastructure presents a first idea in the direction of distributed programming. Closer to applications are Java's socket-classes that make it possible to communicate over the TCP-protocol, with input- and output-streams, thereby worrying only about application-formats of the byte-streams. Nevertheless, even the application-level furnishes structures like applets and servlets, which introduce a range of options, large enough to be only superficially covered here.



sun.com.jini.*
net.jini.*

Concept: Distributed Computing and Programming

Concept: Distributed Computing and Programming — The programming-language Java and the packages `java.net`, `java.io`, `java.applet`, `javax.servlet` and `java.rmi` make it possible to let applicational software (classes and objects) communicate over the network, and by transferring data over the network let it work as a distributed application. This is called distributed computing.

Contrary to that, **distributed programming** may mean that the programming language allows to freely migrate classes and objects over the network and to invoke their methods WITHOUT BOTHERING WHERE THOSE OBJECTS ARE WITHIN THE NETWORK. In a **network-oriented-programming-language**, the organization of network-connections would already be largely furnished on the language-level. To some extent, with some tricks and considerable effort that is also possible using the resources that are furnished by the Java system. The transfer of class-files has to be arranged, a class-loader on the destination-computer has to load the class into the runtime-environment, if the overhead of the Remote Method Invocation (RMI)-mechanism is to be discarded, then a communication by sending serializable message-objects has to be prepared. But Java itself does not make this kind of programming exceptionally easy! Even on the level of operating-systems, the concept of a well-managed, free and safe sharing of resources with other computers in a network, seems to be a highly non-trivial one.

distributed
computing
distributed
programming

Another aspect surfaces at this point: Unless the network-connection itself does not furnish a guaranteed Quality of Service (QoS), the applications themselves have to at least diagnose the quality of the network-connection!

Thus, due to various reasons, Java leaves much of the standard organizational work in handling network-connections to the application-programmer or to additionally furnished software (packages).

Sockets, RMI, and other methods can be seen as hardware or platform-related organizational necessities, that do very little to ease solving problems on the applicational level. An easier way of handling, exchanging and working with remote objects may be conceivable; namely in such a way that the network virtually becomes part of the computer also on the level of the Java programming-language. A Java language-extension of such kind, that uses object-references to access remote objects or to activate them from an object-database has been attempted in the Dejay-project. Migration of objects and concurrent programming (through calling non-blocking remote methods) are possible. For further information search the Internet for the notion “distributed programming” or for example consult the URL <http://www.dejay.org>.

Part VI.

Java Runtime Environment and Security

Java Runtime Environment and Security — WHAT HAS BEEN SHOWN SO FAR. The first part of this text introduced simple computer-models on the levels of hardware, operating-systems and the Java language. The following parts showed the basic means the Java programming-language furnishes to write programs:

- piling statement on statement, using `if-else`-clauses and loops
- proceduralizing the code by dividing it into “named blocks” or rather “methods”
- the “class/object-structure” of bringing together data and the code (methods) for manipulating the data. This implements the concept of programmer-definable types, where a class represents the type and the object represents the variable. The idea of working with class/object-structures also allows to relate classes by a “class-extension”-process which produces indexed hierarchies of code. In practical programming, such a hierarchical structure proves to be of more help than a simple aggregation of related code.
- structures of source-code allowing parts of the code to be operated on (quasi-)parallel, which is called “multithreading”
- a code-library and name-space is given by aggregations of code, in Java called “packaging”, in general context also named “modularization”

The transfer of information between objects by the means of a simple method-call can be considerably changed when using the message passing concept or parts of the Java-Beans conventions. Thus the specifics of the communication-process can be moved into the design of the messages and message-channels (=message-infrastructure). Apart from the greater flexibility and the clearer programming-interface for communication, the programming code can be better focused onto the actual problem to be solved. (Method-calls still govern a message-oriented communication, but these message-calls become infrastructure-related, instead of concerning the specific applicational problem.)

The communication with other Runtime Environments of Java-origin or other environments is dominated by network-considerations on TCP/IP-basis and collections of special software-concepts (RMI, JINI, JNDI).

NOW AND HERE the specification of the Java programming-language and of the Java Runtime System (JRS) are revisited, primarily to consider class-loading and security matters. Then the notion of security is refined and the text introduces security-related Java language-constructs like the singleton-creating class `java.lang.SecurityManager`, the `java.security.AccessController` and some other resources of the `java.security`-package.

Concept/Java: Scope, Access-Areas, Name-Spaces

Concept/Java: Scope, Access-Areas, Name-Spaces — Objects, classes, interfaces, methods, host-computers are referred to by the use of names (identifiers). These identifiers provide access to their referenced entity only in certain surroundings: Recall some known relations of names and their ranges of validity, respectively their areas of access:

(Access-area—name)

(Method—local variable) The name of a local variable of a method is only known to the code within the method. (See page 121)

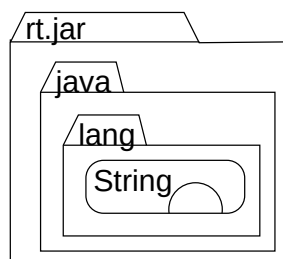
(Class/derived objects—private members of a class) Private members of a class are only accessible inside that class or (across) derived objects. They are not accessible from sub-classes, their instances or from elsewhere or by any other means within the Java Runtime Environment (JRE). (See page 175)

(Package—members of a class without access-modifiers) A class-member without an access-modifier is only accessible from classes or objects of the same package – if necessary, qualified by an identifier. (See page 175)

(Package and Sub-Classes—protected members of a class) Protected members of a class are accessible from classes or objects of the same package – if necessary, qualified by an identifier – AND from sub-classes outside the package. (See page 175)

(JRE—public members of a class) Public members are known throughout the entire Java Runtime Environment, they possibly have to be qualified by their package- or class-name to become accessible.

(JRE—package member) Packages, custom-written or pre-written, like the `java.io`-package (contains Input/Output-related code), which are given as a single file and put into a folder referenced by the CLASSPATH-variable or put into the ext-folder of the Java Runtime System, become accessible to the current Java Runtime Environment (JRE). For example, the package `java.io`, allows the source-code to reference package-members by using their fully-qualified class- or interface-name like `java.io.InputStream`, or source-code beginning with `import java.io.*` may reference package-members simply by using the identifier, here `InputStream`. As an exception to that rule, all the interfaces or classes of the `java.lang`-package, like `java.lang.String` are known to the Java Runtime Environment without explicitly using the `java.lang` qualifier. (See page 244)



(Internet—host's IP-address) An IP-address is a sequence of numbers. An IP-address is valid throughout the IP-network in which it has been defined. Each assigned IP-address identifies exactly one host-computer inside the given IP-network. One host-computer may be identifiable by several different IP-addresses, but one IP-address should never be assigned to two or more host-computers. For an end-user, IP-addresses may occur within Uniform Resource Locators (URLs). (See page 359)

(Internet with DNS—domain-names) A domain-name is given by a sequence of characters, usually well memorizable by any end-user. Only in networks with a working Domain Name Service (DNS)

are to be found valid domain-names. A valid domain-name identifies a computer in a network. Since requests, which use the same domain-name, may be re-routed by the DNS to different hosts, domain-names are not always host-names. End-users and application-programmers access network-environments usually with Uniform Resource Locators (URLs), containing a domain-name. All this applies to any IP-based network; thus especially the Internet. (See page 362)

(File-System—file-name) Any file in the Hierarchical File System (HFS) of an operating-system is known to that operating-system. Usually such a file has to be prefixed by an absolute directory-path to be accessible. With JNDI (Java Naming and Directory Interface) other file-systems and their files can be made accessible by the current Java Runtime Environment (JRE). (See page 46)

(network—name of resource) The JINI and RMI (Remote Method Invocation) technologies widen the name-space of a Java Runtime Environment. RMI by furnishing an interface that enables the call of remote methods, JINI by furnishing a bulletin-board on some third machine that lists available services.

(current directory—program-name) Consider a file containing Java byte-code carrying the name *nameOfByteCode.class*. Let this file be stored in a Hierarchical File System (HFS) of the Java Runtime System's platform. Let the path to that file denote the current directory, seen from a terminal-window of the operating-system. Entering `java nameOfByteCode` into that terminal or a command-line of the operating-system, starts the Java Runtime Environment and loads or runs (provided it has a `main()`-method) that piece of byte-code.

(CLASSPATH-referenced class-name) Consider two files containing Java byte-code carrying the names *nameOfByteCode.class* and *referencingNameOfByteCode.class*. The second may refer the former in some block of source-code. Let both files be stored in a Hierarchical File System (HFS) of the Java Runtime System's platform. Let the file *nameOfByteCode.class* exist inside a folder denoted by the CLASSPATH-variable or let it be located inside the current directory, seen from a terminal-window of the operating-system. Entering `java referencingNameOfByteCode` into that terminal or a command-line of the operating-system, starts the Java Runtime Environment and loads or runs (provided it has a `main()`-method) both pieces of byte-code.

(JRE/Internet—applet) Files of byte-code, derived from the `java.applet.Applet`-class may be known to the client's Java Runtime Environment (JRE). But an applet's byte-code can be only run in an applet runtime-environment! (Exceptions, see page 619, are Applet-classes with a `main()`-method!). The applet runtime-environment (also called applet-sandbox or applet-container) usually restricts the freedom of the applet on the client's platform. This applet's runtime-environment is activated either by entering the command "`appletviewer nameOfHTMLfileWithAppletTags`" into a command-line of the operating-system. Or by entering the Uniform Resource Identifier (URI) of an applet-referencing HTML-page into the command-line of a Java enabled browser.

(Internet—servlet) Files of byte-code, derived from the `javax.servlet.Servlet`-class may be known to the Java Runtime Environment but can be only run in the servlet-container-environment. These usually are activated by the client's use of a Uniform Resource Identifier (URI).

Most examples above featured the structure "The name ... is accessible from ...". This relation hints at what is described by the word "access-area". The environment, in which the unqualified identifier is valid, is called scope of the identifier. The environment in which the identifier can be used as part of a more refined access-procedure (package-qualifier, `import`-statement, URL) is called access-area. A set of identifiers usually for some given access-area is called name-space (the role of an individual identifier is

not to be stressed)¹⁷⁶. So far in the text this icon  was used to mark areas where the notion of name-space became relevant.

scope
access-area

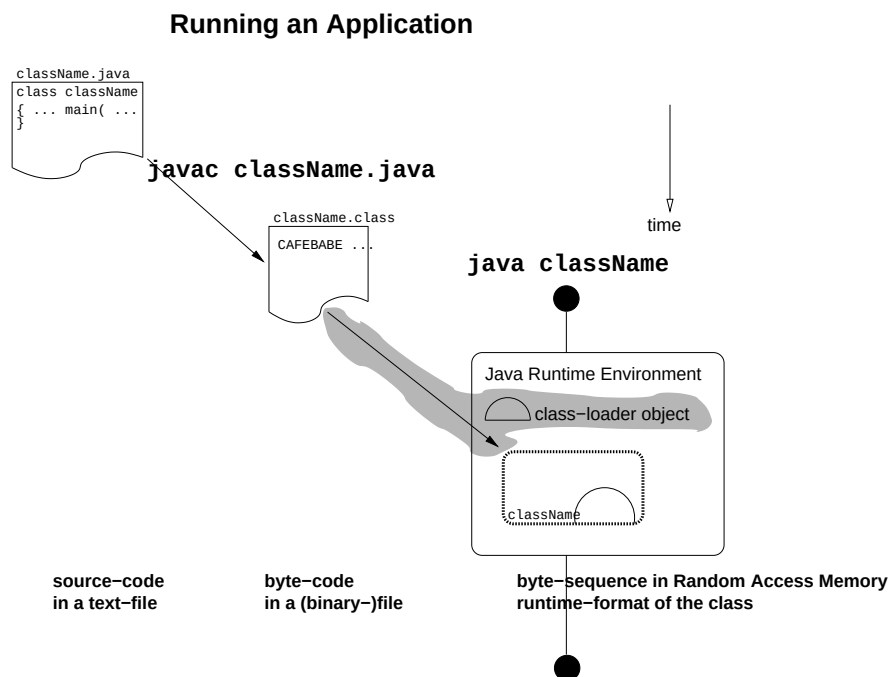
name-space


¹⁷⁶Scoping has been treated first on page 40, access-areas have been treated in a grayed rectangle on page 172, and the idea of a name-space has been exposed on page 123.

Java: Runtime Environment

Java: Runtime Environment — A Java Runtime Environment (JRE) is a platform-specific piece of software for running the portable Java code. A Java Runtime Environment (JRE) represents a virtual computer-platform for running the portable Java byte-code. A Java Runtime Environment (JRE) is given by software, partially native software, running on the native platform and translating the Java byte-code to platform-specific machine-code. “Java Runtime System (JRS)” denotes the files, often together with other resources, used to establish a Java Runtime Environment; but often these denotations are used interchangeably. At this point remember the definitions of the notions “Java Runtime Environment” and “Java Runtime System” on page 55.

Compiling Java source-code and running Java byte-code are two basic procedures in the development-process of Java programs. Recall that a text-file (*name.java*) of Java source-code lets the Java compiler (activated by the command “`javac name.java`” in the terminal-window) generate a file of byte-code (*name.class*, a so-called class-file) which is automatically stored in the Hierarchical File System (HFS) of the computer-platform¹⁷⁷. Then the program is ready to be run:



Start of the Java Runtime Environment (JRE):

java name has to be typed into the command-line or terminal-window or the equivalent in non-command-line operating-systems (Mac OS Version 9.x and below). The file of byte-code with the name *name.class* has to be in the current directory, then the Java Runtime Environment runs the byte-code. (Technicalities have been introduced beginning with page 59.)

appletviewer HTMLfileName.html has to be typed into a terminal-window. Or the Uniform Resource Identifier of such a HTML-file is entered into the command-line of a Java-enabled HTML-browser. The text-file *HTMLfileName.html* has to contain an `<applet>...</applet>`-HTML-tag pair with a reference to an applet's file of byte-code. Then the appletviewer- or

¹⁷⁷An error of the kind “command not found” signals a missing or improperly installed Java Development Kit. An error of the kind “Can't read: *name.java*” hints at an improperly chosen current directory or a missing piece of source-code. These problems have been treated on page 59, where the installation of a Java Development Kit has been considered.

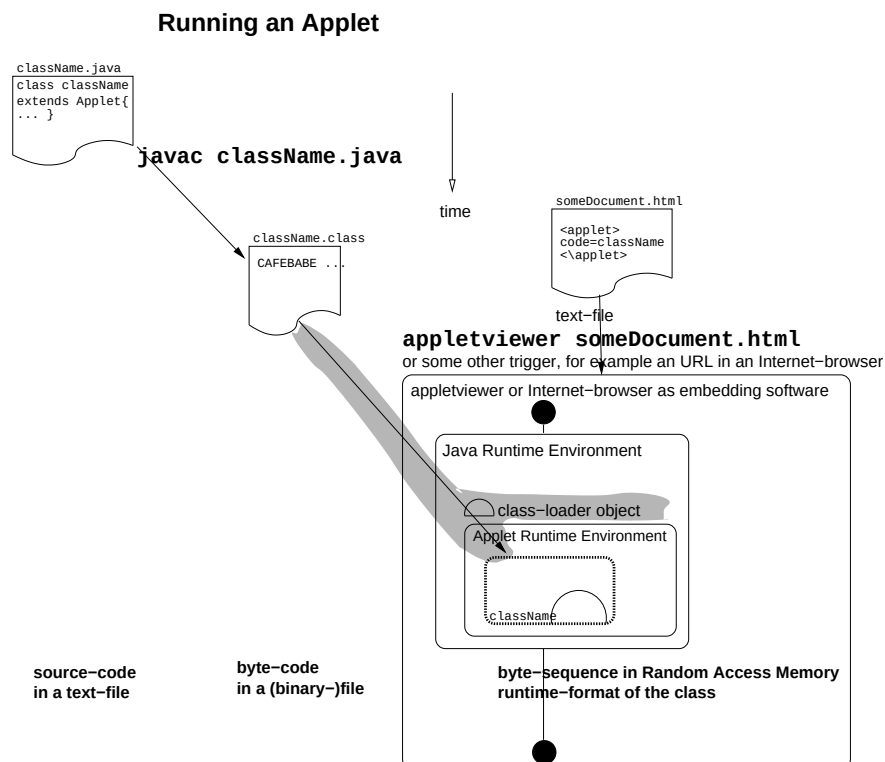
browser-software starts a Java Runtime Environment topped by an applet's sandbox (establishing an applet runtime-environment) and runs the applet's byte-code therein (see the graphic on page 67 or on the following page).

Stop of the Java Runtime Environment (JRE):

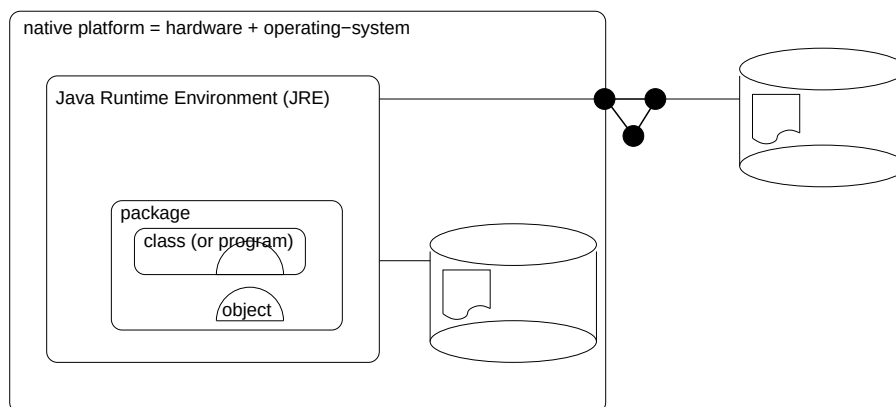
- Stop** When the end of the `main()`-method is reached, or when the end of the last thread's `run()`-method is reached, then the Java Runtime Environment shuts down. (Exception: Daemon-threads are stopped immediately by the Java Runtime Environment when the last non-daemon-thread has finished.)
- Stop** When the method `java.lang.System.exit( int )` is encountered, as demonstrated in the source-code on page 382. But this method-call terminates the Java Runtime Environment by immediately cutting off all threads. A return value 0 of primitive-type `int` signals an intended close-down, other values might be used to signal exceptional states.
- Stop** The Java Runtime Environment also can be stopped externally by the end-user: When the end-user enters the command `Ctrl-C` into the terminal-window from which has been started the Java Runtime Environment, then the Java Runtime Environment stops immediately, as demonstrated in some of the past terminal-listings (see before or on page 404).
- Stop** An error thrown, or an exception not properly caught within a single-threaded program, may stop the Java Runtime Environment (JRE). (page 279)

Control and Information about the Java Runtime Environment (JRE) is given by the following means:

- An end-user may enter the command `java - version` into the terminal-window. The resulting printout gives the number of the Java version installed on the computer-platform.
- Methods of the `java.lang.Runtime` and `java.lang.System`-classes give information about the context of the Java source-code in which they occur (see page 258).



In this context remember the environment diagram on page 54, and compare it with the one below: There are shown symbols for object, class, package, operating-system- and network-environments. Only if the relation of their content, especially with respect to the ideas of a name-space is to be stressed, the environments are represented by rounded boxes:



Since the Java Runtime Environment (JRE) generally¹⁷⁸ acts as the name-space for all the classes it has loaded and all the objects it has instantiated, the following questions arise again: How to make packages, classes and objects, which are stored in files, accessible in the Java Runtime Environment (JRE). How does the JRE turn these files of byte-code into working code? And how can the application-programmer trigger such a conversion process?

packages	By package installation , see beginning with page 244.
classes	By class-loading , this is shown in the next sections.
objects	By instantiation or deserialization after a class-loading-process

Another reformulation of the question above may read: After getting the byte-code from a hard-disk into the Random Access Memory (RAM) of the computer, how is the byte-code — given as a sequence of bytes — turned into a referencable class of the Java Runtime Environment? — In other words: How are the class-definitions, which have been compiled to byte-code, made referencable within a Java Runtime Environment? This transfer of files of byte-code into the computer's Random Access Memory and its transformation into a class known by the Java Runtime Environment is considered more closely in the next sections. Exactly this process is called class-loading:

How to turn a class-file into a loaded Java-Class?

Class-Loading

Class-Loading — Making a class known to the Java Runtime Environment — bringing it into a referencable form inside the computer's RAM — requires some source from where the Java Runtime Environment (JRE) can read the class-file containing the class's byte-code: Thus, to introduce classes to an existent Java Runtime Environment, that runtime environment has to ACCESS A PLATFORM WITH A PERSISTENT STORAGE-DEVICE. The sections about class-loading assume, that the platform in question has some persistent storage-device, which is a part of its Hierarchical File System. Or the platform is assumed to be connected to a persistent storage-device over a network, as already has been indicated by the diagram in the previous section. This means that files of byte-code and the files of the Java Runtime System can be written or found on a storage-device between sessions of running the platform. Thus, there is available a means for transferring information between successive instantiations of the Java Runtime Environments! Most of the personal computers with a hard-disk exactly are platforms of that kind.

class-loading needs a persistent storage-device

loading a class means: A byte-stream is converted to a class's runtime-representation in the RAM

Class-Loading: In Java, this describes the process of introducing class-types to the Java Runtime Environment. Class-loading centers around considering how to turn a sequence of bytes into a referencable class within the current Java Runtime Environment. Often, class-loading includes the retrieval of Java byte-code from a class-file on some persistent storage-device.

Thus, a class defined in a text-file of source-code takes three different states:

- The state of its definition in a text-file, a Unicode-text.
- The state of being in the form of Java byte-code in a class-file — a sequence of bytes, identifiable as Java byte-code by the first four bytes which, hexadecimally coded, yield the sequence CAFEBAFE.
- And finally the state the class does its work within the Java Runtime Environment: The runtime-representation of the byte-code after having been loaded into the computer's RAM by the running Java Runtime Environment.

The two file-related states are those in which the class can be made persistent. These three states have been visualized within the graphics at the beginning of this chapter.

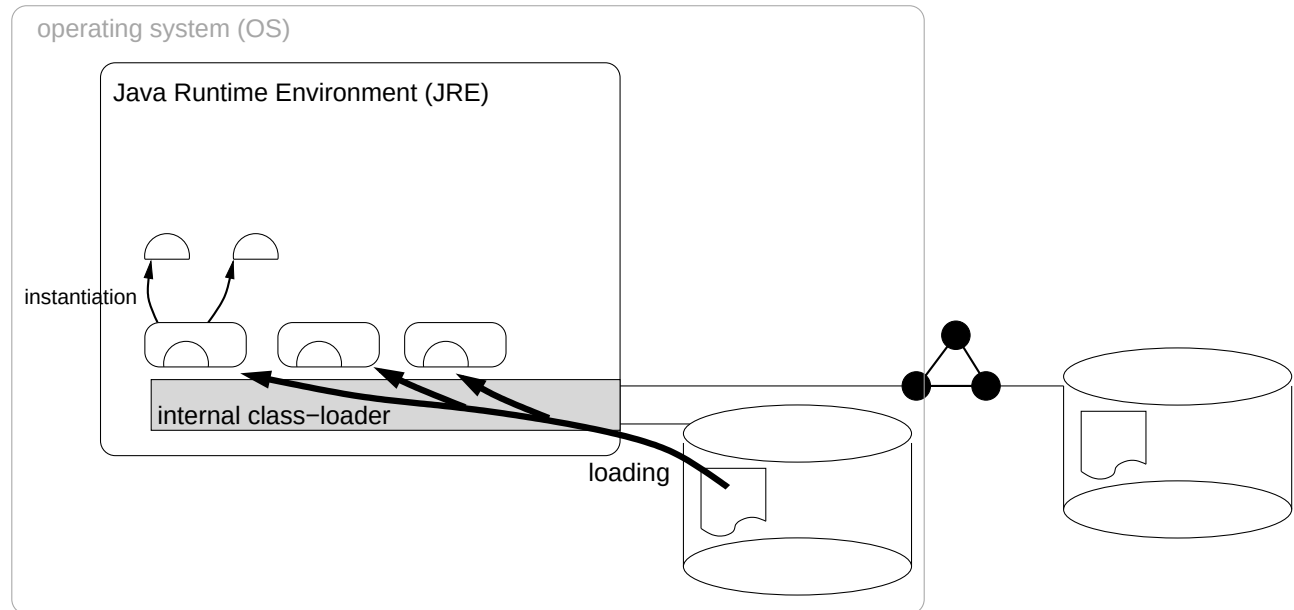
¹⁷⁸ Exceptions may be, for example, inner- or local classes and derived objects; as well as anonymous versions of classes and objects.

The notions “runtime-representation of a class”, “referencable class”, “loaded class”, “valid class reference” mean the same thing in this context.

Class-Loading: The Initial Process

Class-Loading: The Initial Process — The graphics at the beginning of this chapter show that issuing the `java`- or `appletviewer`-commands, during a session of the computer-platform, establishes a Java Runtime Environment. — Early in that process a **default class-loader** searches the Java base-classes (`rt.jar`, short for run-time, and possibly parts of `i18n.jar`, short for internationalization) and loads them into the Java Runtime Environment¹⁷⁹. That default class-loader is also called **bootstrap**-, **internal**-, **primordial**-, or **null-class-loader** (and as a part of the Java Runtime System this class-loader commonly is written in native code of the platform).

bootstrap-,
internal-,
primordial-,
null-class-loader
loads only
core-classes



A user or a programmer do influence this initial class-loading-process only by initiating it. Invoking the “`java -verbose HelloWorld`”-command within a terminal-window, with the explicitly added `-verbose`-option, and having a valid `HelloWorld.class` in the current directory, gives an impression of that initial class-loading-process (and the successive class-loading, treated later):

```
andreadpietro@linux:~/JavaCodeFiles/JRESecurity/JREClassLoading > javac HelloWorldCL.java
andreadpietro@linux:~/JavaCodeFiles/JRESecurity/JREClassLoading > java -verbose HelloWorldCL
[Opened /usr/lib/jdk1.3/jre/lib/rt.jar]
[Opened /usr/lib/jdk1.3/jre/lib/i18n.jar]
[Opened /usr/lib/jdk1.3/jre/lib/sunrsasign.jar]
[Loaded java.lang.Object from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.io.Serializable from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Comparable from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.String from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Class from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Cloneable from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.ClassLoader from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Throwable from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Error from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.ThreadDeath from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Exception from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded java.util.Vector from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.StringBuffer from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Boolean from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Character from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Number from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Float from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Double from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Byte from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Short from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Integer from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.lang.Long from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded java.util.StringTokenizer from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.io.File from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.io.FileSystem from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.io.UnixFileSystem from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded java.lang.Runtime from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded java.lang.Math from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded java.util.TimeZone from /usr/lib/jdk1.3/jre/lib/rt.jar]
```

¹⁷⁹In Java Versions 1.1 and lower, the initial class-loader could load all the classes (including the core-classes found in `classes.zip` file) from the `CLASSPATH` directories. For security reasons, in Java Versions 1.2 and higher only the core-classes (found in a `rt.jar`-file, short for Run-Time) are loaded by the initial class-loader.

```

[Loaded java.util.SimpleTimeZone from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.util.Calendar from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.util.GregorianCalendar from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.util.TimeZone$1 from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.util.TimeZoneData from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.util.ResourceBundle from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded java.util.Date from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded java.security.SignatureSpi from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.security.Signature from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded sun.security.provider.DSA from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded sun.misc.Launcher$AppClassLoader from /usr/lib/jdk1.3/jre/lib/rt.jar]          <<<<< secondary class-loader loaded
...
[Loaded sun.misc.Launcher from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded java.net.URLConnection from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded sun.net.www.URLConnection from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded sun.net.www.protocol.file.FileURLConnection from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded java.net.ContentHandler from /usr/lib/jdk1.3/jre/lib/rt.jar]
...
[Loaded com.sun.crypto.provider.SunJCE]
...
[Loaded java.lang.RuntimePermission from /usr/lib/jdk1.3/jre/lib/rt.jar]
[Loaded HelloWorldCL]
  an object's ClassLoader-class: class sun.misc.Launcher$AppClassLoader
  an object's ClassLoader-object: sun.misc.Launcher$AppClassLoader@6f241
  a class's class-loader class: class sun.misc.Launcher$AppClassLoader
  a class's ClassLoader-object: sun.misc.Launcher$AppClassLoader@6f241
java.lang.Object has null-class-loader.
[Loaded java.lang.ShutdownLock from /usr/lib/jdk1.3/jre/lib/rt.jar]
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/JREClassLoading >

```

The list above includes also non-core-classes as can be seen by the last line before the program's run: There is loaded the program's own class. The original output is much larger, but even the short version shows some of the well-known classes like `java.lang.String` and others.

Class-Loading: The Secondary Process

Class-Loading: The Secondary Process — As could be seen before, the “java”-command triggers the INTERNAL CLASS-LOADER to load the CORE-CLASSES of the Java Runtime System. Part of these core-classes are secondary class-loaders. OTHER CLASSES are loaded by those SECONDARY CLASS-LOADER(S): In the source-code of the program below, the method `java.lang.Class.getClassLoader()` returns references to class-loader objects. There, core-classes (`java.lang.String`) let that method return a null-reference, indicating that they have been loaded by the internal- or null-class-loader. Non-core-classes are recognized as been loaded by another class-loader! (output: `sun.misc.Launcher$AppClassLoader`) This secondary class-loader has been loaded itself by the internal class-loader as can be seen also in the previous printout, above (`[Loaded sun.misc.Launcher$AppClassLoader from /usr/lib/jdk1.3/jre/lib/rt.jar]`).

Among those core-classes that have been loaded, are numerous class-loader classes, identifiable by a substring “ClassLoader” in their name. As part of the initialization-process after issuing a java-command, the Java Runtime Environment instantiates the appropriate class-loader and uses it to load the class the user has specified (details of the program below are explained later):

secondary
class-loaders
search classes to
load in the
current directory
and in the
directories in the
CLASSPATH

The following program is given by a class with a static `main()`-method and an instance of itself. The class' method as well as the constructor of its instance contain code which returns references to the associated class-loader(s). As can be seen below, the (secondary) class-loader of the custom-written class can be accessed from the class and from the derived instance. The type of the class-loader and the internal name of the `ClassLoader`-object are printed out. Whereas the class-loader of the `java.lang.String`-class can be identified as null, because this class has been loaded by the internal class-loader.

```

// how to get a ClassLoader-object
public class HelloWorldCL extends java.lang.Thread
{ public HelloWorldCL(){ super(); this.start(); }
  public void run()
  { try
    { ClassLoader cl = this.getClass().getClassLoader();
      System.out.println(" an object's ClassLoader-class: " + cl.getClass().toString());
      System.out.println(" an object's ClassLoader-object: " + cl.toString());
    }catch( Exception e ){ System.out.println(e); } // another ALL PURPOSE CATCH
  }
  public static void main( String unused[] )
  { new HelloWorldCL();
    try
    { ClassLoader cls = Class.forName("HelloWorldCL").getClassLoader();
      System.out.println(" a class's class-loader class: " + cls.getClass().toString());
      System.out.println(" a class's ClassLoader-object: " + cls.toString());
      ClassLoader cl0 = Class.forName("java.lang.Object").getClassLoader();
      // The line above will return the null-class-loader
      if( cl0 == null ) System.out.println( "java.lang.Object has null-class-loader." );
      else System.out.println( cl0.getClass().toString() );
    }catch( Exception e ){ e.printStackTrace(); } // ALL PURPOSE CATCH
  }
}

```

```

} }

/*
C:\WINDOWS\JavaCodeFiles\JRESecurity\JREClassLoading>javac HelloWorldCL.java

C:\WINDOWS\JavaCodeFiles\JRESecurity\JREClassLoading>java HelloWorldCL
a class's class-loader class: class sun.misc.Launcher$AppClassLoader
an object's ClassLoader-class: class sun.misc.Launcher$AppClassLoader
a class's ClassLoader-object: sun.misc.Launcher$AppClassLoader@71732b
an object's ClassLoader-object: sun.misc.Launcher$AppClassLoader@71732b
java.lang.Object has null-class-loader.

C:\WINDOWS\JavaCodeFiles\JRESecurity\JREClassLoading>
*/

```

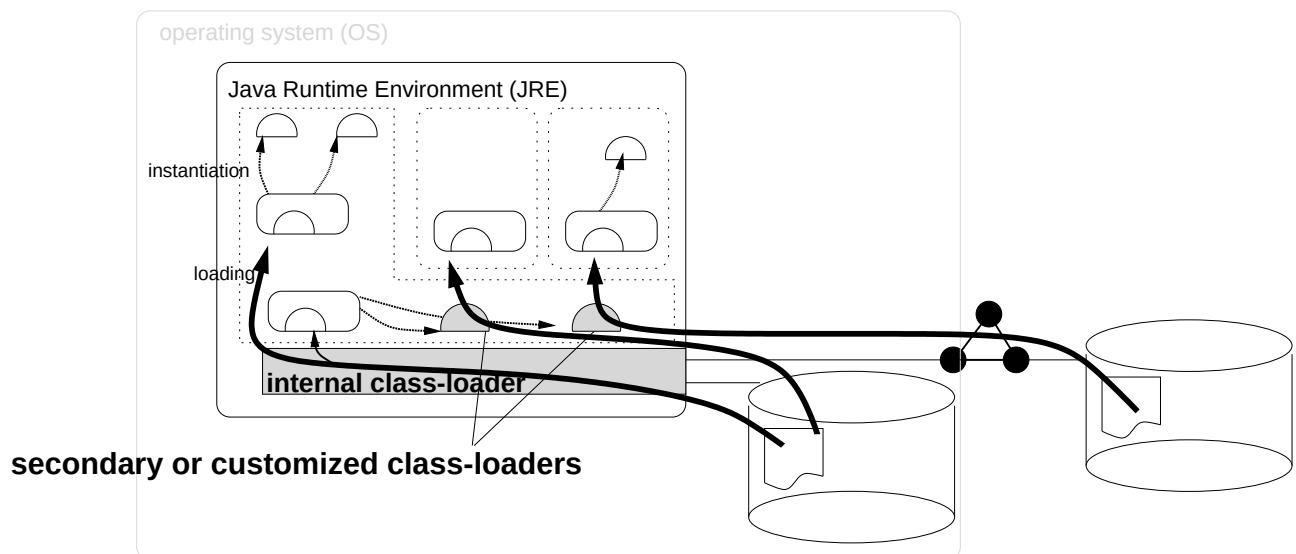
Every use of the `java`-command OR a reference to a compiled and accessible class triggers that automatic class-loading-process. When such a *secondary* class-loader searches for a class-file, then the so-called

1. **current directory** is searched first,
2. then all of the **directories indicated by the CLASSPATH-variable** are searched
3. and finally a `java.lang.NoClassDefFoundError` is issued.

(Besides the previous instantiation error, the infamous `java.lang.ClassNotFoundException`, issued at run-time, indicates that some referencing went wrong.)

WHAT IS A CLASS-LOADER IN THE JAVA LANGUAGE-CONTEXT? **All CLASS-LOADERS ARE JAVA OBJECTS**, EXCEPT FOR THE INTERNAL CLASS-LOADER. Again, except for the internal class-loader all other `ClassLoader`-objects are derived from extensions of the abstract class `java.lang.ClassLoader`. The `java.lang.ClassLoader`-class contains no abstract methods, nevertheless it has been declared abstract to encourage the writing of extensions.

class-loader in
Java: an object
(except for the
internal
class-loader)



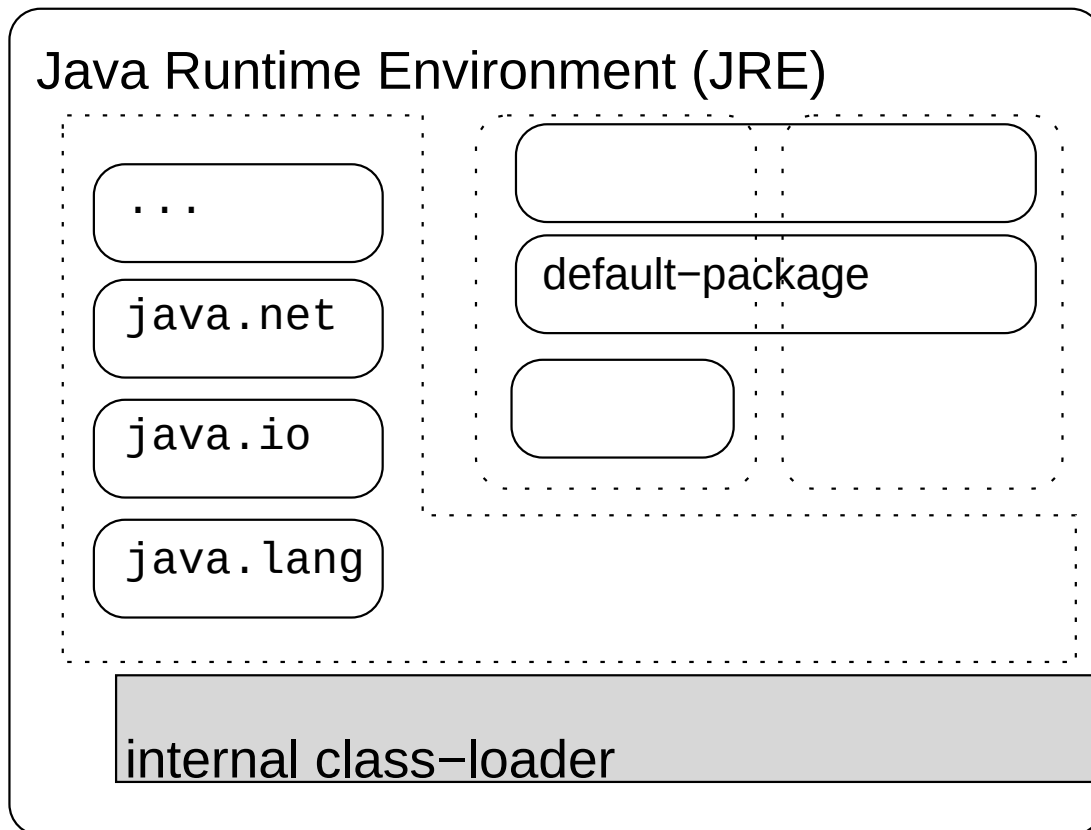
CLASS-LOADER'S NAME-SPACE: The initial class-loader and each of the loaded class-loaders produce an environment for the classes they load and the derived instances. These environments are name-spaces since they react to access-modifiers (`public`, `protected`, `default`) like a package. EVERY CLASS-LOADER PRODUCES A NAME-SPACE with access-restrictions like those of a package. In the graphics, these name-spaces are indicated by the dotted lines. Thus, each loaded class and its derived object has to belong to one name-space! (In the same way each loaded class and derived objects belong to one package.) That means the loaded classes and their instantiated objects are known to outside callers only by their class-loader-qualified names and public members, in cases of derived classes also by protected members. A programmer can determine the class-loader and thus the associated name-space as described in the program above.

class-loader
name-space: like
that of a
package, but
NOT a package

Reconsider the set of all loaded classes and derived objects within a Java Runtime Environment: Since each loaded class and derived objects belong to one package, packages partition that set. Since each loaded class and its derived object has to belong to one name-space of a class-loader (including the null-class-loader), the name-spaces of the associated class-loaders partition that set too! (this is also visualized around page 469). What about a relation between these two partitions of the set of all loaded classes and

derived objects in a given Java Runtime Environment? Classes within the default-package¹⁸⁰ may be loaded by different class-loaders. The initial class-loader loads a lot of different core-packages. So there exists no general and obvious relation readily recognizable¹⁸¹.

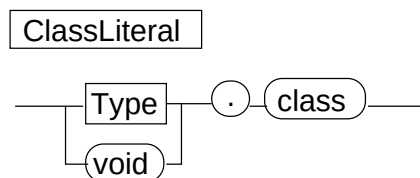
name-spaces of class-loaders and packages



CLASSES AND OBJECTS INFORMING ABOUT PROPERTIES OF CLASSES AND OBJECTS: Each object of class-type `java.lang.ClassLoader` automatically instantiates an object of class-type `Class` for each of its loaded classes. For the loaded class, such a `java.lang.Class`-object provides basic information and basic functionalities (described on page 252 and in the ➡ ➡ Java Platform API Specification [JPAPIS]). The `Class`-object can be referenced from a **class**:

class →
Class-object

```
Class x = Class.forName( className )    // or by using the class-literal
Class x = className.class
```



ClassLiteral := *Type* . class | void . class — Type 76

Having been given an **object** of some class-type, a reference to the associated `Class`-object can be produced by using the `java.lang.Object.getClass()`-method:

object →
Class-object

```
Class x = objectName.getClass()
```

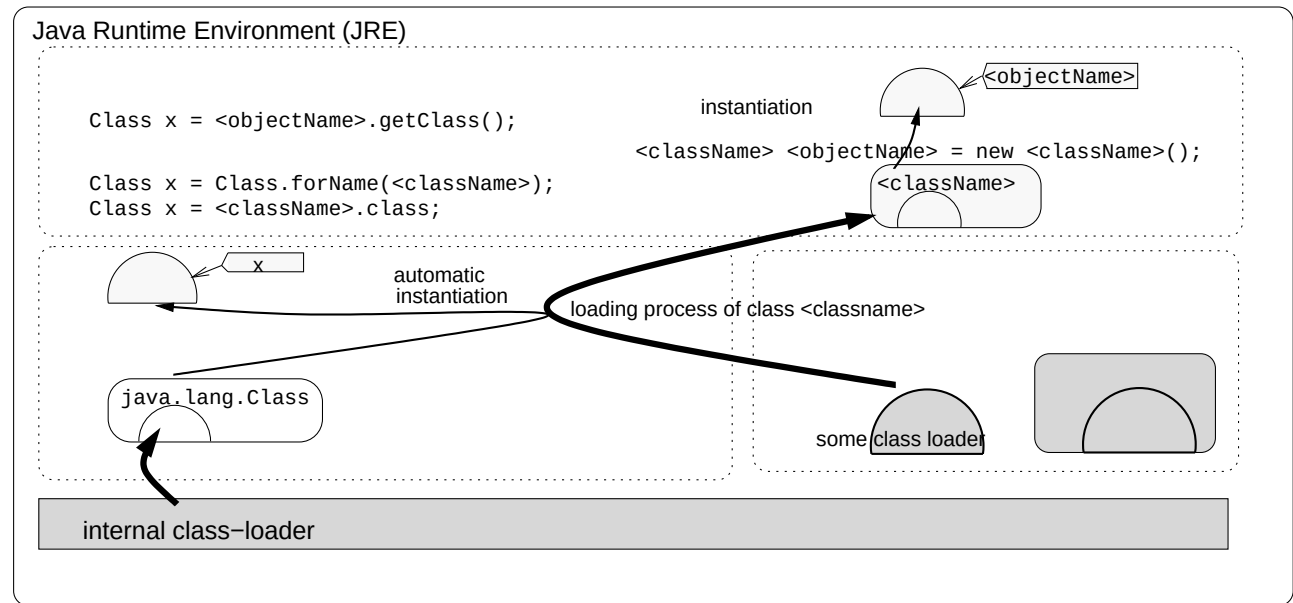
(Both of the methods above are factory-like methods because they return references of pre-existent objects or code, page 185). Having a reference to a `Class`-object, the associated class-loader can be referenced by using the instance-method `java.lang.Class.getClassLoader()`.

Class-object →
ClassLoader-
object

¹⁸⁰Java-code from source-code without any package-declaration runs in the default-package, as introduced on page 241.

¹⁸¹Class-loader name-spaces are also treated in the Java TechTip of Oct. 31st of 2000.

Is it possible for two different class-loaders to load different classes with the same name? These classes then are – analogously to packages – only referencable by qualifying them with a reference of their respective class-loader.



This secondary class-loading, by objects of class-type `java.lang.ClassLoader`, especially makes it possible to load and instantiate customized (readily written or custom-written) class-loaders (For more details about the loading-process see the next sections).

Class-Loading: Customized Class-Loaders

Class-Loading: Customized Class-Loaders — Reconsider the nearest graphics above to visualize the following statements: Loaded classes and derived objects always are in the same name-space¹⁸². Especially the `ClassLoader`-class and the `ClassLoader`-object are found in the same name-space. `ClassLoader`-objects remain outside the name-spaces they define¹⁸³!

How can a class or an object see whether another class or object exists in another package? Only public members can be accessed, and if the class has been extended into the other package, then all protected members remain accessible from that class or within a derived object.

`java.lang.ClassLoader` — Simple Extension

`java.lang.ClassLoader` — Simple Extension — An easy way to instantiate a fully-functional class-loader is to trivially extend the `java.lang.ClassLoader`-class and instantiate the extension. Although the class is declared abstract, its core methods (`loadClass()`, `defineClass()`, `findSystemClass()`, `findLoadedClass()`) are fully-implemented and final (the `loadClass()`-method isn't declared final only for reasons of backward compatibility¹⁸⁴). If the name of the class to be loaded is known, and the search is limited to the directories in the `CLASSPATH`-variable, then the instance-method `java.lang.ClassLoader.loadClass()` already furnishes the complete loading services.

`java.net.URLClassLoader` — Simple Instantiation

`java.net.URLClassLoader` — Simple Instantiation — A further customizable class-loader can be easily derived from the fully-implemented class with the name `java.net.URLClassLoader`:

¹⁸²This does not forbid to EXTEND public classes over different name-spaces, for example remember the exception-hierarchy on page 281, which extends over several packages!

¹⁸³Imagine a chain of class-loaders, each class-loader loading the class of its successor. Since each class-loader is supposed to define its own name-space, there emerge two possibilities: Either (I) a class-loader exists inside its name-space or (O) a class-loader exists outside its name-space. — And since a class-loader ALWAYS loads classes into its own name-space and instantiations happen ALWAYS within the name-space, other class-loaders are instantiated within that name-space, either with (I) giving a nested name-space; or with (O) producing a name-space outside. Excluding nested name-spaces (hierarchically related name-spaces) thus means `ClassLoader`-objects and -classes remain outside the name-spaces they define!

¹⁸⁴In Java Versions 1.1 and lower, that method was declared abstract and had to be implemented by the user using the fully-implemented and final `defineClass()`-method.

1. The programmer needs only to furnish an array of Uniform Resource Locators (URLs) these will be searched consecutively by the `URLClassLoader`-object¹⁸⁵.
2. And, as an option, the programmer may furnish a reference to another class-loader requesting it to search first. Both cases are covered by the two constructors of the class `java.net.URLClassLoader`.
3. Finally, the programmer has to furnish the name of the class to be loaded by the `URLClassLoader`-object, and the programmer should advise the class-loader to load the class with the `loadClass()`-method (as demonstrated in the Java source-code below).

The following program demonstrates this by searching, loading and instantiating a class:

```
/* The program below loads a class ToBeSearched by searching it within given
   list of Uniform Resource Locators (URLs).
*/
class URLLoad
{ static java.net.URL urlSearchList[] = new java.net.URL[3];
  public static void main( String unused[] )           // main()-method
  { try
    { // Uniform Resource Locator's format is independent of platform used
      { urlSearchList[0] = new java.net.URL("file:/Users/andreadipietro/JavaCodeFiles/JRESecurity/PolicyBasedSecurity/");
        urlSearchList[1] = new java.net.URL("file:/Users/andreadipietro/JavaCodeFiles/JRESecurity/DataCodingSecurity/");
        urlSearchList[2] = new java.net.URL("file:/Users/andreadipietro/JavaCodeFiles/JRESecurity/");
      }
    } catch( java.net.MalformedURLException e ){ e.printStackTrace(); }
    try
    { java.lang.ClassLoader parent = Class.forName("URLLoad").getClassLoader();
      java.net.URLClassLoader urlCL =
        new java.net.URLClassLoader( urlSearchList, parent );
      Class x = urlCL.loadClass( "ToBeSearched" ); // <-- The Searching Triggered --
      System.out.println( " " + x.getClassLoader());
    }
    /* Which class-loader is used, URLClassLoader or parent class-loader,
       depends in which directory the class-file of the class below is placed.
    */
    x.newInstance();
  } catch( java.lang.ClassNotFoundException e ){ System.out.println(e); }
  catch( java.lang.IllegalAccessException e ){ System.out.println(e); }
  catch( java.lang.InstantiationException e ){ System.out.println(e); }
} }
/*
(ToBeSearched uncompiled) [localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro% javac URLLoad.java
(ToBeSearched uncompiled) [localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro% java URLLoad
(ToBeSearched uncompiled) java.lang.ClassNotFoundException: ToBeSearched
(ToBeSearched uncompiled) [localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro%
//
// When file ToBeSearched.class is in current directory or a directory of the CLASSPATH-variable:
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro% javac URLLoad.java
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro% java URLLoad
sun.misc.Launcher$AppClassLoader@33056f
class ToBeSearched loaded!
class ToBeSearched instantiated!
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro%
// When file ToBeSearched.class is in a directory of the urlSearchList:
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro% java URLLoad
java.net.URLClassLoader@7c6c7
class ToBeSearched loaded!
class ToBeSearched instantiated!
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro%
*/
/*
if ToBeSearched DECLARED WITHOUT MODIFIER "public",
class can be only called from classes and objects in the name-space of the URLClassLoader urlCL
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro% java URLLoad
java.net.URLClassLoader@7c6c7
class ToBeSearched loaded!
java.lang.IllegalAccessException: ToBeSearched
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] andreadipietro%
*/
```

Here the piece of code to be searched for:

```
// works together with URLLoad.java
public class ToBeSearched
{ static{ System.out.println("class ToBeSearched loaded!"); }
  { System.out.println("class ToBeSearched instantiated!"); }
}
```

¹⁸⁵This list of URLs gives an equivalent to that of the CLASSPATH-variable, which gets searched by the standard secondary class-loader.

```

/*
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/JREClassLoading > javac ToBeSearched.java
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/JREClassLoading > java ToBeSearched
class ToBeSearched loaded!
Exception in thread "main" java.lang.NoSuchMethodError: main
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/JREClassLoading >
*/

```

The terminal printout above, producing the `java.lang.IllegalAccessException`, demonstrates the sensitivity of the class-loader's name-spaces to the use of access-modifiers. Notice also how another class-loader can be given the priority in the searching-process by referencing that higher-priority class-loader in the constructor of the lower-priority class-loader (this setting is also known as **class-loader delegation**).

class-loader
delegation: initial
and secondary
class-loaders
ALWAYS load first,
if they can.

When a customized class-loader has been called, it first automatically consults the initial and secondary class-loaders before customizingly delegating the class-loading. Why? This configuration assures that no core-classes or CLASSPATH loadings can be overridden by any customized class-loader, thus allowing class-imitations to take over the Java Runtime Environment.

Actually the secondary class-loader usually is a `java.net.URLClassLoader` for which the platform's CLASSPATH-variable is transformed into an array of URLs.

Delegation: 1. In the context of class-loading: Class-loaders may call another class-loader which searches first for the class in question. For an example see the source-code above.
2. In the context of event-processing: The source-event-drain-model (page 271) can be specialized into the delegation event-model (page 276). This specialized model strives to minimize any organizational overhead on the side of the event-source, only the registration of drains remains with the source. Everything else is delegated to the relation between event and drain. — The supplying of information is delegated to a newly created object: The message- or event-object. References of this object, dispatched to the drain-objects trigger the call of associated handler-methods.
Contrary to the delegation event-model, in the chain-model, events have to propagate along some given hierarchy. (A method-call hierarchy in the case of exception-throwing, on page 282 or a containment-hierarchy of screen-components in the case of the very little used AWT chain-model (AWT acronym for Abstract Window Toolkit), on page 591.)

The Full Loading-Process of Instances derived from `java.lang.ClassLoader`

The Full Loading-Process of Instances derived from `java.lang.ClassLoader` — AGAIN, WHAT DOES A CLASS-LOADER DO? A class-loader often searches for a file or some other resource to get AN ARRAY OF BYTES (hopefully Java byte-code) and tries to transform that array into a referencable Java class in the current Java Runtime Environment. Of course, a class-loader (customizable or internal) needs some hints on the location (URLs, paths in a hierarchical file-system) from where to get the file of byte-code which represents the class to be loaded.

The entire process of loading a class into a Java Runtime Environment can be described coarsely in the following steps:

Asks class already loaded? Use method `java.lang.ClassLoader.findLoadedClass()`. Yes? Then exit.

(invokes the `java.lang.SecurityManager`-object¹⁸⁶ and asks if access to the class is permitted? No? Then exit.)

Delegates loading if possible: The first thing the `loadClass()`-method does is calling the `loadClass()`-method of the internal class-loader or the `loadClass()`-method of the class-loader whose reference has been given as an argument of present class-loader's constructor. (The non-static method `java.lang.ClassLoader.getParent()` retrieves a reference of that delegation class-loader.) NON-INTERNAL CLASS-LOADERS ASK THE INTERNAL CLASS-LOADER FIRST¹⁸⁷ and then take over, unless the security-manager requires a different even more restricted approach.

(Security-manager: Loading of class allowed? No? Then exit.)

Retrieves an array of bytes Either by reading a file from disk or by reading the file-data from the network into an array of bytes. This can be done from a wide range of input formats (class-file-format, JAR-file-format, input-streams, byte-array produced during run-time)

¹⁸⁶To be introduced on page 462.

¹⁸⁷This makes the loading-process safe against loading forged core-classes.

Transforms array of bytes into Java-referencable form The following steps have been already incorporated in the `java.lang.ClassLoader.defineClass()`-method:

1. The `defineClass()`-method brings the array of bytes into a referencable form,
2. recursively loads super-classes,
3. checks the byte-code for inconsistencies with the byte-code-verifier (page 451) furnished by the Java Runtime Environment (the internal class-loader does not verify classes when loading them).
4. A block of source-code to be run when the class is loaded is the so-called static-initializer, refer to page 152. (Remember the notion of an instance-initializer presented on page 152 and how it differs from a constructor.)
5. Finally, the `defineClass()`-method makes the class known to the JRE by instantiating the `java.lang.Class`-object and returning a reference to it.

Resolves the class By calling the `java.lang.ClassLoader.resolveClass()`-method explicitly, if necessary. This method automatically searches for references to other classes and loads the related classes, so that the entire set of classes, needed to instantiate the object-graph (graph of references), is loaded after this method returns.

If the name of the class to be loaded is known, and the search is limited to the CLASSPATH, then the `java.lang.ClassLoader.loadClass()`-method incorporates¹⁸⁸ all the steps described above.

loadClass() automatically searches the CLASSPATH, uses the default class-loader, defines and resolves the class, returns the reference to a `java.lang.Class`-object that makes the newly loaded class usable¹⁸⁹.

In all other cases the application-programmer has to custom-write a method like “myClassLoading-Method()” which contains the implementation of the steps above, by using the methods introduced below. So, a custom-written class-loading method uses the following fully-implemented `final` (non-overridable) methods furnished by the abstract `java.lang.ClassLoader`-class:

findLoadedClass() decides whether the given name denotes a class already existing in the loaded state.

findSystemClass() searches the CLASSPATH and uses the default class-loader.

defineClass() takes any array of bytes; if that array represents the valid byte-code of a class, then turns that array of bytes into a loaded class of the current Java Runtime Environment, returning a `java.lang.Class`-object¹⁹⁰. This is the `final` method that makes every class-loader automatically instantiate an associated `Class`-object for each loaded class.

resolveClass() recursively calls all the references and, if necessary, automatically does the needed class-(down)loading using the appropriate class-loaders it finds by analyzing the `Class`-objects of the references.

Maximally programmer implemented Class-Loader

Maximally programmer implemented Class-Loader — A simple class-loader for producing a valid reference in the Java Runtime Environment and an associated `Class`-object, is given by the following one:

```
// how to transform a byte-array into the runtime-representation of a class
/* source-code in file named ByteArrayLoader.java */

public class ByteArrayLoader extends java.lang.ClassLoader //-> (public or not public)
{
    Class transformToClassFromByteArray( String className, Class c, byte [] byteArray )
    { System.out.println( "byte-array to runtime-representation of class" );
      c = defineClass( className, byteArray, 0, byteArray.length );
                                     // automatic verification
    }
```

¹⁸⁸In Java Versions 1.1 and below, this process had to be programmed by the application-programmer using the help of the fully-implemented, `final` methods of the `java.lang.ClassLoader`-class, especially the `defineClass()`-method.

¹⁸⁹In Java versions prior to Version 1.2 this method had to be implemented by the application-programmer.

¹⁹⁰The `java.lang.Class`-objects, instantiated by any class-loader, can be found ALL inside the name-space of the initial class-loader and inside the `java.lang` package! For a visualization see the graphic on page 438.

```

        resolveClass( c );
        return c;
    }
    public void test(){ System.out.println( "Will be called later from a loaded class." ); }
}

```

The two pieces of source-code, above and below, are put into two separate files.

```

/* source-code in file named LoaderDemo.java */

class LoaderDemo
{
    static byte [] getBytes( String fileName )
    { java.io.ByteArrayOutputStream bytesOut = new java.io.ByteArrayOutputStream();
      int bufferForByte = 0; // -1 indicates End Of File
      try
      { java.io.BufferedInputStream bytesIn =
        new java.io.BufferedInputStream( new java.io.FileInputStream( fileName ) );
        while(true)
        { bufferForByte = bytesIn.read();
          if(bufferForByte == -1) break;
          else bytesOut.write( bufferForByte );
        }
      }catch(Exception e)
      { System.out.println( "File: " + fileName + e.toString() ); return null;
      }
      return bytesOut.toByteArray();
    }
    static Class c;

    static void main( String fileNameAndClassName[] )
        // first class-name, second fileName without .class postfix
    { String className = fileNameAndClassName[1];
      String fileName = fileNameAndClassName[0];
      ByteArrayLoader loader = new ByteArrayLoader();
      byte [] theByteArray = getBytes( fileName );
      c = loader.transformToClassFromByteArray( null, c, theByteArray );
        // or className instead of null
      try{ ((Thread)c.newInstance()).start(); } // class of type Thread expected
      catch( Exception e ){ e.printStackTrace(); } // ALL PURPOSE CATCH
    } }

/* ----- ByteArrayLoader declared public, as above -----
C:\WINDOWS\JavaCodeFiles\JRESecurity\JREClassLoading>javac LoaderDemo.java

C:\WINDOWS\JavaCodeFiles\JRESecurity\JREClassLoading>java LoaderDemo HelloWorldThread.class HelloWorldThread
byte-array to runtime-representation of class
HelloWorldThread
Will be called later from a loaded class.
null
C:\WINDOWS\JavaCodeFiles\JRESecurity\JREClassLoading>
*/
/* ----- ByteArrayLoader NOT declared public ----- //-----> (public or not public)
C:\WINDOWS\JavaCodeFiles\JRESecurity\JREClassLoading>javac LoaderDemo.java

C:\WINDOWS\JavaCodeFiles\JRESecurity\JREClassLoading>java LoaderDemo HelloWorldThread.class HelloWorldThread
byte-array to runtime-representation of class
HelloWorldThread
java.lang.IllegalAccessException
    at java.lang.reflect.Method.invoke(Native Method)
    at HelloWorldThread.run(HelloWorldThread.java:7)

C:\WINDOWS\JavaCodeFiles\JRESecurity\JREClassLoading>
*/

/* The java.lang.IllegalAccessException comes from the loaded class.
   For more on this exception see the following piece of source-code.
*/

```

Even the name of the to be loaded class need not to be known. But here it has been assumed that the loaded class extends the Thread-class, indicated by casting the object to a Thread-type and then starting the thread. The class' reflection tools allows to get information about the class' members (page 629). Written in a separate file¹⁹¹ the source-code of the class to be loaded reads: (The Java compiler produces the byte-code, stored in the class-file to be loaded by the class-loader above.)

```

/* source-code in file named HelloWorldThread.java */

public class HelloWorldThread extends java.lang.Thread
{ public void run()
  {
    System.out.println("HelloWorldThread");
  }
}

```

¹⁹¹Due to the public modifier of the class; there can be only one public class in each file of source-code, see page 246.

```

    { try
      { ClassLoader cl = this.getClass().getClassLoader();
        System.out.println(
          cl.getClass().getDeclaredMethod( "test", null ).invoke(cl,null) );
        // example of introspective use of Class-objects
      }catch( Exception e ){ e.printStackTrace(); }
    } } }

/*
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] marycosway% javac HelloWorldThread.java
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] marycosway% java HelloWorldThread
Exception in thread "main" java.lang.NoSuchMethodError: main
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] marycosway%
*/
/* The source-code above is not an application and
   therefore cannot be triggered to run using the java-command.
   The missing main()-method is noticed immediately when issuing the java-command.
*/
/* //-----> (public or not public)
   The piece of source-code above triggers the java.lang.IllegalAccessException,
   after having been loaded,
   when during runtime issuing the call of the class-loader's test()-method.
   This shows that class-loaders are not in the same name-space as their loaded classes,
   unless declared public!
*/

```

Custom-written class-loaders may be programmed to also fetch non-Java byte-code and transform the succession of bytes to a valid Java class maybe adding a valid Java object of that class-type, like image-files or document-files and any other resources. — Programs like this also may call the `main()`-method of some class. This means to establish something like a little Runtime Environment on top of the current Java Runtime Environment (see also page 630). In a multithreaded approach, this makes it even possible to RUN MULTIPLE JAVA-APPLICATIONS IN THE SAME JAVA RUNTIME ENVIRONMENT (also consult the previously referenced page). A somewhat similar idea concerning applets is realized in an elementary way in the source-code on page 619 which produces a mini-runner for applets.

Class-Loaders of Arrays?

Class-Loaders of Arrays? — Arrays have been already introduced as variables of reference-type or more precise as variables of array-type (there have been introduced two more reference-types in Java: Class-type and interface-type, for more see page 143). Variables of array-type have faint aspects of a class- and object-structure, for example the use of the `new`-keyword for instantiation as described on page 234. BUT ARRAYS OF PRIMITIVE-TYPE HAVE NO CLASS-LOADER, whereas arrays containing variables of reference-type have the class-loader of the type of the array's elements.

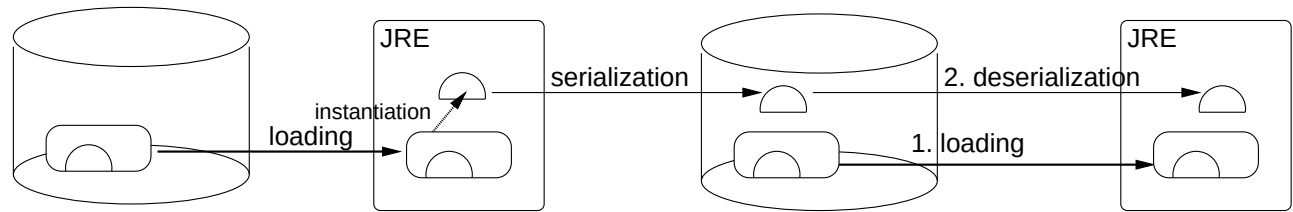
Simple Class-Loading: new-Keyword and the Class.forName()-Method

Simple Class-Loading: new-Keyword and the Class.forName()-Method — The simpler ways to trigger class-loading may easily be forgotten when occupied with the diversity of the customizable class-loading-process: The instantiation with the `new`-keyword triggers the class-loading-process that uses the secondary class-loader which has been introduced before. The static method `java.lang.Class.forName()` of the `Class`-class also can be used to introduce a class which can be found as a class-file in one of the directories of the `CLASSPATH`-variable, without the need of instantiation of an object of this class-type. An example for both methods of class-loading already gave the source-code and the graphic beginning with page 153. In daily Java programming, these methods of class-loading should be given preference; only if unavoidable, the use of customized class-loaders may be taken into account.

Object Storing and Loading: Serialization

Object Storing and Loading: Serialization — The process of serialization has been introduced already beginning with page 306. The previous sections occupied themselves with considering how to make a program transform an array of bytes in class-format into a runtime-representation of a class. This section (re-)considers how to take an instantiated object of some class-type and store it persistently, as well as retrieving it from the stored form and re-instantiating it. Of course, the re-instantiation needs the

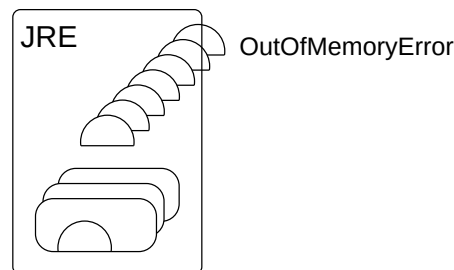
original class to be reloaded too, this is the point at which class-loading becomes of pivotal importance in deserialization of objects:



As shown in the last section, usually the class-loading process may be triggered only by declaring a variable or by instantiating an object. Nevertheless, as also shown before, class-loading may be prepared explicitly to happen by searching directories, for example, of the `CLASSPATH`-variable and/or various URLs. (The `load()`- and `store()`-methods of the `java.util.Properties`-class are solely modified `read()`- and `write()`-methods on to-be specified streams and have only a very indirect relation to class-loading or persistence.)

RAM Recycling: Garbage-Collection and Weak References

RAM Recycling: Garbage-Collection and Weak References — During a session, a Java Runtime Environment (JRE) loads or has loaded a set of classes and usually instantiates a larger set of objects. If the number of loaded classes and instantiated objects grows too large, then the Java Runtime Environment may finish its work by throwing an unrecoverable `java.lang.VirtualMachineError`, especially a `java.lang.OutOfMemoryError`.



(Deinstantiation of Objects) The de-instantiation of object can be ignored by those Java Runtime Environments in which the application-programs are known not to use excessive memory or a restart of the runtime-environment implies no loss. In any cases where a guaranteed Quality of Service (QoS) is intended and an application may fully use available resources, an `OutOfMemoryError` may become a virulent problem. To avoid unnecessarily early shutdowns of a Java Runtime Environment it is possible to devise an automatic runtime-based cleanup of objects which are not further needed. This is called automatic **garbage-collection**. The associated software runs in a low-priority thread and is called garbage-collector. Within a given Java Runtime Environment, how can be identified objects which are not needed any more?

For example, local objects within methods become inaccessible when the method returns. Those objects are dynamically deallocated when the implementation of the Java Virtual Machine (JVM) leaves their access-area (their environment where their identifier is accessible). This can be described on byte-code level: If the program-run leaves the access-area of the declaration, then the name and its associated reference cease to exist. There cannot exist any reference onto the object, that means the information gets lost. So the garbage-collector, running on a low-priority thread on the byte-code interpreter, deallocates the reserved memory.

Whether the Java Runtime Environment (JRE) has been put into existence with the “java”-command that starts the thread of the `main()`-method or whether the JRE is run from a Java-enabled browser-environment by starting an applet’s thread, this first thread instantiates the first objects. Further objects may be instantiated from this or other secondary threads. So there may occur several graphs of references, where the node is an object and the arrows denote references to other objects. During the running of the threads, name-spaces are changed and previously referencable objects may not be referencable any more. Then those objects, that have no references pointing onto them, can be erased from the computer’s memory. This task is done automatically by the Java Runtime Environment’s garbage-collector; it affects

only the objects, not the classes. The Java garbage-collector uses its own low-priority thread to collect unused objects. The Java programmer can enforce a garbage-collection-turn by calling the static method `System.gc()`.

For more on garbage-collection see literature like the Java Virtual Machine specification; but usually the application-programmer has to know solely of the existence of the garbage-collection and of the associated mechanism: Before the garbage-collector erases an object, it calls the `finalize()`-method of that object in a thread of its own, here the application-programmer can enter code that is to be run (like closing of a database or a network-connection, see also the remarks on the object-lifespan on page 153).

(Unload Classes?) How can loaded classes be unloaded? Usually not by the application-programmer and usually neither by the Java Runtime Environment; there exists no easy means to cause the unloading of an unused class.

(Concept of weak references) In cases where many objects occupy the computer's Random Access Memory (RAM) and all those objects are needed (for example image-objects or voluminous database information), it may be desirable to free *used* objects for garbage-collection, but to let them be erased only when there develops the immediate threat of a `java.lang.VirtualMachineError`. The application-programmer can accomplish that task by using the tools provided by the package `java.lang.ref`. These tools usually only need to be used when extensive caching is to be programmed, for an introduction to this area consult further literature.

(Overflow) The following program tries to strain a Java Runtime Environment (JRE) by instantiating a lot of String-objects until no more of the computer's Random Access Memory can be reserved for the JRE:

```
class Overflow
{ public static void main( String unused[] )
  { java.util.Vector v = new java.util.Vector();
    java.lang.Runtime rt = java.lang.Runtime.getRuntime();
    int i=0;
    long javaTotal, javaFree, previousTotal = 0;
    while(true)
    { javaTotal = rt.totalMemory();
      javaFree = rt.freeMemory();
      String s = i +
        " fills the computer's Random Access Memory (RAM) reserved for the JRE";
      v.add(s);
      double percentUsed = 100*(1-(double)(javaFree)/(double)(javaTotal));
      if(previousTotal != javaTotal)
        System.out.println( i++ + "    total memory: " + javaTotal +
                              "    free bytes: " + javaFree +
                              "    Java's RAM used: " + percentUsed + " %");
      previousTotal = javaTotal;
    } } }

/*
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/JREClassLoading > javac Overflow.java
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/JREClassLoading > java Overflow
0 total memory: 2031616 free bytes: 1526576 Java's RAM used: 24.859028477822577 %
1 total memory: 2162688 free bytes: 581680 Java's RAM used: 73.10384114583333 %
2 total memory: 3211264 free bytes: 1349744 Java's RAM used: 57.96845105229591 %
3 total memory: 3473408 free bytes: 604272 Java's RAM used: 82.60290757665094 %
4 total memory: 5398528 free bytes: 2248848 Java's RAM used: 58.34331136191199 %
5 total memory: 5660672 free bytes: 609304 Java's RAM used: 89.23618962554269 %
6 total memory: 9109504 free bytes: 3777592 Java's RAM used: 58.53130971785072 %
7 total memory: 9502720 free bytes: 678280 Java's RAM used: 92.86225417564656 %
8 total memory: 15851520 free bytes: 7048872 Java's RAM used: 55.531885901162795 %
9 total memory: 15982592 free bytes: 1202328 Java's RAM used: 92.4772652646079 %
10 total memory: 26574848 free bytes: 10888720 Java's RAM used: 59.02621907752774 %
11 total memory: 27230208 free bytes: 1764144 Java's RAM used: 93.52137155911552 %
12 total memory: 44576768 free bytes: 18178328 Java's RAM used: 59.22017495750252 %
13 total memory: 44707840 free bytes: 2223928 Java's RAM used: 95.02564203504352 %
14 total memory: 66846720 free bytes: 23214888 Java's RAM used: 65.27146283318015 %
Exception in thread "main" java.lang.OutOfMemoryError
<no stack trace available>
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/JREClassLoading >
*/
```

With this program can be seen the resilience and robustness of the Java Runtime Environment when running a resource-intensive program. The Java Runtime Environment itself can be configured to use larger parts of memory by specifying options like `java -Xms 128M -Xmx 128M className`. For an explanation and description of those features of the java-tool, see the   Java Platform API Specification [JPAPIS]¹⁹².

¹⁹²Also consult the TechTip of December 22 of the year 2000.

Security: Basic Concepts

Security: Basic Concepts — Consider a house: This house gets water, electricity, postal service and repair services from outside. Water, electricity and postage usually reach the house without shortage and interruption, which constitutes a sign of Quality of Service (QoS) on the side of the utility network. A key to the front door enables the user of the house to control access to the house. But the wish for safety often produces much more elaborate security measures, a double lock on the front door, an alarm system. Sometimes even resources like a refrigerator have an extra lock. At other times, repair-persons may enter the house, and are granted a very limited access to the house's resources (TV-set, antenna). The postage comes in paper-envelopes (a very weak security measure) to prevent people from peeking inside. And letters are usually signed by hand to prove their authenticity. To overcome increased security measures, often a corresponding increase of counter-security efforts are necessary, from a better lock-opener to a better social plot to get access to the house. — Many of the real-life situations can be transferred directly to security considerations on computer-platforms:

When buying a product the owner expects a certain quality of service. The notion **Quality of Service (QoS)** describes all those features and services which make the product functioning as expected. For example, a computer-platform (hardware and software) should contain enough memory and a sufficiently large space on the hard-disks to be of service for the intended task: A small platform, designed solely for writing letters, usually cannot be sensibly used for editing movies. So, the Quality of Service also depends on balancing the intended use and the actual usability of the product.

Quality of Service concerns itself with those standards which the product should meet to satisfy the customer's needs. One part of this concept investigates non-standard situations and how to keep the product going — this part of the concept that tries to anticipate unexpected and exceptional behavior — this part describes the safety of the product and the **security** of using it. As an example, consider a security computer-platform equipped with two independent hard-disks, one of which is regularly updated to be a copy of the other; this may not change the day to day impression of the Quality of Service; but if one of the hard-disks crashes, then this security measure will have enhanced the overall Quality of Service significantly.

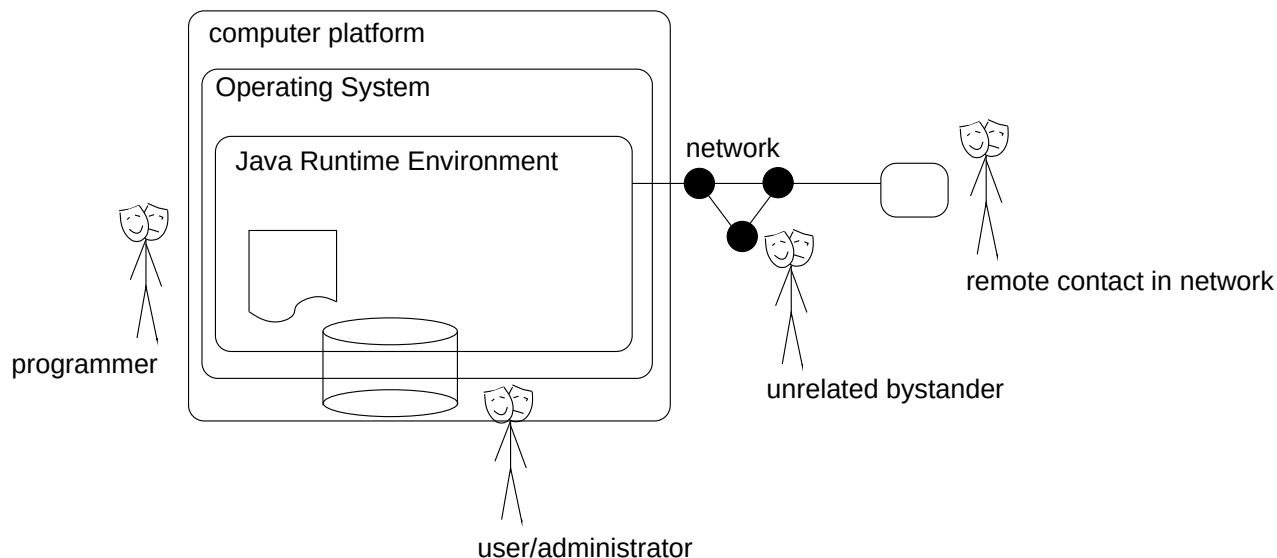
A **Quality of Service** point of view may ask: "How do I keep the system functioning properly in day to day work?" The **security** point of view asks: "What can go wrong with my (properly functioning) system and what counter-measures can be taken to keep the system working?"

As already could be seen, Quality of Service and security considerations for a computer-platform begin with the choice of the hardware. Another drastic measure to increase security of hardware is to lock it up in special rooms.

Security measures may be lax or "paranoid" depending on what damages can be done. Damages may come from so different states of mind like ignorance ("Nobody is interested in copying the data on this disk.") or challenge ("Am I able to copy the data on this disk?"). — In the following text are pursued only those security aspects that concern software-writing and -using.

The groups of people involved in deploying and using a software-product may be roughly characterized as follows:

- (system-programmer)/application-programmer
- end-user/administrator of the platform
- remote contact within a network
- unrelated bystander



The group of system-programmers writes operating-systems and (Java) runtime-systems, this group gives only little means to be controlled by the rest of the mentioned groups. All other groups may include ignorants and challengers. When considering software-security, a person in the remaining groups should be able to effectively protect hers or his work from destructive influences of the rest of them. Typical questions of group-members may be:

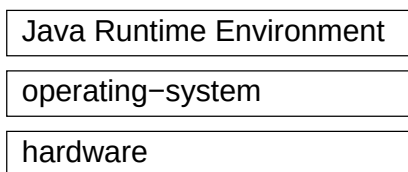
application-programmer Do I have an unnoticed error in my program? Can a user misuse my program, deliberately or accidentally?

end user/administrator That program shouldn't damage my data or my computer-system. Is the program easy to use, does it confuse me? Does it give others access to my data?

remote contact in network Is that really the person or an impostor? Does anybody collect copies of the data we exchange?

unrelated bystander Can I send an unnoticed program? Can I view or even change data? Can I enter the other's system? Can I take control over the system? Do they the same with me?

At this point, three levels of people's influence are to be stressed: The Java Runtime Environment (JRE), the (native) operating-system and the hardware.



All these levels have to furnish security features contributing to the overall security; a *single* problem on *one* of these levels may render inefficient the *entire* security of the platform: With a house, a locked main door becomes of little use, if the window has been left open.

(In-)Security is pervasive

Hardware access can be potentially damaging, especially if practicable for a user or unrelated bystander. For example: When upgrading hard-disks; often improperly erased hard-disks are thrown away while the data on them still can be read. A battery change may invalidate all security configurations which have been set by the operating-system.

Operating Systems may include a log-in procedure for users and other access-control mechanisms like granting access to files and directories (Unix-based operating-systems like Linux or Mac OS X do that, Mac OS Version 9.x and BeOS don't, they assume the computer-hardware to be properly secured). For a computer connected to a network, any person gaining control of the operating-system has the control of the entire computer-platform, this includes all groups mentioned above. A user may accidentally erase data from the hard-disk, an unrelated bystander or a known contact on the network may try the same if they, more or less accidentally, gain access to the operating-system of the user.

In a network, which connects different types of operating-systems (Unix-based, BeOS, Mac OS), a network-wide manipulation tends to become much more difficult.

Java Runtime Systems installed in operating-systems first depend on the safety of the hardware and the operating-system. If there are security deficiencies on the hardware side or on the side of the operating-system, then security features of the Java Runtime System may be easily rendered useless. A hex-editor may be used to read or write-into files that contain Java classes, even Java base-classes. It is useful to keep in mind that Java cannot easily solve those security-problems which are rooted on the levels of the operating-system, of the hardware or of the network(s).

Leaving aside these underhanded manipulations, provided the hardware and the operating-system provide sufficient security; the Java Runtime System together with the Java programming-language furnish their own security concepts. These concepts intend to make Java programming robust and the use of Java programs safer. These Java concepts then strengthen the overall security.

Almost all issues mentioned above focus on the notion of access, this aspect becomes the pivotal point of security considerations in any concept of a programming-language and -environment:

the meaning of access

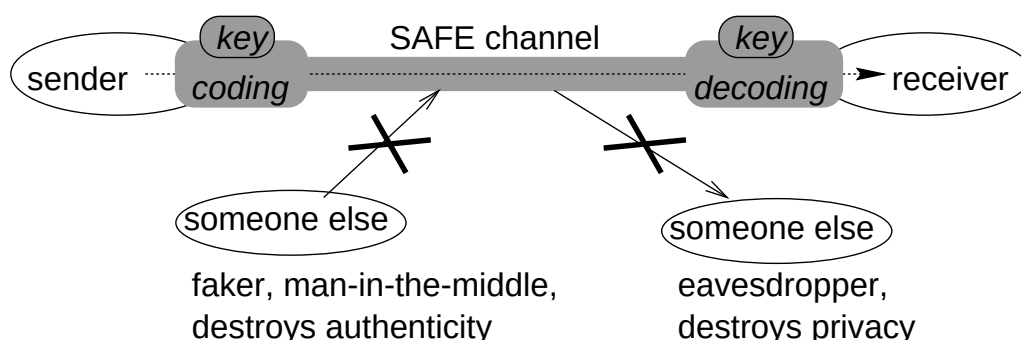
ACCESS: What or who grants which subject access to what resource?

Access and Protection are two aspects of the same thing: Discrimination. "To protect" or "to make exclusive" means giving an access-privilege or permission only to chosen few. The advantage of the positive view (giving access) is that a privilege forgotten to give, usually causes less damage than a protection which has been forgotten. (If all the doors and windows of a house are closed by default, then the closing of a little window in the back of the house is not easily forgotten.)

There are two fundamentally different kinds of access-control: One that restricts software-access to platform-specific resources like hard-disk, network sockets, monitor. The other method of access-control protects data and messages by establishing safe channels of communication. This means, by transforming data into byte-sequences which are generally difficult to re-transform by unprepared persons:

Access-Control with Processed Data External persons or data may have to use an electronic equivalent to a signature in order to identify themselves. Those external people may even need to encrypt their data to avoid being overheard by an unrelated bystander, producing a safe "logical channel". This requires to send specially processed data for acknowledgment by the receiving platform. (Further information about this give the sections about authentication and cryptography beginning with page 477)

processed (signed, certificated or encrypted) data

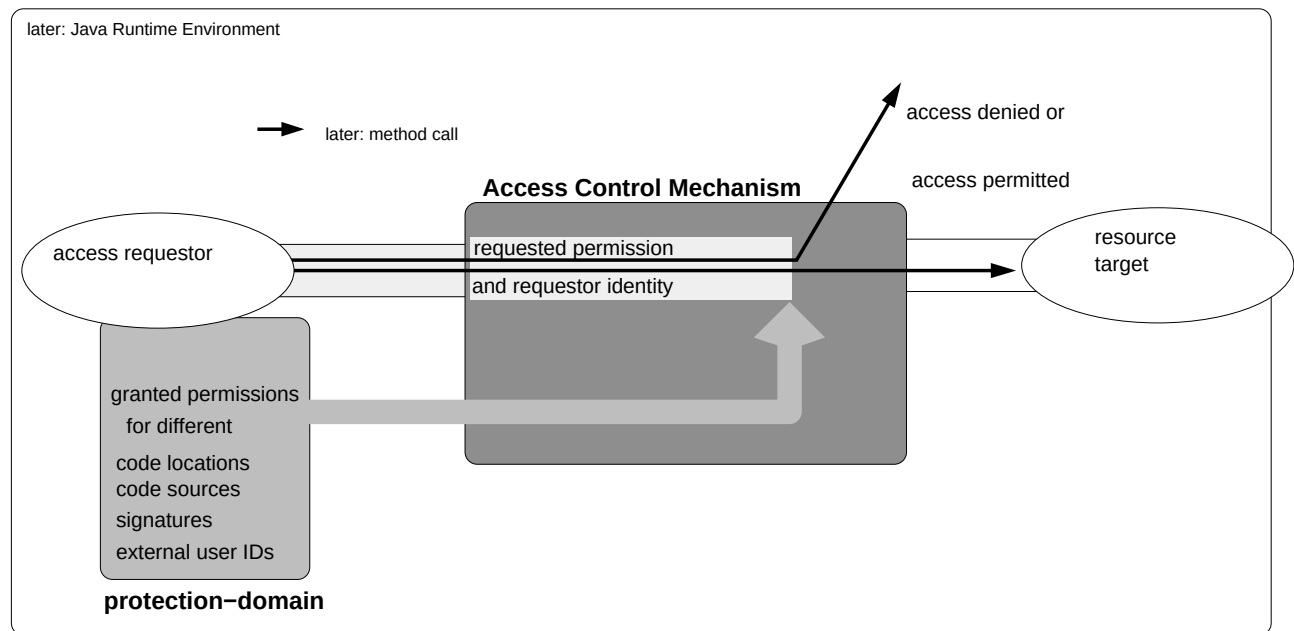


Policy-Based Access-Control Inside the computer-platform, known resources like a hard-disk could be screened from outside influence. According to the graphic below, the user or administrator could formulate an access-granting policy, which concerns those resources (hard-disk) inside the secured environment. Such a runtime-wide **policy** may grant different permissions to different privilege requesters (the set of information on such an access-requestor is also called calling-context). For a *given class*, or a *given object* derived from a class, the associated calling-contexts and the related permissions coming from the runtime-wide policy and class-specific class-loader are collected and called **protection-domain** of that class or object.

policy, gathers access-requesters and given permissions

The access-control mechanism, then automatically should be able to, first identify privilege-requesters by their calling-context, second COMPARE THE INFORMATION GIVEN BY THE PROTECTION-DOMAIN TO THE REQUESTED ACCESS, and third to decide whether the granted permission includes the requested permission. Finally, the access-control mechanism grants access to system resources (targets) or not.

protection-domain, gathers permissions for one access-requestor



The concepts above are realized as a working implementation in the Java Development Kit Version 1.2 and above.

Even the entire process of software-production, -implementation and -use can be accompanied by security-considerations: — **WRITING** of source-code can support security aspects by enabling the programmer to make the support relatively simple and easy. The software-developing-process itself can be improved for safety by using clear concepts and maybe a well-designed development-environment. (object-orientation, clear and easy to grasp implementation of concepts) — **TRANSFORMING** source-code to runnable code can include checks for the integrity of the code. (byte-code-verifier, page 451) — **RUNNING** the code may be accompanied by further checks for code safety. (access-control, authentication, encryption)

Java Language- and Java Compiler-Security

Java Language- and Java Compiler-Security — The Java programming-language and the process of translating Java source-code to runnable Java byte-code (compilation to byte-code) have been given various features to make Java programs robust and secure:

Simpler design no pointer of pointers; no class-extension from two or more classes at once, instead the use of Java-interfaces (page 136 and page 185); striving for conceptual clarity (comparing to C++)

Type safety requires the programmer to exactly know what kind of data is worked with. Identifiers of objects can only be type-cast to types of super-classes with respect to their initial class of definition (see page 169).

Automatic memory management a garbage-collector thread of the runtime-system relieves the programmer from considerations when to erase objects from the computer's Random Access Memory (an improvement compared to the programming-language C++ where deallocation usually has to be organized by the programmer) (see page 444).

Name-spaces defined by class-loaders and package-declarations. The name-space separation of loaded classes and derived objects can be loosened by access-modifiers `public` and `protected`. Even on that level, the programmer has to be actively concerned with communication of program-parts which reside in different name-spaces (page 239 and page 437).

Exception-mechanism for managing less expected situations without having to leave the program, or to structurally rewrite it (page 279).

Unchangeable constants/unoverridable methods/unsubclassable classes Variables declared with the keyword `final` have fixed values, once the values are assigned. This fix remains permanent for the time the Java Runtime Environment remains running. — Methods, which have been declared `final`, are unoverridable and are called faster, due to programmer-controlled early binding (see

page 168). — And classes declared `final` allow no sub-classing to take place. The Java programming-language means these declarations to be uncircumventable, giving an effective protection against a future manipulation.

In addition to the structural design of the Java programming-language, the Java compiler furnishes a dynamical checking-method for minimizing the risk that even syntactically sound source-code produces degenerate byte-code:

Byte-code-verifier checks byte-code before running according to whether the byte-code conforms to basic rules of safe conduct. The byte-code-verifier cannot cover all possibilities, but makes sure that essentialities of code-integrity are respected (for example, proper type-casting under various branching conditions). The byte-code-verifier is activated automatically before non-core-class code is run. Java Systems of version numbers 1.1 and 1.0, do NOT verify byte-code taken from the CLASSPATH-directories. But Java code can be explicitly byte-code-verified by using a “java”-command option as follows: “java -verify *fileName*” The byte-code-verifier remains invisible, until something goes wrong and the byte-code-verifier throws a `VerifyError`.

Java Runtime Security

Java Runtime Security — There still remain various security features acting especially during the runtime of Java byte-code.

Java Runtime Environment checks array-bounds and object-casting which results in throwing infamous `ArrayIndexOutOfBoundsException`-objects and `ClassCastExceptions`.

Runtime access-control mechanism as indicated by the graphic above, the implementation of a policy-secured environment. This is a part of the Runtime Environment like the exception-mechanism. The access-control mechanism can be customized using the `java.security`-package. This Application Programming Interface (API) will be introduced in the following sections. In its simple default version it isn't seen by the programmer and neither by the user. In a first step a user may change the default policy by editing a policy-file given in the simple ASCII-text-format. The Java classes `java.lang.SecurityManager` and `java.security.AccessController` are at the core of that mechanism. The necessary information about the loaded classes is given by the class-loaders.

Signing and encrypting means identifying and/or hiding data. This is done with tools (similar to the commands “javac” and “java”) by the user and alternatively by the programmer using an Application Programming Interface (API). A user is given Java-tools like `jar`, `keytool`, `jarsigner`, `policytool`. The APIs for the programmer include the `java.security`-package, the Java Cryptography Extension (JCE), and more¹⁹³.

¹⁹³Packages like the Java Secure Socket Extension (JSSE) or the Java Authentication and Authorization Service (JAAS) will not be treated in this text, the others will be introduced only.

Security: Policy-Based

Security: Policy-Based — First is given a general introduction to the Java-specific concept of access-control. Then the policy-based security-mechanisms are presented from **two differing perspectives**: From that of a Java-user and from the perspective of a Java programmer.

Java user and administrator use tools and edit text-files: This means adding security-options to the “java”-command and edit policy-files with a standard text-editor.

A Java programmer may instantiate an object of type `java.lang.SecurityManager`, use the service of the class `java.security.AccessController`, override associated `check...()`-methods and, in advanced cases, even handle objects of type `java.security.ProtectionDomain`. Rather seldom, this even may include to extend classes like `java.security.Permission`.

The following chapter should enable the reader to proficiently use the *basic* features of the Java access-control mechanism.

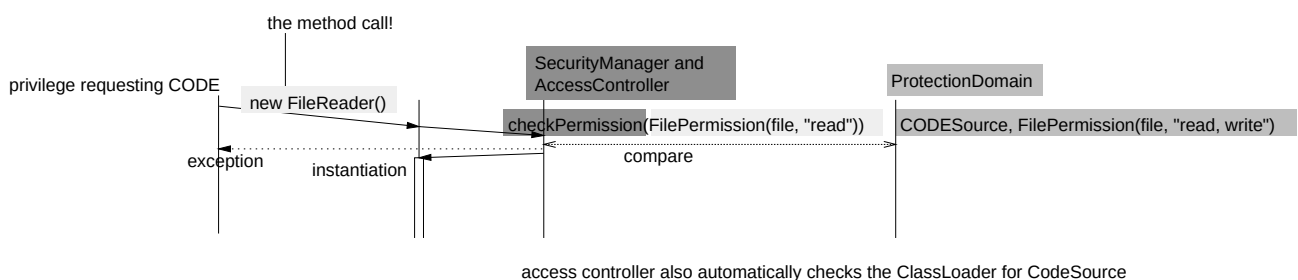
Java: Access-Control Mechanism

Java: Access-Control Mechanism — Upon request, the access-control mechanism uses a given policy (a list of accessors and their permissions) to release locks from resources like hardware or data. (Unlike access-control, cryptography tells how to lock data by transforming the data into apparently meaningless sequences of bytes, the meaning is recovered only by a proper re-transformation.)

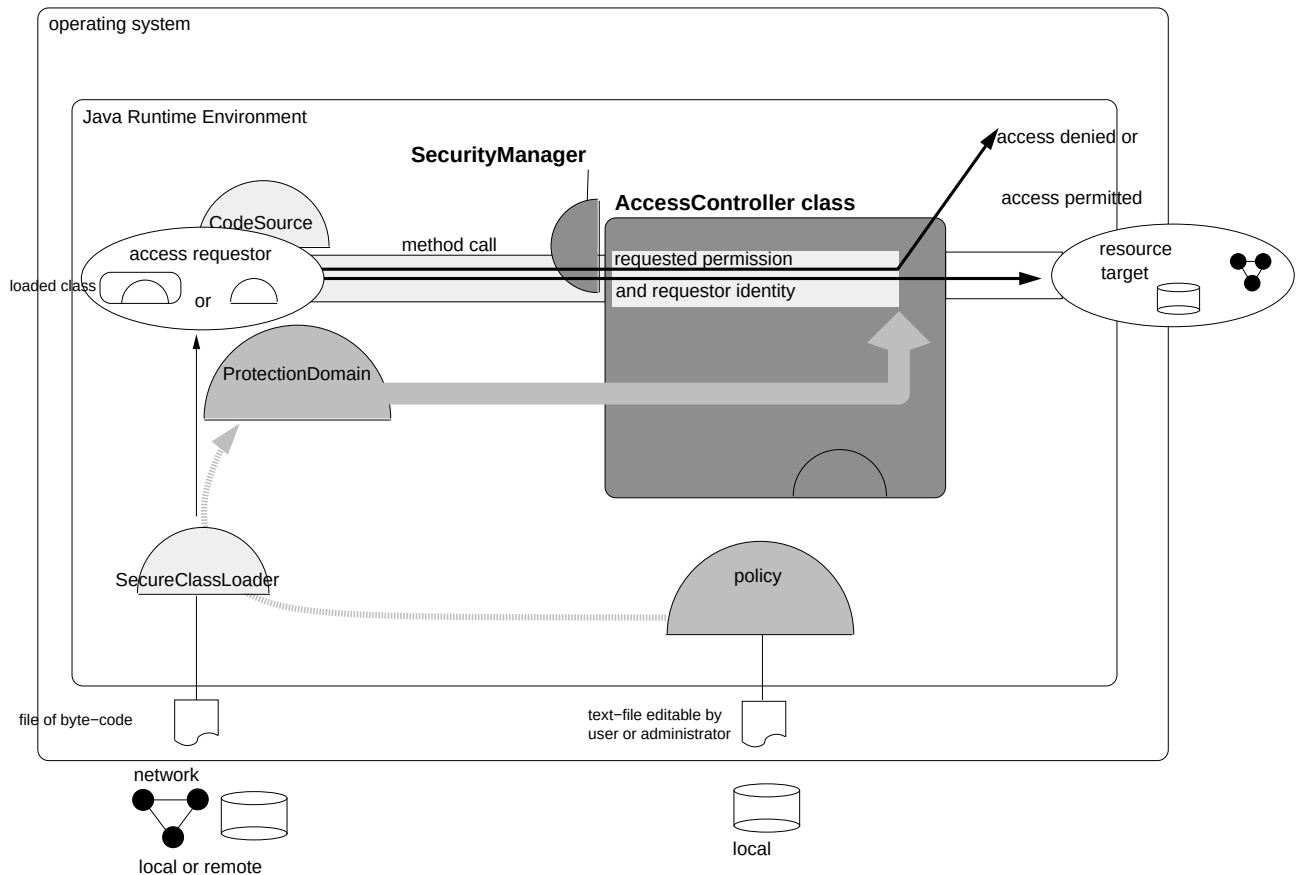
ACCESS-CONTROL-PROCEDURES FOCUS ON METHOD-CALLS DIRECTED TO ACCESS-CONTROLLED RESOURCES; for example while establishing a `java.io.FileReader`-stream-object, the Java Runtime Environment EITHER THROWS AN EXCEPTION OR RETURNS THE REQUESTED RESULT (without documenting any checks that have been done).

The essence of the access-control mechanism also can be described as follows: Method-calls of Java core-classes, which request the access to the system’s resources, immediately forward the call (over a `SecurityManager`’s `check...()`-method) to the `AccessController`’s method `checkPermission( permissionRequested )`. The information about the calling context is used by the access-controller to consult a protection-domain to produce the granted permission: *permissionGranted*. If the *permissionGranted* is equal to or implies the *permissionRequested*, then THE METHOD-CALL RETURNS SUCCESSFUL; IF NOT, AN EXCEPTION IS THROWN.

access-control:
either access or
exception



The class `java.security.AccessController` compares `Permission`-objects: The `Permission`-objects that are given by the call-dependent `checkPermission()`-methods with those `Permission`-objects inside the requestor-specific `ProtectionDomain`-object. The graphic below gives an impression of the different classes and objects taking part in access-control-processes:



The source of all the access-permissions within a given Java Runtime Environment represents one single Policy-object within the Java Runtime Environment, this object gets its information from one or several text-files as will be described in sections below. On loading a class, any secure class-loader automatically distills the relevant access-permissions from the Policy-object into an object of class-type Protection-Domain and associates these permissions with a CodeSource-object: Any information about the origin (URL, signers) of the loaded class is automatically encapsulated within a CodeSource-object. Thus, the objects of class-type `java.security.CodeSource` and `java.security.ProtectionDomain`, which are produced by a class-loader of type `java.security.SecureClassLoader`, accompany any such loaded class, or objects derived from it, like guards. The AccessController compares the requested permissions (coming from method-calls of the loaded class and the instantiated object) to given permissions (coming from the related protection-domain).

Security-managers are singletons, and once instantiated, they have become an immutable part of the current Java Runtime Environment. If there exists a security-manager within a Java Runtime Environment, then the policy-based access-control-mechanism has been activated. As already described in the paragraphs facing the graphic above, if there exists a security-manager, then method-calls of Java core-classes automatically consult the security-manager for a check. The security-manager delegates the policy-oriented checking to the class `java.security.AccessController`. The loaded class' or instantiated object's method-call of the Java core-class implies some requested access-permission: The access-controller compares the requested access-permission to the permissions granted, taken from the protection-domain of the loaded class or its instantiated object. The result of that comparison can be seen in the graphic above, either the access is granted and the method-call proceeds as usual, or access is denied and an exception is thrown.

End-users and application-programmers usually do not notice anything of this security infrastructure, until a `java.security.AccessControlException` is thrown by the access-controller.

Secure Class-Loaders and Protection-Domains

Secure Class-Loaders and Protection-Domains — A secure class-loader takes all the relevant permission-entries of the Policy-object and usually adds some of its own, like the permission for reading-access to the code-source of the loaded class. All this access-requestor-specific information is collected in an object of class-type `java.security.ProtectionDomain`.

Once a class has been loaded, its protection-domain is fixed, even if the Policy-object is changed afterwards. So, when a protection-domain has been instantiated, it cannot be changed within that session of the Java Runtime Environment. Only the classes that are loaded after the policy-change occurred, have an associated protection-domain that uses the new policy.

Infrastructure-programming covers the implementation of classes like `ProtectionDomain`, `Policy`, `Permission`, `Permissions`, `PermissionCollection` and the `policy.provider`-file, such a work usually has very little to do with any applicational problem.

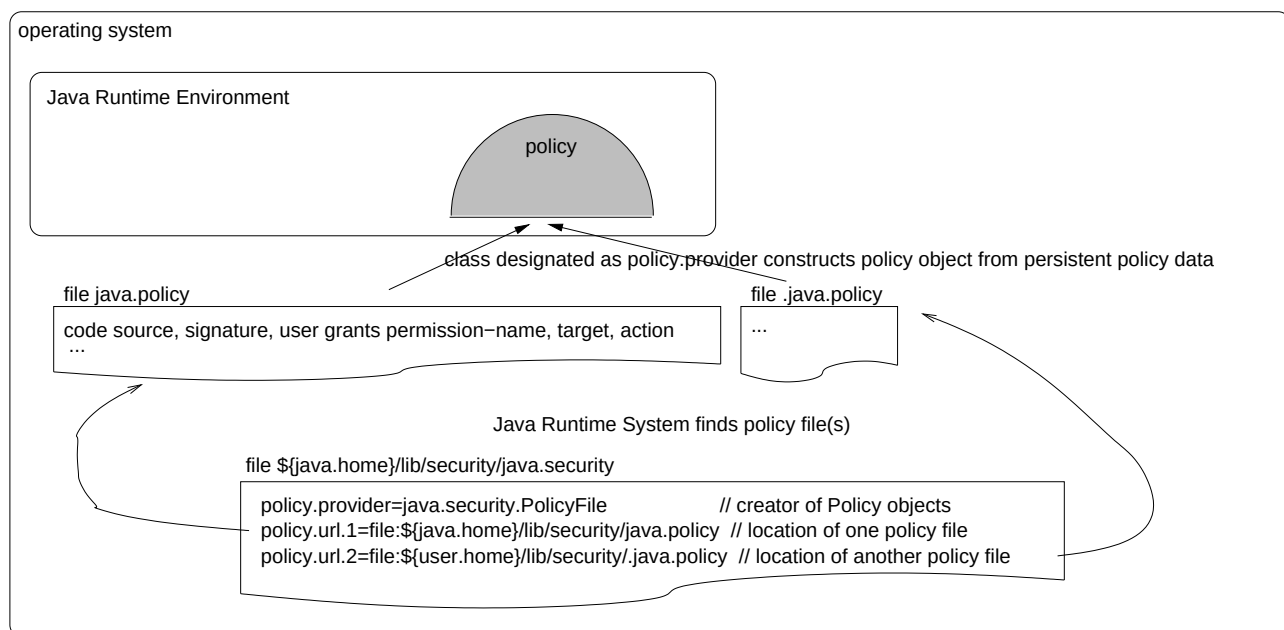
From File to Policy-Object

From File to Policy-Object — Initial and therefore sensitive information about the general security settings of the Java Runtime System is stored in an ASCII-coded text-file¹⁹⁴ identified by the URL:

```
file:${java.home}/lib/security/java.security
(editable by users and administrators)
```

When starting the Java Runtime Environment, this text-file tells (see the entry `policy.provider= ...` below) what class is used to automatically create the initial Policy-object from what policy-text-file. So-called policy-files are simple human-readable text-files for persistently documenting the permissions for individual privilege-requesters (they can be edited by users and administrators).

The class `sun.security.provider.PolicyFile`, named `policy.provider` in the file `java.security`, automatically transforms the indicated files to a `java.security.Policy`-object. (Programmers may substitute the provider with their custom-written one, which develops into a rather specialized task, involving the closer knowledge of classes like `ProtectionDomain`, `CodeSource`, `Permissions`, `PermissionCollection` and `Permission`. This closer knowledge is not given in this text.)



So, the Java Runtime Environment allows access to its resources according to the rules found in text-files: When starting the Java Runtime Environment, the permissions in these text-files are transferred into the instantiated Policy-object. If necessary, the active Policy-object can be changed during the existence of the Java Runtime Environment. The Policy-object is consulted by secure class-loaders, which - for each loaded class - extract the relevant permissions and collect them, together with information about the code-source, in an associated object of class-type `ProtectionDomain`.

Security-Manager and Access-Controller

Security-Manager and Access-Controller — THE ENTIRE ACCESS-CONTROL-PROCESS RELIES ON ONE OBJECT AND ON ONE CLASS:

¹⁹⁴Text-files coded in ASCII-format are one of the simplest means to store human-readable data.

1. The `java.lang.SecurityManager`-object instantiable only once in a Java Runtime Environment's lifetime. A programmer can instantiate a security-manager or a user can let the system install a default security-manager by using an option of the "java"-tool (page 461). Method-calls of Java core-classes are pre-programmed to automatically first call a `check...()`-method of the `SecurityManager`-object. Beginning with Java Version 1.2, these methods usually forward these calls to the access-controller:
2. Calling the `checkPermission()`-method of the `java.security.AccessController`-class makes this class compare permissions assigned to method-calls with permissions coming from protection-domains.

Memorize that **JAVA CORE-CLASSES USE THE (INSTALLED) `SecurityManager`-OBJECT FOR CHECKING WHETHER THEIR METHOD-CALLS ARE PERMITTED OR RESTRICTED.** If access to resources has been granted, user or application -programmer are not notified; only if something goes wrong an exception is thrown. The `SecurityManager`-object forwards those calls to the `AccessController`-class for evaluation with the access-requestor-specific object of type `ProtectionDomain`. **WITHOUT A `SecurityManager`-OBJECT (INSTANTIATED EITHER BY THE USER OR THE PROGRAMMER) THE METHOD-CALLS OF JAVA CORE-CLASSES ARE NOT CHECKED FOR ANY SECURITY-POLICY.**

method-calls of
Java core-classes
are forwarded by
the
security-manager
to the
access-controller

As an example of further investigation consider the following program. This piece of source-code is meant to read a file on the hard-disk (In contrast to the other program on page 298, this program here reads entire lines from the buffered stream.):

```
public class Merge
{
    static void fileToConsole( String fileName )
    { try
      { java.io.FileReader fr = new java.io.FileReader( fileName );
        java.io.BufferedReader br = new java.io.BufferedReader( fr );
        String s;
        while( true )
        { s = br.readLine();
          if( s == null ) break;
          System.out.println(s);
        }
        br.close();
      }catch( Exception e ){ e.printStackTrace(); }
    } // end method

    static void merge( String source1AndDestination, String source2 )
    { try
      { java.io.FileWriter fw = new java.io.FileWriter( source1AndDestination, true );
        java.io.FileReader fr = new java.io.FileReader( source2 );
        java.io.BufferedReader br = new java.io.BufferedReader( fr );
        String s;
        while( true )
        { s = br.readLine();
          if( s == null ) break;
          fw.write( s, 0, s.length() );
        }
        br.close(); fw.close();
      }catch( Exception e ){ e.printStackTrace(); }
    }

    public static void main( String fileNames[] )
    { int toBeMerged = fileNames.length;
      switch( toBeMerged )
      { case 2: merge( fileNames[0], fileNames[1] );
        case 1: fileToConsole( fileNames[0] ); break;
        default: System.out.println("Merge: enter one or two file-names");
      }
      try
      {
          System.out.println( System.getProperty("java.version") );
          ClassLoader cl0 = Class.forName("Merge").getClassLoader();
          System.out.println(cl0.getClass().toString());
      }catch( Exception e ){ e.printStackTrace(); }
    } } // class and main()-method
}

[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% javac Merge.java
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java Merge DemoFile.txt Merge.java
This is the text of the demonstration-file.public class Merge{ static void fileTo .... // class and main()-method
1.3.0
class sun.misc.Launcher$AppClassLoader
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%
*/
```

Who can how influence the access-control mechanism when using this program?

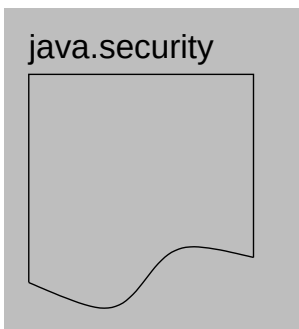
Java User and Administrator

Java User and Administrator — The user or administrator may wish to use applications, applets, servlets or Java-Beans in a productive way; at the same time these pieces of software should be hindered to misuse the platform on which they run. Java versions below Version 1.2 furnished little support to handle problems of that kind; and some Java-enabled browsers or older appletviewers produce a Java Runtime Environment which is not user-controllable.

Beginning with Java Version 1.2, a user or administrator can establish rules according to which the Java Runtime Environment runs byte-code. The user or administrator furnishes policies by writing them down in ASCII-text-files and registers them in the `java.security-text-file`¹⁹⁵. Options of the `java-` and `appletviewer-tools` then allow the user to control the use of these policy-files. With applications, the user or administrator activates the entire access-control-mechanism by using the option of installing a default security-manager (`java -Djava.security.manager applicationName`).

policy-text-file
registered within
the java.security-
text-file
AND
tool options
activating
security-
management

Text-File: `java.security`



Text-File: `java.security` — When a user or administrator starts the Java Runtime Environment by issuing the command “java”, then the text-file named `java.security` is read for establishing basic security settings: What classes provide cryptographic services, what class provides the policy-information, possibly to be read in from places given by Uniform Resource Identifiers and more.

On the level of the operating-system the administrator or the user can use a simple text-editor to edit the ASCII-text-file named `java.security`. Of course, also on the level of the operating-system, the `java.security-text-file` has to be protected from unauthorized access; otherwise this text-file might be easily altered or replaced.

The following variant of a Uniform Resource Locator gives the standard location of the `java.security-file`; the placeholder `${java.home}` has to be substituted automatically by the directory-path in URL-notation (URL-encoding, for example, requires slashes as folder-separators, see page 375) leading to the directory in which the files of the Java Runtime System have been installed. For example, the placeholder `${java.home}` may be substituted by the path `/usr/lib/jdk1.3/jre`:

```
file:${java.home}/lib/security/java.security
```

This ASCII-text-file may contain documenting comments to be read by the user, indicated by a “sharp” symbol `#` at the beginning of each of these lines. — The rest of the lines indicates settings to be read by the Java Runtime Environment: The default cryptographic class is given by the line

```
java.security
```

```
security.provider.1=sun.security.provider.SUN
```

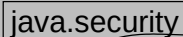
This qualified class-name specifies the default cryptographic service provided with the Java Runtime System. (The class is found inside the archive-file `rt.jar` under the prefix `sun/security/`. More on page 485) — The default class which is used to instantiate the first `java.security.Policy-object`, after the Java Runtime Environment has been started, is given by

```
java.security
```

```
policy.provider=sun.security.provider.PolicyFile
```

On starting a Java Runtime Environment (JRE, Version 1.2 or higher), this class (`PolicyFile`) reads the policy-defining text-files below, to generate the `Policy-object` for the JRE. (This represents a first step in the process of establishing customized access-control enforcement in the Java Runtime Environment: The source of policies being persistent text-files (below), written by user or administrator, then a `PolicyFile` transforming the policy-information into a `Policy-object`, after that `SecureClassLoaders` using that `Policy-object` to produce objects of type `ProtectionDomain`, which finally are used by the access-controller.)

¹⁹⁵ Again recall that thus the entire access-control mechanism rests on the effectiveness of the operating-system in protecting unwanted access to the `java.security-text-file` and the associated `policy-text-files`.



```
policy.url.1=file:${java.home}/lib/security/java.policy
```



```
policy.url.2=file:${user.home}/.java.policy
```

When reading the `java.security`-file the Java Runtime Environment replaces the placeholder `${java.home}` by the value of the system-property `java.home`. (The text-files above are: `java.policy` for the system-wide policy and `.java.policy` in the home directory of the user¹⁹⁶.) A user or administrator may change the entries or add further Uniform Resource Locators (URLs) by inserting another line, for example:



```
policy.url.3=http://www.somedomain.com/somefile.txt
```

These entries have to be numbered consecutively beginning with number 1 (No, chime-like counting as commonly used, see also footnote on page 234). Policy-files contain simple text, the conventions are presented in the next paragraphs.

The automatic substitution-process (`${system.property}` to `text`) is also called **property-expansion**. Property expansion is used when the Java Runtime Environment is started and the `java.security`-file together with the indicated policy-files are automatically read. In these ASCII-files any tokens of the form `${...}` are scrutinized for their content and if the name of a system-property is found, then the value of the system-property is inserted. Next to properties like `java.home` and `user.home`, the property `file.separator` can be abbreviated by using a simple slash `/` inside the dollar-prefixed-braces; so the directory-paths can be indicated independently of the current operating-system.

The automatic property-expansion can be disabled by changing the entry `policy.expandProperties` to `false`:



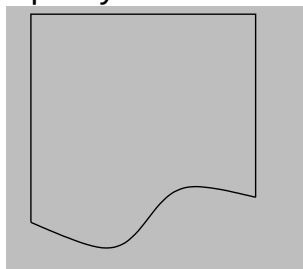
```
policy.expandProperties=false
```

With this, all strings in the ASCII-files are read as literals, this effectively means that NO property-expansion will occur.

Property-Expansion: Values of `→` system-properties are inserted automatically into `java.security`- and policy-files like `java.policy` when starting a Java Runtime Environment.

Policy Text-Files

policy file



Policy Text-Files — Like the `java.security`-file, a policy-file is an ASCII-text-file and can be edited by user and administrator with a standard text-editor¹⁹⁷. Those policy-files are used for persistently storing the access-relations of privilege-requesters and resources. The Uniform Resource Locator of the default policy-file is given by

```
file:${java.home}/lib/security/java.policy
```

When a Java Runtime Environment is started on the given platform, this policy-file and any additional policy-files that are specified in the `java.security`-file are automatically read and processed together by the class with the identifier `sun.security.provider.PolicyFile`, the so-called default policy-provider. The default policy-provider automatically instantiates an object of

class-type `java.security.Policy` which itself will equally automatically be referenced by secure class-loaders. Like any class-loader, those secure-class-loaders load classes into the current Java Runtime Environment; but secure-class-loaders usually add information (source of the code) of the loaded classes to the policy-information and turn this into a requestor-specific object of type `ProtectionDomain`. Then those loaded classes and their derived objects are permitted or denied access to the system's resources by the class `AccessController`. This class analyzes the permissions given by the protection-domain and the permissions required for successfully issuing a method-call. – Thus information taken from a policy-file undergoes multiple transformations until effectively used in access-controlling (policy-file $\rightarrow$ Policy-object $\rightarrow$ ProtectionDomain-object).

Administrator and user need only to edit the policy-files according to their wishes; typical entries are:

¹⁹⁶Note the leading period (`.`), making it an “invisible” file in Unix-based operating-systems.

¹⁹⁷Besides editing policy-files with an ASCII-editor, they can be edited more comfortable with a tool furnishing a Graphical User Interface (GUI): Entering `policytool`, into a terminal-window of the operating-system, starts a more comfortable way of formulating policies.

policy file

```
grant codeBase http://www.thePrivileged.com/privilegedFolder/
{ permission java.io.FilePermission
  "${user.home}/${/}folder1${/}folder2${/}accessibleFile"
  "read, write, delete, execute";
};
```

Above is shown how code, having been loaded from some “privilegedFolder” elsewhere, indicated by the codeBase above (the codeBase-information can be produced by asking the code’s class-loader for a `java.security.CodeSource`-object) is granted unlimited access to a file with the path `${user.home}/folder1/folder2/accessibleFile` (on a Unix-based platform) or the path `${user.home}\folder1\folder2\accessibleFile` (on a Windows-based platform).

NOTE and CAUTION: The **codeBases** are given as Uniform Resource Locators (**URLs**) in URL-encoding and format, BUT the **permission**-related locators are given as **paths** within the file-system of the native operating-systems, generalizable by using **property-expansion**. — Another entry may read:

policy file

```
grant
{ permission java.net.SocketPermission "localhost:8080" "listen"; };
```

codeBase as URL
and permission
as property-
expandable
HFS-path

If the current platform is connected to an IP-network, then the line above allows *everyone* in the network to listen to the current platform on TCP/UDP-port 8080. No codeBase-specification means everyone has been granted the permission.

Exhaustive and up-to-date information about the specifics of the individual permission-classes gives the ➡ Java Platform API Specification [JPAPIS] of the `java.security`-package. As already could be seen, the policy-files associate privilege receivers with permissions. The general syntax, including some of its semantics, is shown below: (brackets [...] indicate *one or none* occurrences of the enclosed entity):

policy file

```
grant [ signedby signerAliases, ] [ codeBase URLOfPrivilegedCode ]
{ permission PermissionClassName
  [ targetIdentifierOrHFspath [ listOfActions ] ];
  [ otherPermissions ]
};
```

Outside the braces “...{ }” is collected the information about the privilege-requesting code, inside the braces “{...}” are collected the permissions that are given to the code on the current platform:

signerAliases of public key in the key-management-system. For more on the use of signed data see page 492.

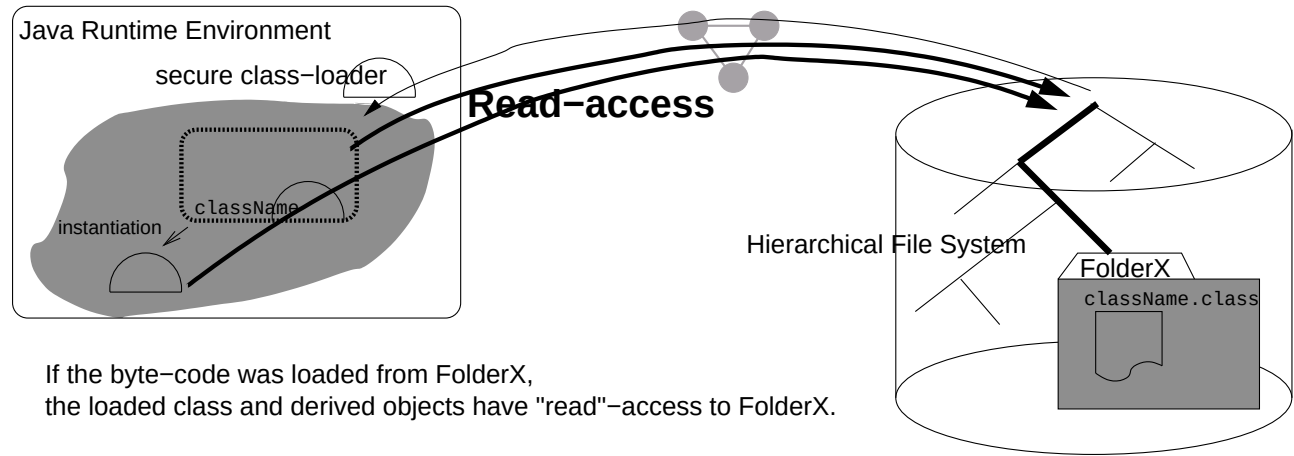
URLOfPrivilegedCode The Uniform Resource Locator (URL) of the loaded code that is to be given privileged access.

PermissionClassName The package-qualified name of the `Permission`-class, that characterizes the permission to be given.

targetIdentifierOrHFspath A string with which most permissions need to fix the name of the resource of the current platform, this may be a path in the local file-system (property-expanded, if necessary), or the name of another service the current platform furnishes.

listOfActions Permissions that name a native resource (like a file or a socket) often use a string listing comma-separated functions available to the privileged code (reading, writing, listening and more). Some permissions, like the `java.security.AllPermission`, are not associated with any actions, neither with a target.

Ideally a **BLANK POLICY-FILE FORBIDS ANY ACCESS** to any of the system’s resources, with the following **exception**:



any loaded code
has
"read"-access to
its code-source

EXCEPTION: Code automatically has permission to read files from the location of its own code-source. With code-source is meant the folder which contains the class-file of the byte-code of the loaded class and its derived objects. "Read"-access means especially: **Files of Java byte-code received and stored within some folder, when loaded, produce a class within a Java Runtime Environment (JRE), that even under the tightest security restrictions, retains a permission to "read"-access the folder from where the code has been loaded.** So for programs loaded by the standard class-loader, this rule resembles an implicit permission of the kind

grant

codeBase="file:PathToCode(in URL-format)"

{ permission java.io.FilePermission "PathToCode(in platform-format)-", "read" }

Platform(-specific)-format means: HFS-paths possibly generalized by the use of constructs for property-expansion (HFS=Hierarchical File System). For more on the background¹⁹⁸ also see page 469.

The **FOLLOWING ENTRY PERMITS EVERY IMAGINABLE ACCESS** and its activation is equivalent to running the code with no security-checks at all:

policy file

```
grant { permission java.security.AllPermission; };
```

If one policy-file of the policy-files from which the `java.security.Policy`-object is compiled, contains the line above, then all other permissions remain intact but are effectively overridden by the "carte blanche" above.

The **DEFAULT POLICY FOR APPLICATIONS** can be seen by looking at the policy-files that come with the Java Runtime System; the **DEFAULT POLICY FOR APPLETS** often is installed unchangeably into browsers, the `appletviewer`-tool allows a similar configuration of the policy for applets like that for applications. Exactly this is the point where Java versions beginning with Number 1.2, do considerably reduce the effective difference between applets and applications: Applets, implicitly within their applet runtime-environments, always use a security-manager. Whereas applications still can be run without any `SecurityManager`-object in place.

basic difference
of applets and
applications

Selecting Policies for Running Applications and Applets

Selecting Policies for Running Applications and Applets — Administrators and users with access to the `java.security`-file have some control over the Java Runtime Environment (JRE).

- **How to run a program free from any policy-checks:** Generally, work without a security-manager¹⁹⁹: Run an application, that does not install a security-manager, with the simple command `# java applicationName`. Then the application runs without being checked by a policy. An `appletviewer` automatically instantiates a security-manager, so applets are always checked by some default- or even custom-policy.

¹⁹⁸This access exception can be disabled by overriding the `checkRead()`-method of the security-manager. For more information see the paragraphs about the security-manager, especially on page 463.

¹⁹⁹Seldom, the program may call the access-controller directly, then security-checks are possible without a security-manager installed.

The option `-DsomeSystemProperty` of the `java`-tool invokes a default system-property given in some property-file. The option `-DsomeSystemProperty=someValue` assigns the indicated value to the system-property for this invocation of the Java Runtime Environment (JRE).

- **How to specify default policy files for the Java Runtime Environment (JRE)** has been shown on page 457.

- **HOW TO RUN A PROGRAM WITH THE DEFAULT POLICY FILES:** Use the “`java`”- or “`appletviewer`”-commands with the option `-Djava.security.manager`:
`# java -Djava.security.manager applicationName`
 This ensures a check according to the default policy.

A standard way to start the entire access-control mechanism!

- **How an administrator can enforce the use of the default policies:** Resetting the entry `policy.allowSystemProperty` in the `java.security`-file to false, disables the changing of system-properties via the use of command-line options:

```
java.security
```

```
policy.allowSystemProperty=false
```

If the administrator is the only one who can edit the `java.security`-file and the policy-files, then the administrator has the ultimate control over the behavior of the Java Runtime System!

- **How to run a program with an EXCLUSIVE user choice of policy:** Use the “`java`”- or “`appletviewer`”-commands with the option `-Djava.security.manager` and `-Djava.security.policy==URLofPolicyFile`:
`# java -Djava.security.manager -Djava.security.policy==URLofPolicyFile applicationName`
 This ensures a check according to the policy chosen by the user.
- **How to ADD a user choice of policy to the default policies:** Use the same command as in the case above, but instead of the double equal-sign “`==`”, use a single equal sign “`=`”!

The answers to the questions above outline the possibilities found to change the behavior of the entire Java Runtime Environment when running programs. For more on the commands like `java` or `appletviewer` see the   Java Platform API Specification [JPAPIS], that can be downloaded together with a Java Development Kit, the information therein should be extensive and up-to-date.

The file-merging program on page 456 can be tested using different policy-files configured by the user. Observe that an exclusively used empty policy-file represents the strictest possible policy giving no permissions; nevertheless the contents of a file inside the byte-code’s folder can be read: The reason for this is given by the above-mentioned STANDARD PERMISSION, WHICH LETS CODE READ FROM FILES WITHIN ITS CODE-SOURCE, in this case from the current system’s current directory! A non-permitted try of getting access produces an exception of the kind:

```
java.security.AccessControlException: access denied (java.io.FilePermission fileName write)
```

Some Practical Hints

Finally some practical hints for configuring policy-files: Usually a program is written to work without any respect to a policy.

- First make the program run in a satisfactory way when using the command “`java className`”.
- Second try `java -Djava.security.manager className` and see what happens. If the `.java.policy`-file doesn’t grant any permission, then test the proper program-run by granting permission to everything with: `grant{ permission java.security.AllPermission;;` Under default circumstances this should recover the normal program-behavior.
- Now test for normal program-behavior by narrowing the given permission to selected code-bases:
`grant codeBase`
`"file:${user.home}/Desktop/Tests/-" { permission java.security.AllPermission;;`
- If the program still works, then make the permissions more specific for example with:
`grant codeBase "file:${user.home}/Desktop/Tests/-"`
`{ permission java.io.FilePermission "/home/andreadipietro/Desktop/-", "read,write";`
`};`

Usually the exceptions thrown by the access-controller present a good guideline. Simple keyboard-typing errors may not be noticed properly by the Java system, note also that the `codeBase` needs a *Uniform Resource Locator*, whereas some permission-targets are *simple paths* maybe using property extension. Especially, if the policy-file is to be used under differing operating-systems, then the file-separators may be substituted by `$/{}`. A line of source-code (anticipatedly taken from the program on page 468) inserted into the program in question lists the currently valid permissions for a given class, named *className*:

```
System.out.println( Class.forName( className ).getProtectionDomain() );
```

During runtime, a standard permission marked unresolved usually indicates *typing errors*. The Java Runtime Environment may react seemingly confused to any unsystematical approach.

Java Programmer

Java Programmer — User and administrator may edit text-files (`java.security`, `java.policy`, `.java.policy`) and issue commands in terminal-windows (“`java -Djava.security.manager className`”); thereby using command-line options to control the programs, as could be seen in the sections before.

How can a Java application-programmer influence the security-policy from within the Java source-code? How does the Java application-programmer relate to the classes `java.lang.SecurityManager`, `java.security.AccessController`, `java.security.Policy` and objects of the type `SecurityManager` and `Policy`? How can those classes and objects be used in Java application-programming and what services do they provide? — First remember that the basic functions of these objects and classes remain the same as described in the context of the graphic on page 454. The central points of decision – on giving or on denying access – remain an *object* of type `SecurityManager` and the `AccessController`-class.

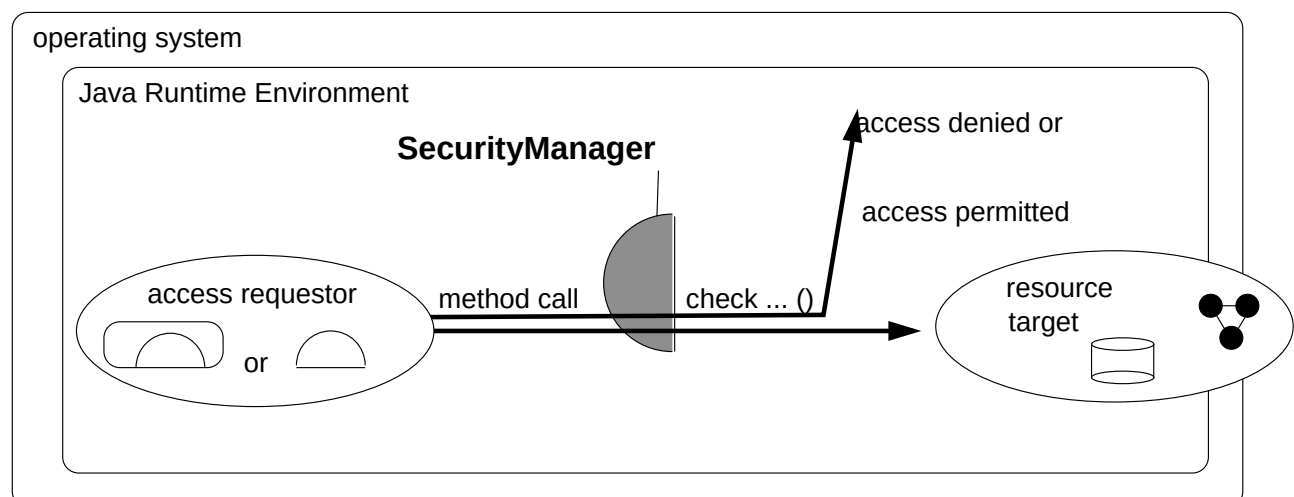
Java Security-Manager

 **SecurityManager** — Java core-classes, which operate on system resources like the file-system, the network-connection, the Java Runtime Environment (JRE) and threads, use methods which have been made access-controllable: The first thing an access-controlled method does, after having been called, is to call a corresponding `check...()`-method of the security-manager²⁰⁰. When the security-check has been successful, the `check...()`-method returns without further notice

and the rest of the access-controlled method is run. — Whereas an unsuccessful security-check triggers a (runtime-)exception to be thrown and thus stops the method-run before it really began.

Compare the content of the previous sentence with the graphics on page 454. In the previously referenced graphic can be seen that the security-manager (an object of class-type `java.lang.SecurityManager`) delegates the access-decision to the class `java.security.AccessController` (Java Versions 1.2 and higher). Or a security-manager may decide on throwing a security-exception completely by itself (mainly in Java versions below Number 1.2) as shown in the graphic below.

This fully automated process within the Java Runtime Environment (JRE) constitutes the core of the policy-oriented security-measures in Java (Java language-concept and Java Runtime Environment).



²⁰⁰Thus, in this case, each `check...()`-method of the security-manager represents a triggerable method (page 127). Nevertheless a programmer still may be free to call such a method directly or to use it in a method-definition of one's own.

Users and administrators activate the security-checking in different ways: Among other things, the `appletviewer`-command automatically gives Java Runtime Environment a security-manager. Alternatively, the `java`- and the `appletviewer`-commands may furnish the Uniform Resource Locator (URL) of a `SecurityManager`-class. Or finally, the *Java application-programmer* may install a security-manager from out of the Java source-code by calling the `java.lang.System.setSecurityManager()`-method:

Activating
security-checking

Each Java Runtime Environment has instantiated either **NONE (NULL) OR ONE** single object²⁰¹ of type `java.lang.SecurityManager`. **ONCE INSTALLED, A SECURITY-MANAGER CANNOT BE CHANGED** during the lifetime of that Java Runtime Environment, neither by the Java programmer nor by the user or administrator! A Java Runtime Environment that is restrained by a security-manager is also called **sandbox**.

sandbox

Security-Sandbox: 1. Any computing-environment that furnishes security features.

2. A Java Runtime Environment (JRE), enriched by a security-manager possibly delegating decisions to an access-controller (access-controller available and used with Java Versions 1.2 and higher). Any program running in that environment can be restrained from accessing resources (no access to local file-system, only read-access to certain folders, only network connections to prescribed URLs ...).

3. The security-environment, constructed with the help of a security-manager for running applets, the applet runtime-environment.

Most methods of the security-manager are methods named `check...()`, the name indicates what resource is checked by the automatic call from access-controlled methods.

For example, the constructor of a `java.io.FileReader`-stream-object (calls a constructor given within the class `java.io.FileInputStream` which in turn) first calls the `checkRead()`-method of the `SecurityManager`-object, if existent. A non-existent security-manager causes the `checkRead()`-method to return immediately without any further notice. As another example, consider all those non-void methods of the `java.io.File`-class, after having been called, they first call the `checkRead()`-method of an existent security-manager before they continue to run.

A Java programmer may define a custom security-manager by extending the default `SecurityManager`-class and overriding the relevant `check...()`-methods. The extreme cases of overriding have been described on page 255.

Why should a programmer restrict the freedom of the own program by installing a security-manager? The program may grant the user (for the user's own safety) only restricted access to resources or the program may load external classes that are to be controlled. Furthermore, the source-code may become useful in proving the program to be non-involved in any damage done to the computer-system.

self-inhibiting
Java programs?

The `check...()`-Methods of the Security-Managers

The `check...()`-Methods of the Security-Managers — The default security-manager has standard `check...()`-methods of the following kind:

```
public void checkRead( String s )
{ java.security.AccessController.checkPermission(
    new java.security.FilePermission(file, "read") );
}
```

Thus the default version of the `java.lang.SecurityManager`-object automatically and fully delegates security-checks to the `java.security.AccessController`-class and the related, currently relevant `java.security.ProtectionDomain`-object (Java Version 1.2 and higher²⁰²).

A conceptual imitation of the effective work of such a custom `check...()`-method may read like:

```
public void checkRead( String s )
{ if(dontReadFile) throw new java.lang.SecurityException("checkRead");
}
```

²⁰¹If a given class in a runtime-environment permits the instantiation of an object only once, then the object is called a **singleton** (with respect to the given environment), (see also page 160). `SecurityManager`-objects are singletons within a given Java Runtime Environment (JRE).

²⁰²In Version 1.1 and lower, the default security-manager produces an autistic Java Runtime Environment. A conceptual equivalent of one of its `check...()`-methods may be given as

```
public void checkRead(String s)
{ throw new java.lang.SecurityException("checkRead");
}
```

In these Java versions the `java.lang.SecurityManager`-class is abstract, to use the (strict) default security-manager, the class has first to be extended trivially

```
class StrictSecurityManager extends java.lang.SecurityManager {}
```

then the security-manager is ready for instantiation.

Such an implementation may be found in a definition of a custom `SecurityManager`-class that doesn't use the access-control mechanism.

All `check...()`-methods that exist in the default security-manager can be found in the   Java Platform API Specification [JPAPIS] of the class `java.lang.SecurityManager`.

Whereas, a list of the Java core-methods that call such `check...()`-methods (core-class-methods → `check...()`-methods) may be furnished by now with the   Java Platform API Specification [JPAPIS] as well as a list that relates `check...()`-methods with default permissions (`check...()`-methods → default permissions).

Below is given a simple standard example of a custom security-manager which permits only the access of files in the current directory with a name that begins with the letters "Merg":

```
class CustomSecurityManager extends java.lang.SecurityManager ///////////////
{ public void checkRead( String fileName )
  { if( ! fileName.startsWith("Merg") )
    { throw new java.lang.SecurityException( "checkRead: " + fileName +
                                             " does not begin with Merg: No access" );
    }
    super.checkRead( fileName );
    // Java Version 1.2 and higher for additional policy-based access-control.
  } }

public class MergeWithSM
{
  static void fileToConsole( String fileName )
  { try
    { java.io.FileReader fr = new java.io.FileReader( fileName );
      java.io.BufferedReader br = new java.io.BufferedReader( fr );
      String s;
      while( true )
      { s = br.readLine();
        if( s == null ) break;
        System.out.println(s);
      }
      br.close();
    } catch( Exception e ){ e.printStackTrace(); }
  } // end fileToConsole()-method

  static void merge( String source1AndDestination, String source2 )
  { try
    { java.io.FileWriter fw = new java.io.FileWriter( source1AndDestination, true );
      java.io.FileReader fr = new java.io.FileReader( source2 );
      java.io.BufferedReader br = new java.io.BufferedReader( fr );
      String s;
      while( true )
      { s = br.readLine();
        if( s == null ) break;
        fw.write( s, 0, s.length() );
      }
      br.close(); fw.close();
    } catch( Exception e ){ e.printStackTrace(); }
  }

  public static void main( String [] fileNames )
  { System.setSecurityManager( new CustomSecurityManager() ); ///////////////
    int toBeMerged = fileNames.length;
    switch( toBeMerged )
    { case 2: merge( fileNames[0], fileNames[1] );
      case 1: fileToConsole( fileNames[0] ); break;
      default: System.out.println("Merge: enter one or two filenames");
    }
  } } // class and main()-method
/*
(CASE: ---- NO PERMISSION ---- file-name does not begin with 'Merg')
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% javac MergeWithSM.java
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java MergeWithSM DemoFile.txt
java.lang.SecurityException: checkRead: DemoFile.txt does not begin with Merg: No access
    at CustomSecurityManager.checkRead(MergeWithSM.java:4)
    at java.io.FileInputStream.<init>(FileInputStream.java:61)
    at java.io.FileReader.<init>(FileReader.java:38)
    at MergeWithSM.fileToConsole(MergeWithSM.java:12)
    at MergeWithSM.main(MergeWithSM.java:42)
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%

(CASE: ---- PERMISSION ----)
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java MergeWithSM MergeWithSM.java
.... (this listing)
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%
*/
```

The single line `System.setSecurityManager( new CustomSecurityManager() );` indicates that during runtime, the actions of the program are monitored by a security-policy. This makes it possible even for the Java programmer to separate security considerations from the essential programming task.

Since the security-manager cannot be changed during the lifetime of a Java Runtime Environment, the `check...()`-methods remain immutable (the equivalent to effectively being un-overridable or declared `final`). A programmer is confronted with another aspect of a security-manager: **A SECURITY-MANAGER SEPARATES PROBLEM-SPECIFIC CODE (INSIDE THE STANDARD PROGRAM-CODE) FROM CODE THAT IS SECURITY-SPECIFIC** (implementations of `check...()`-methods in a security-manager). A programmer supports this approach when avoiding to explicitly call those `check...()`-methods. In this view, the `check...()`-methods should remain triggerable in the sense that they are called by core-classes of the Java system. But, nobody, except convention, ignorance or the fear of a runtime-exception, forbids the application-programmer to call those methods directly from the source-code of the program.

User and Administrator Use of a Security-Manager

User and Administrator Use of a Security-Manager — A *programmer* may customize the default security-manager by overriding `check...()`-methods and instantiate the security-manager. A *user* or *administrator* may override the programmer's choice by choosing from readily furnished security-managers when starting a program by specifying the system-property `java.security.manager` for example as follows:

```
#java -Djava.security.manager=ASecurityManager nameOfApplication
```

This disables a programmer-defined security-manager. But still the administrator may disable the user from using that option by changing an entry in the `java.security-file`:

```
java.security
```

```
policy.allowSystemProperty=false
```

This demonstrates the cascading priorities of being able to choose a security-manager for a Java Runtime Environment: Java programmer, user and finally the administrator of the system, the choice of the latter always overrides the choice of the former.

Why a Security-Manager if there is an Access-Controller for consulting an External Policy?

Why a Security-Manager if there is an Access-Controller for consulting an External Policy? — The relative ease of use of the access-controller confines user and administrator and even the programmer to the use and configuration of pre-written `Permission`-objects. Whereas overriding a `check...()`-method of the security-manager gives the programmer more freedom. The `check...()`-method may interact with the user and ask for further authentication. Or the `check...()`-method may implement a more elaborate pattern-matching procedure than a custom programmable permission²⁰³.

Java Access-Controller: Policy

Java Access-Controller: Policy — Recall the graphic on page 454. Objects of type `SecureClassLoader` load classes and for each loaded class they produce a `CodeSource`-object and an object of type `ProtectionDomain`. A protection-domain gathers those permissions that are relevant for the loaded class; mainly furnished by the `Policy`-object. (The `Policy`-object represents the entire Java Runtime Environment's security-policy, this object is just a collection of code-sources with their respective `Permission`-objects.) Method-calls from the loaded classes or their derived objects trigger a `java.security.AccessController.checkPermission()`-call (either over the standard `java.security.SecurityManager.checkPermission()`-call or seldom coming directly from the source-code). Finally the class `java.security.AccessController` basically matches `CodeSource`-objects and compares `Permission`-objects that are associated with these code-sources. An exception is thrown if access is denied; whereas if the access is granted, the `checkPermission()`-call returns without further notice.

There may exist **ONE OR NONE ACTIVE** `java.security.Policy`-object in a Java Runtime Environment. By default such an object may not be editable²⁰⁴. A new `Policy`-object also is produced when call-

²⁰³The security-manager in relation to policy-files also has been discussed in the Java TechTip of Sept. 26th, 2000.

²⁰⁴There may exist multiple `java.security.Policy`-objects but only one `Policy`-object may be SET (`java.security.Policy.setPolicy()`), this means being referenced by secure class-loaders. A policy can be changed during the lifetime of the Java Runtime Environment by granting the permission `java.security.SecurityPermission("setPolicy")`.

ing the non-static `java.security.Policy.refresh()`-method; that `Policy`-object then can be read out with the program below.



A secure class-loader consults the `Policy`-object when producing a protection-domain for a to-be-loaded class. The `Policy`-object gives the runtime-wide range of permissions, specialized to class-specific permissions within the class-related protection-domain. Finally, when granting or denying access, the access-controller consults protection-domains for matching code-sources and for comparing permissions.

This text does NOT show how to write a `policy.provider` like the default `policy-provider` given by the class `sun.security.provider.PolicyFile`, that generates a `Policy`-object from a file. This text shows NEITHER how to directly instantiate a proper `java.security.Policy`-object (using equally un-commented objects of class-type `java.security.PermissionCollection`). This text only intends to show from where a `Policy`-object comes and for what it can be used. — The following program prints out the policy which is valid in the current Java Runtime Environment:

```
// reads the policy via the code-source object of this source-code
public class ReadPolicy
{
    public static void main( String unused[] )
    { String fileName =
"/Users/andreadipietro/JavaCodeFiles/JRESecurity/PolicyBasedSecurity/ReadPolicy.class";
        try
        { java.security.Policy p = java.security.Policy.getPolicy();
          java.security.CodeSource cs =
            new java.security.CodeSource(
                new java.net.URL( "file", "localhost", fileName ), null );
          java.security.PermissionCollection pc = p.getPermissions( cs );
          for( java.util.Enumeration e = pc.elements(); e.hasMoreElements(); )
              System.out.println(e.nextElement() );
          } catch( Exception e ){ e.printStackTrace(); }
    } }
/* same output for URL-object = null! */
/*
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% javac ReadPolicy.java
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java ReadPolicy
(java.util.PropertyPermission java.vendor read)
(java.util.PropertyPermission java.specification.version read)
(java.util.PropertyPermission com.apple.macos.useScreenMenuBar read,write) <-- Apple-specific
(java.util.PropertyPermission line.separator read)
(java.util.PropertyPermission java.class.version read)
(java.util.PropertyPermission mrj.version read) <-- Apple-specific
(java.util.PropertyPermission java.specification.name read)
(java.util.PropertyPermission java.vendor.url read)
(java.util.PropertyPermission java.vm.version read)
(java.util.PropertyPermission os.name read)
(java.util.PropertyPermission os.arch read)
(java.util.PropertyPermission os.version read)
(java.util.PropertyPermission java.version read)
(java.util.PropertyPermission java.vm.specification.version read)
(java.util.PropertyPermission java.vm.specification.name read)
(java.util.PropertyPermission java.specification.vendor read)
(java.util.PropertyPermission java.vm.vendor read)
(java.util.PropertyPermission file.separator read)
(java.util.PropertyPermission path.separator read)
(java.util.PropertyPermission java.vm.name read)
(java.util.PropertyPermission java.vm.specification.vendor read)
(java.net.SocketPermission localhost:1024- listen,resolve)
(java.lang.RuntimePermission stopThread)
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%
*/
```

This program only reads out most of the current policy. Print-outs of permissions specifically designed for some given code can be seen with the program on page 468, there is printed out the protection-domain of a program!



Remember from the graphic on page 454 the different roles of the objects of type `Policy` and `ProtectionDomain`: For the current Java Runtime Environment, the `Policy`-object collects all permissions for the various code-sources. Whereas each securely-loaded class gets its own policy-based object of type `ProtectionDomain`. The protection-domain gathers the code-specific code-source, signers and permissions mainly coming from the Java Runtime-specific `Policy`-object! (“mainly” refers to the additional “read”-permission to the loaded code’s code-source.)

Java Access-Controller: Custom Permissions

Java Access-Controller: Custom Permissions — The following example is furnished only to give an impression of how to implement and use an own custom permission: Accomplishing this requires to extend the so-called `java.security.BasicPermission`.

```
public class CustomPermission extends java.security.BasicPermission
{ static{ System.out.println("CustomPermission is loaded");}
  public CustomPermission( String name ){ super( name ); }
  public CustomPermission( String name, String actions ){ super( name, actions ); }

  public static void main( String unused[] )
  { CustomPermission perm = new CustomPermission("theName");
    try
    { java.security.AccessController.checkPermission( perm );
      System.out.println( perm.toString() + " has been granted!" );
    }catch( java.security.AccessControlException e)
    { System.out.println( perm.toString() + " has NOT been granted!" );
    } } }
/*
with default security settings
(CustomPermission theName ) has NOT been granted!

[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% javac CustomPermission.java
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java CustomPermission
CustomPermission is loaded
(CustomPermission theName) has NOT been granted!
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%
*/

/*
With "/lib/security/java.policy"-file in home-directory of Java Runtime System (JRS)
or the ".java.policy"-text-file (if necessary newly created) in the home-directory of user,
containing the entry:
    grant{ permission CustomPermission "theName";};

[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java CustomPermission
CustomPermission is loaded
(CustomPermission theName) has been granted!
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%
*/
```

Only if the use of standard permissions or a custom-written security-manager turns out to be impracticable, then take into account the use of fully-deployable custom-written permissions²⁰⁵.

Above, the default configuration of the `java.security`-file lets the program get the information whether the permission has been granted or not, from one of two default policy-files (`java.policy` for the system-wide policy and `.java.policy` in the home-directory of the user). Below is shown (again) how the policy-text-file has to be customized: (in most cases the home-directory of the user is the one easier accessible)

```
/* Entry within the file ${java.home}/lib/security/java.policy or
   ${user.home}/.java.policy
*/
grant
{
    permission CustomPermission "theName";
    //    permission java.security.AllPermission;
    //    The 'AllPermission' functions as an alternative for testing, here outcommented.
};
```

User or administrator get full control by editing the text-file above (emptying or leaving it).

Java Access-Controller: Protection-Domain

Java Access-Controller: Protection-Domain — Policy-text-files, a Java Runtime's active `Policy`-object, and class-specific object's of type `java.security.ProtectionDomain` gather associations between code-sources and all its permissions.

²⁰⁵In custom permissions, other than `java.security.BasicPermission`, implementing the `implies()`-method takes some consideration.

When starting, the Java Runtime Environment may not find a custom-written `Permission`-class. Instead of issuing a `ClassNotFoundException`, objects of class-type `java.security.UnresolvedPermission` are instantiated. The first call of a `checkPermission(customPermission)`-method triggers a reloading-process of the so-far unresolved `Permission`-class. If the custom `Permission`-class can be loaded then, the standard security procedures follow, otherwise the call returns without notice and no security-check occurs.

Users or administrators enter URL's of code-sources and associated permissions into the policy-text-files to be stored persistently. If policy-files contain different permissions for the same code-source, on creating a Java Runtime Environment, then the default-class `sun.security.provider.PolicyFile` merges them automatically into one `Policy`-object. When loading a class, a secure class-loader (of class-type `java.security.SecureClassLoader`) creates an object of type `CodeSource`, a reference to this object and the associated permissions are collected within an object of type `ProtectionDomain`.

Now consider the Java Runtime Environment to have a security-manager activated, thus doing access-control. Then the call of an access-controlled method, of the loaded class above, triggers the access-controller to take the requested permission, and compare this information with the granted permissions taken from that code-source's protection-domain: If the permissions match, access is granted without further notice; if the permissions do not relate into matching, the access-controller throws an exception.

Thus a protection-domain gathers code-specific permissions. **A protection-domain of a specific piece of loaded Java-code is an aggregation of:**

Code-source from where the class-file has been loaded into the JRE (code-source) (URL)

Permissions Its set of associated permissions.

If inside the policy-text-files, there exists only a single entry concerning a specific code-source, then the grant-entry (plus the read-permission on the location of the source-code) constitutes a persistent representative of a protection-domain. A protection-domain can be seen as a kind of code-specific mini-policy. A **fully-resolved protection-domain** references a `CodeSource`-object and contains all the permissions that are given to a loaded class. Protection-domains which are not fully-resolved, for example those that are given by single policy-files, are merged together until they form a fully-resolved protection-domain. The protection-domain then can be used by an access-controller for a proper decision when to grant access and when to deny access to the platform's resources.

In Java, a protection-domain is given by an object derived from the class with the identifier `java.security.ProtectionDomain`²⁰⁶. A programmer can retrieve the associated fully-resolved protection-domain of a given class by using the instance-method `java.lang.Class.getProtectionDomain()` (returns successfully when no security-manager is used or if the policy gives this code a `java.lang.RuntimePermission("getProtectionDomain", null)`). Core-classes that have been loaded by the initial class-loader have no protection-domain, that means `getProtectionDomain()`, returns `null`. Null protection-domains are also called **system protection-domains**. The permissions, for the current class or object, that are actually relevant for granting access can be given by lines inside the following program:

Classes loaded by the null class-loader have null protection-domains



```
class ReadProtectionDomain
{ static void printInfo( String className )
{ try
{ Class cl = Class.forName( className );
String sloader;
try
{ java.lang.ClassLoader loader = cl.getClassLoader();
sloader = loader.toString();
}catch( java.lang.NullPointerException e ){ sloader = "null"; }
System.out.println( "class-loader of " + className + ": " + sloader +
"\nprotection-domain of " + className + ": " +
cl.getProtectionDomain()
); // PRINTOUT OF PROTECTION-DOMAIN
}catch( java.lang.Exception e){ e.printStackTrace(); }
}
public static void main( String unused[] )
{ printInfo( "java.lang.String" );
printInfo( "ReadProtectionDomain" ); // this program itself!
} }

/*
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% javac ReadProtectionDomain.java
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java ReadProtectionDomain
class-loader of java.lang.String: null
protection-domain of java.lang.String: ProtectionDomain null
java.security.Permissions@30c19b (
(java.security.AllPermission <all permissions> <all actions>)
)
```

²⁰⁶A protection-domain can be instantiated from given `java.security.CodeSource` and `java.security.Permissions` objects (note the "s" in the last class-name). A closer look at the inner workings is omitted here, because it mainly concerns programmers when programming their own `Policy`-objects or `SecureClassLoaders`. This activity, as many other highly specialized activities, is located outside the horizon of this text.


```

class-loader of ReadProtectionDomain: sun.misc.Launcher$AppClassLoader@33056f
protection-domain of ReadProtectionDomain:
ProtectionDomain (file:/Users/andreadipietro/JavaCodeFiles/JRESecurity/PolicyBasedSecurity/ <no certificates>)
java.security.Permissions@19821f (
  (java.io.FilePermission /Users/andreadipietro/JavaCodeFiles/JRESecurity/PolicyBasedSecurity/- read)
  (unresolved CustomPermission theName null)
  (java.lang.RuntimePermission exitVM)
  (java.lang.RuntimePermission stopThread)
  (java.util.PropertyPermission java.vendor read)
  (java.util.PropertyPermission java.specification.version read)
  (java.util.PropertyPermission com.apple.macos.useScreenMenuBar read,write)
  (java.util.PropertyPermission line.separator read)
  (java.util.PropertyPermission java.class.version read)
  (java.util.PropertyPermission mrj.version read)
  (java.util.PropertyPermission java.specification.name read)
  (java.util.PropertyPermission java.vendor.url read)
  (java.util.PropertyPermission java.vm.version read)
  (java.util.PropertyPermission os.name read)
  (java.util.PropertyPermission os.arch read)
  (java.util.PropertyPermission os.version read)
  (java.util.PropertyPermission java.version read)
  (java.util.PropertyPermission java.vm.specification.version read)
  (java.util.PropertyPermission java.vm.specification.name read)
  (java.util.PropertyPermission java.specification.vendor read)
  (java.util.PropertyPermission java.vm.vendor read)
  (java.util.PropertyPermission file.separator read)
  (java.util.PropertyPermission path.separator read)
  (java.util.PropertyPermission java.vm.name read)
  (java.util.PropertyPermission java.vm.specification.vendor read)
  (java.net.SocketPermission localhost:1024- listen,resolve)
)

[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%

```

Objects of type `java.util.PropertyPermission` allow to access the property-files of the Java Runtime System! In the default case the output includes the protection-domains with their information given by

- the platform-wide `java.policy`-file,
- the user-domain-wide `.java.policy`-file²⁰⁷ and finally
- THE PERMISSION TO READ THE PLACE FROM WHICH THE CLASS-FILE OF THE CURRENT PROGRAM ORIGINATED!

This last permission, which generally needn't be entered explicitly, is given automatically by a secure class-loader, an object of class-type `java.security.SecureClassLoader`, that automatically adds it to any protection-domain. Protection-domains that originate on the side of the (secure) class-loader extend the system- and user-defined-policy. The standard additional permission, the read-permission for the directory from where the class-file has been loaded, already has been mentioned on page 460. As can be seen above, this permission supplements the list already printed out on page 466.

A fully-implemented `java.security.SecureClassLoader`, like the `java.net.URLClassLoader`, automatically associates such a protection-domain to the loaded class. That protection-domain is associated with the to-be-loaded class when the class-loader calls its `defineClass()`-method. A coarse impression may give this pseudo-code:

```

java.security.Policy policy = java.security.Policy.getPolicy();
java.security.Permissions perms = policy.evaluate( someCodeSource );
java.security.ProtectionDomain protectDom =
    new java.security.ProtectionDomain( someCodeSource, perms);
someClass =
    java.security.SecureClassLoader.defineClass(
        className, byteArray, 0, byteArray.length, protectDom, signers);

```

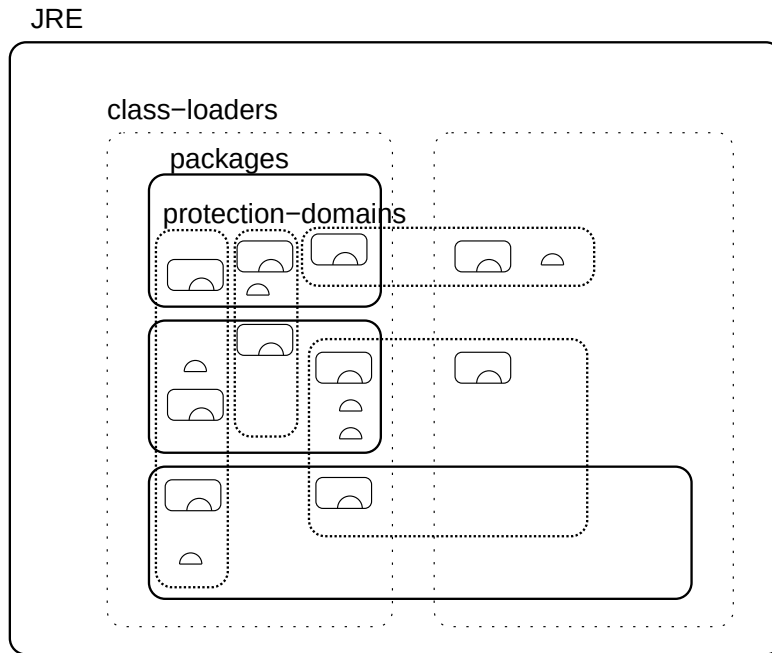
(The idea of signing data is introduced in the sections about cryptography.) Finally, be aware of the fact, that a programmer always can extend the class `java.security.SecureClassLoader` to write a secure class-loader which automatically handles most of the policy-related tasks. Note that the relevant permissions from the system's policy then are automatically combined generating a fully-resolved protection-domain.



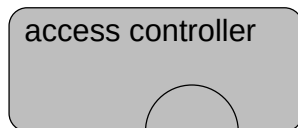
Since each class in a Java Runtime Environment belongs to exactly one fully-resolved protection-domain, those *sets of classes with the same protection-domain* partition the set of all loaded classes and derived objects within a Java Runtime Environment. This partition may not respect the name-spaces of class-loaders or packages (protection-domains construct no name-spaces):

²⁰⁷Remember to note the leading period (.), making it an “invisible” file in Unix-based operating-systems.

Name-Spaces and Protection-Domains in a Java Runtime Environment (JRE)



Thus note: Differing class-loaders may produce the same protection-domain! This is the case if the source of the code, the signatures and the permissions of the loaded code coincide to be exactly the same.



Recall the, by now well-known graphic on page 454 about Java's security-related access-control context. While presenting some of the context of policy-controlled access restrictions, the core element of that graphic shows the class `java.security.AccessController` filtering an access-request as follows: A (n access-controlled) method, usually a method of a class from the Java base-classes calls the security-manager which relays the call to the static method

`checkPermission( Permission )`. The transferred `Permission`-object represents the access-right which is requested by the method-call.

```
java.security.AccessController.checkPermission( java.security.Permission )
```

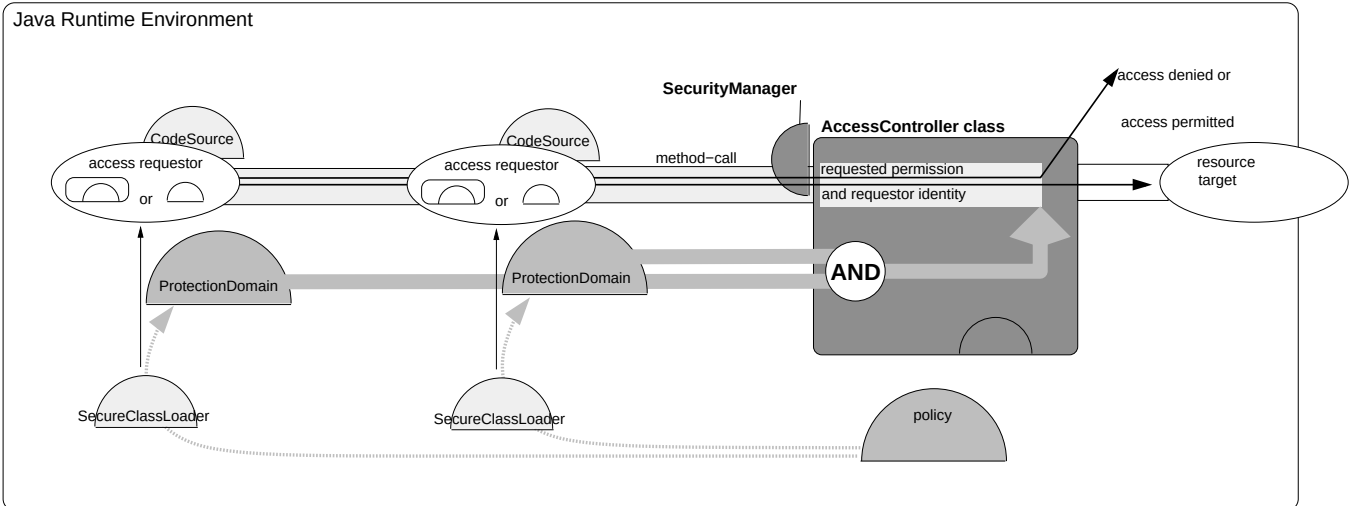
The access-controller compares the requested permission to the permissions inside the protection-domain of the calling-class: If the access to a resource has been granted, then the `checkPermission()`-method returns without notice. If no access is to be granted, then the method returns by throwing an exception!

Java Access-Controller: Method-Call History

Java Access-Controller: Method-Call History — The block of source-code of a method may contain method-calls itself; thus from methods may be called other methods: The source-code of the `HelloWorld`-class on page 71 contains a `main()`-method that calls the method `System.out.println()`. The `Recurring`-class on page 126 and the `Callback`-class on page 127 exemplify further the use of method-calls within method-blocks. On page 244 has been introduced how (by qualification with the package-name) to call methods within the code of other packages.

For each thread the `AccessController`-class tracks the method-call history and stores it in an object of class-type `AccessControlContext`. This becomes important when there occurred *method-calls within method-calls* accompanied by differing protection-domains. Then BY DEFAULT THE ACCESS-CONTROLLER GRANTS ACCESS ONLY IF EACH INDIVIDUAL METHOD OF THE METHOD-CALL HISTORY WOULD HAVE BEEN GRANTED THE REQUESTED ACCESS. The method-call history becomes important to the access-controller when there are involved different protection-domains. That means classes or objects, which issue the method-calls, may be located in different directories or come from differing class-loaders. (Later, a class that implements a `PrivilegedAction`-interface makes the `AccessController`-class forget about the previous method-call history.)

Method-Call History of a Single Thread



Ultimately, when a security-checked method-call activates the access-controller, the following happens: The access-controller considers the entire method-call history of the associated thread: Each method-call is considered with the fully-resolved protection-domains of its class, independently of the code-sources. That means each code-source that is involved in the method-call history has to be granted the permission to access the desired resource. In this case the permission to access the desired resource is given only if EACH individual fully-evaluated protection-domain of the calling-history grants this permission. In other words, if there exists one class in the calling-history whose fully-resolved protection-domain does not give the desired permission, then the access-controller throws an exception and thereby denies the access to the resource.

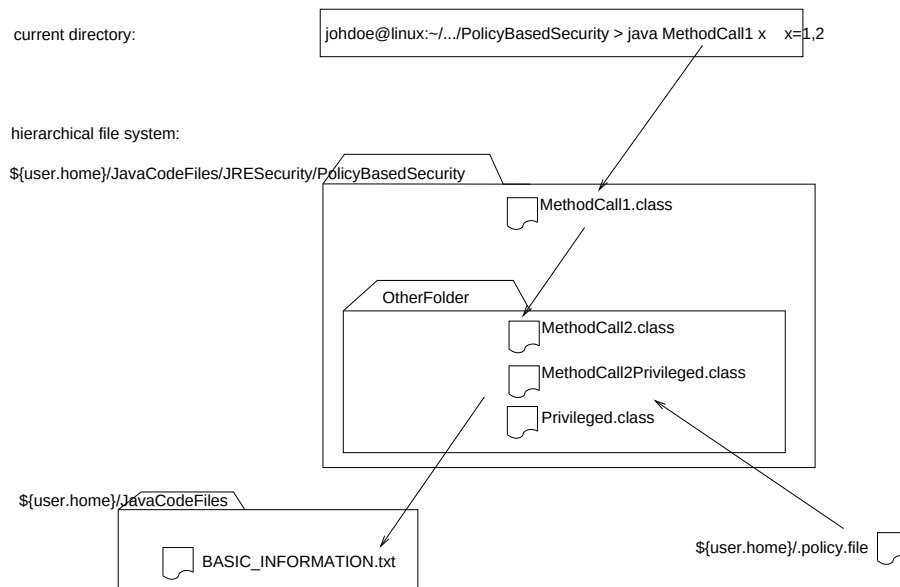
A secure class-loader uses the overall Policy-object for filtering and rearranging code-specific permissions for the loaded code, preselecting decision-relevant permissions for the access-controller: Here can be seen another use of the code-specific protection-domains; an object of type `AccessControlContext`, that can be retrieved from the `AccessController`-class, collects and resolves all these `ProtectionDomain`-objects.

Java Access-Controller: Privileged Code

Java Access-Controller: Privileged Code — All classes or objects in the method-call history of one thread have to be given effectively the same permission, for the access-controller to grant access to a given resource.

For a programmer there is a way to override this mechanism: If a call of the method comes out of an object of interface-type `java.security.PrivilegedAction` or if a call comes from an object of interface-type `java.security.PrivilegedExceptionAction`,²⁰⁸ then only the ensuing method-call history becomes relevant for the access-controller. The previous method-call history of the current thread thus is ignored by the access-controller when considering permissions for access-rights to be granted. An example of the use of the interfaces `java.security.PrivilegedAction` or `java.security.PrivilegedExceptionAction` is given below.

²⁰⁸Not to be confounded with an exception-type also used below: `java.security.PrivilegedActionException`.



The class that does the second method-call need not be in another name-space! It does not need to be loaded by another class-loader. To get a different set of permissions, the class simply can be in another directory. The different set of permissions already puts that class into another protection-domain:

```
/* Here, the folder "JavaCodeFiles" is assumed to have been copied into the directory
 * marked by the property "user.home".
 */
/* PREREQUISITES PREREQUISITES PREREQUISITES PREREQUISITES PREREQUISITES PREREQUISITES
   Inside the text-file 'java.policy' in the directory ${java.home}/lib/security
   or inside the text-file '.java.policy' in the user's home-directory ~/
   (the directory Windows 98, for example:
        C:\Programs\JavaSoft\JRE\1.3.1\lib\security\java.policy )
   have to be inserted the following lines:
```

```
grant codeBase // below is a Uniform Resource Locator (URL)
"file:${user.home}/JavaCodeFiles/JRESecurity/PolicyBasedSecurity/OtherFolder/-"
{
    permission java.util.PropertyPermission "user.home","read";
    permission java.io.FilePermission
    "${user.home}${/}JavaCodeFiles${/}BASIC_INFORMATION.txt","read";
        // above is a relative path of the platform-specific
        // Hierarchical File System (HFS-path) interspersed
        // with property-extension constructs.
    // permission java.security.AllPermission; // <-- for testing purposes
};
// Within the string "...ity/PolicyBasedSecurity/-" the
// symbol '*' means: All files and directory-names inside
// the directory of the prefixing path-name.
// symbol '-' means: All files inside the directory of the prefixing path-name,
// INCLUDING the files of all subdirectories.
// A string "<<ALL FILES>>" covers the entire file-hierarchy.
```

```
PREREQUISITES PREREQUISITES PREREQUISITES PREREQUISITES PREREQUISITES PREREQUISITES */
```

Adding the grant-clause above to a policy-file (java.policy or .java.policy) gives only code from within the folder “OtherFolder” the permission to read the file “BASIC_INFORMATION.txt”! (Symbols '*' and '-' in paths and URL's within policy-files have special meanings as described above.)

```
/*
The source-code below prints out the contents of a text-file.
This text has to be read from the Hierarchical File System (HFS).
The user has the choice between two getFile()-methods.
These two getFile()-methods are implemented in classes, both having a class-loader,
that differs from the class-loader of this code.
(The user of this program can choose between those methods by entering 1 or 2 when
starting this program.)
The source- or byte-code of the classes 'MethodCall2' and 'MethodCall2Privileged'
are to be placed into a folder './OtherFolder' ('.' means relative to the directory
in which the code below is run.)
*/
```

```
// file MethodCall1.java

//import OtherFolder.*; // *****
/* outcommented like the package declaration in the files MethodCall2.java and
   MethodCall2Privileged.java

   For working with a standard package,
   remove the outcommenting parts marked by *****
   and compile the entire source-code with 'javac MethodCall1.java'
   --- The outcommenting shows how to incorporate classes from other directories
       without needing a package construct!
*/

class MethodCall1 // -- main()-method call below, the first method-call
{ public static void main( String oneOrTwo[] )
  { //System.out.println( System.getProperty( "user.home" ) );
    // The line above has to be outcommented, when security-check is activated!
    try
    {
      //Class cl = Class.forName("MethodCall1");
      //System.out.println(cl.getClassLoader().toString() + cl.getProtectionDomain());
      java.io.FileReader fr = null;
      switch( Integer.parseInt( oneOrTwo[0] ) )
      { case 1: MethodCall2 x = new MethodCall2();
          fr = x.getFile(); // <--- getFile()-method call, the second method-call
          break;
        case 2: MethodCall2Privileged y = new MethodCall2Privileged();
          fr = y.getFile(); // <--- getFile()-method call, the second method-call
          break;
      }
      java.io.BufferedReader br = new java.io.BufferedReader( fr );
      String s;
      while( true )
      { s = br.readLine();
        if( s == null ) break;
        System.out.println(s);
      }
      br.close();
    }catch( java.lang.Exception e){ e.printStackTrace(); }
  } }
/* The permissions as
   given in the '.java.policy'-file work in the privileged case
   because there they permit the access.
   In the non-privileged case
   those permissions are tested against the default permissions which do not allow the
   unrestricted access of other directories.
*/

/*
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% javac -classpath ../OtherFolder MethodCall1.java
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java -classpath ../OtherFolder -Djava.security.manager MethodCall1 1
MethodCall2 loaded
MethodCall2 instantiated
java.security.AccessControlException: access denied (java.util.PropertyPermission user.home read)
    at java.security.AccessControlContext.checkPermission(AccessControlContext.java:272)
    at java.security.AccessController.checkPermission(AccessController.java:399)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:545)
    at java.lang.SecurityManager.checkPropertyAccess(SecurityManager.java:1278)
    at java.lang.System.getProperty(System.java:560)
    at MethodCall2.getFile(MethodCall2.java:7)
    at MethodCall1.main(MethodCall1.java:27)
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java -classpath ../OtherFolder -Djava.security.manager MethodCall1 2
MethodCall2Privileged loaded
MethodCall2Privileged instantiated
This text ..... (Contents of the file ~/JavaCodeFiles/BASIC_INFORMATION.txt)
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%
*/

```

Before compiling and running the code, check the constellation of the files and directories as described by the graphic above. And check whether the PREREQUISITES are met. The program above runs with two options (1 or 2) either instantiating the 'MethodCall2'-class or by instantiating the 'MethodCall2Privileged'-class.

1. In the first case the reading fails because the extension of the policy-file, which has been marked by "PREREQUISITE", only gives access to the file "BASIC_INFORMATION.txt" to Java-code from inside the "OtherFolder" but NOT from inside the folder "PolicyBasedSecurity". So the method-call history does not have all method-calls given the same access-permissions.
2. In the second case, of the privileged calling, the file "BASIC_INFORMATION.txt" is read because now only the permissions given to the code that issues the second call are relevant.

The following two pieces of source-code have to be compiled in or into a directory that is part of the operating-system's CLASSPATH-variable; only then the Java Runtime Environment will find it. Above within the print-out of a terminal-window is shown how to extend the CLASSPATH-variable during compile- and runtime by using the java-tool's -classpath-option.

```
// file MethodCall2.java

//package OtherFolder; // *****

public class MethodCall2
{ static{ System.out.println( "MethodCall2 loaded" ); }
  { System.out.println( "MethodCall2 instantiated" ); }
  public java.io.FileReader getFile() throws java.io.FileNotFoundException
  // line above, public only with package
  { return new java.io.FileReader(
      System.getProperty( "user.home" ) + "/JavaCodeFiles/BASIC_INFORMATION.txt" );
  } }
}
```

Here the source-code for the case in which only part of the method-call history is considered by the access-controller. In conjunction with the entry of the policy-file above (surrounded by the PREREQUISITE-lines) this part of the code permits the access to the text-file:

```
// file MethodCall2Privileged.java

//package OtherFolder; // *****

public class MethodCall2Privileged
{ static{ System.out.println( "MethodCall2Privileged loaded" ); }
  { System.out.println( "MethodCall2Privileged instantiated" ); }
  public java.io.FileReader getFile() throws java.io.FileNotFoundException
  // why public? Because different class-loader!
  { Object o = null;
    try
    { Privileged priv = new Privileged();
      o = java.security.AccessController.doPrivileged( priv );
    } catch( java.security.PrivilegedActionException e){ e.printStackTrace(); }
    return (java.io.FileReader)o;
  } }
class Privileged implements java.security.PrivilegedExceptionAction
{ public Object run() throws java.io.FileNotFoundException
  { return new java.io.FileReader(
      System.getProperty( "user.home" ) + "/JavaCodeFiles/BASIC_INFORMATION.txt" );
  } }
}
```

To test the program above just enter “java MethodCall1 1” or “java MethodCall1 2” and in either case — MethodCall2 or MethodCall2Privileged — the text “BASIC_INFORMATION.txt” should be printed into the terminal-window or standard output (NO security checking activated!).

To run the program with activated security measures, as demonstrated above in the print-out of the terminal-window, enter “java -Djava.security.manager MethodCall1”. The classpath-options may be omitted, when the outcommented package-declarations are used.

Java Access-Controller: Policy-Secured Environment extends to Objects

Access-Controller: Policy-Secured Environment extends to Objects — Up to now, files were resources that are protected by the access-controller. How about instantiated objects in the Java Runtime Environment (JRE). Can objects be considered as resources too? A wrapper-class called `java.security.GuardedObject` makes this possible:

```
// Object in policy-secured environment
class ObjectPolicySecured
{ public static void main( String unused[] )
  { String resource = new String("This string is guarded by the access-controller");
    java.security.Permission permissionNeeded = new java.lang.RuntimePermission("exit");
    java.security.GuardedObject g =
      new java.security.GuardedObject(resource, permissionNeeded);
    System.out.println( g.getObject() );
  } }
// any kind of permission will do as long as it is granted when access needed
/* PREREQUISITE PREREQUISITE PREREQUISITE PREREQUISITE PREREQUISITE PREREQUISITE
```

```
// to be found inside a policy-file, like java.policy or .java.policy
grant
{
    permission java.lang.RuntimePermission "exit";
};
    PREREQUISITE PREREQUISITE PREREQUISITE PREREQUISITE PREREQUISITE PREREQUISITE
*/
/*
(No access controlling)
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% javac ObjectPolicySecured.java
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java ObjectPolicySecured
This string is guarded by the access-controller
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%

(access fails when security-manager has been activated)
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro% java -Djava.security.manager ObjectPolicySecured
Exception in thread "main" java.security.AccessControlException: access denied (java.lang.RuntimePermission exit)
    at java.security.AccessControlContext.checkPermission(AccessControlContext.java:272)
    at java.security.AccessController.checkPermission(AccessController.java:399)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:545)
    at java.security.Permission.checkGuard(Permission.java:86)
    at java.security.GuardedObject.getObject(GuardedObject.java:69)
    at ObjectPolicySecured.main(ObjectPolicySecured.java:7)
[localhost:~/JavaCodeFiles/JRESecurity/PolicyBasedSecurity] andreadipietro%
*/
```

Applets and Security

Applets and Security — Appletviewers are programs which start a Java Runtime Environment together with a security-manager, so that any (foreign) applet automatically encounters a restricted Java Runtime Environment. (Generally, a Java Runtime Environment that is restrained by a security-manager is also called sandbox²⁰⁹.) Some appletviewers can be configured by their users but other appletviewers may not. Usually applets are denied any access to file-systems and networks, except for reading-access to their code-source-directory. Furthermore applets generally cannot invoke applications, load libraries or define native methods. And applets have to mark their windows on the monitor as applet-generated. Appletviewers that support Java Versions 1.1 and below do not work with policies or an access-controller.

Summary: Policy-Based Access-Control Mechanisms in Java

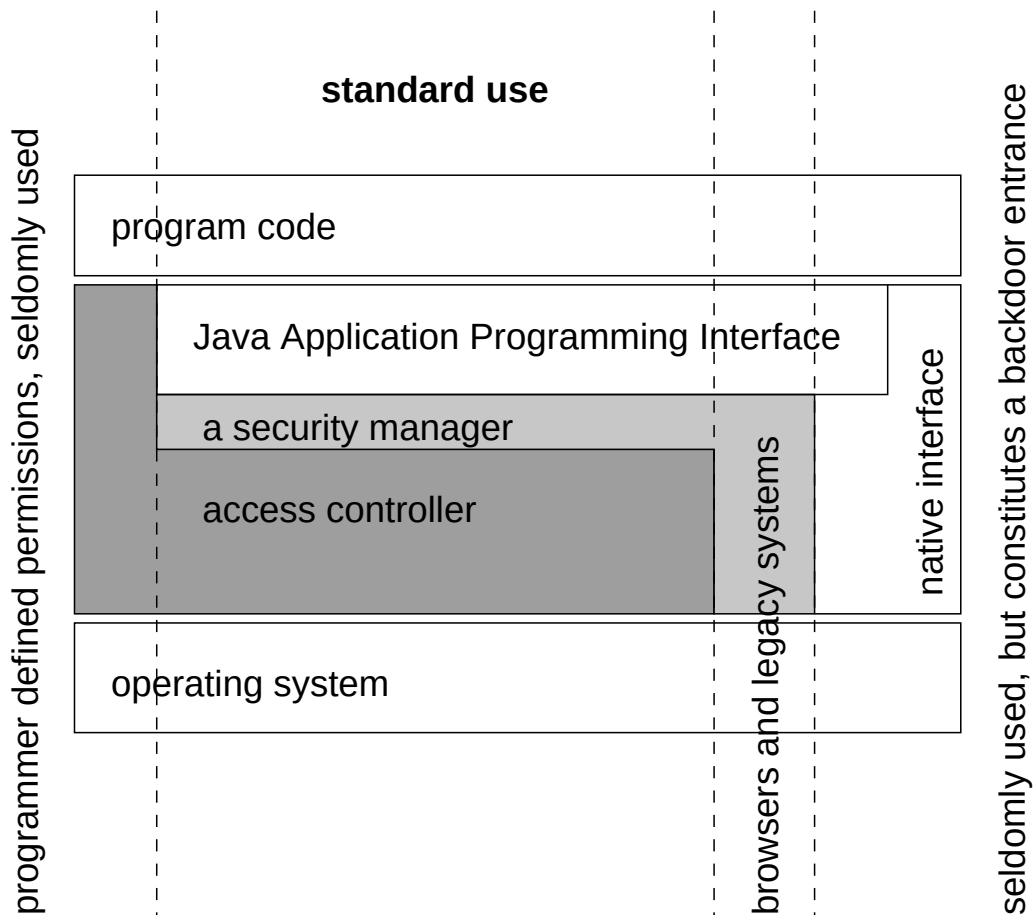
Summary: Policy-Based Access-Control Mechanisms in Java — In general, policy-based access-control in Java wants to control access to resources of the computer-platform. This concerns classes that have been loaded and which then address the system's resources by method-calls. The source from which these classes have been loaded, together with information of the system's policy furnish the necessary information for the access-controller to grant or restrict access. A method-call which has been granted access, returns without any notice of a security-check. An unpermitted method-call exits prematurely by throwing a security-exception and thus cannot run and return orderly. This mechanism makes it possible to restrict programs (thus their programmers) from reading files from directories or from connecting to other platforms over a network-connection.

Platform-administrators can keep users from running Java code without a security-manager, this requires the operating-system to furnish at least the security the Java Runtime Systems furnishes. Finally, even the physical access to the hardware of the platform should be controllable for assuring a controllable software access.

As could be seen previously, there are many ways to configure access-control on the level of the Java Runtime System and the operating-system. This diversity contains some crucial points that may ease security breaches like: A permission to change the policy, a permission to set a security-manager, a secure class-loader's `defineClass()`-method can introduce any kind of protection-domain to a code. These internal dangers should be made aware of, when trying to configure a secure Java Runtime System (JRS).

The previous text also showed different uses of security-manager and access-controller in programming: The following graphic of alternatives tries to indicates some of those different uses:

²⁰⁹That concept of an Applet Runtime Environment has been introduced beginning with page 67. Applets have been introduced around that same page and on page 396, see also page 613.



- As can be seen in the graphic above, the **standard use** of the security concept of Java can make most programming-related security-functions almost invisible. The disinterested programmer may ignore most security concepts, the only contact remains the standard Java Application Programming Interface (API). — But, as could be seen the first part of this chapter, an interested user or administrator actually can furnish the needed security-measures without changing the program. Thus, the standard use of the `SecurityManager`-object and the `AccessController`-class may occur virtually invisible to programmer or user. Java Runtime Environments of Version 1.1 or 1.0 solely use security-managers that exercise their control with overridden `check...()`-methods.
- An application-programmer is also given the means to use the `AccessController` with custom-written `Permission`-objects. Methods that are meant to be access-controlled then have to call first the `checkPermission()`-method with the custom-written permission as an argument. This option should be only used if the standard permissions turn out to be unusable for the intended kind of access.
- Finally the (seldom? used) Java Native Interface may constitute a way to circumvent the security measures established by the use of the access-controller.

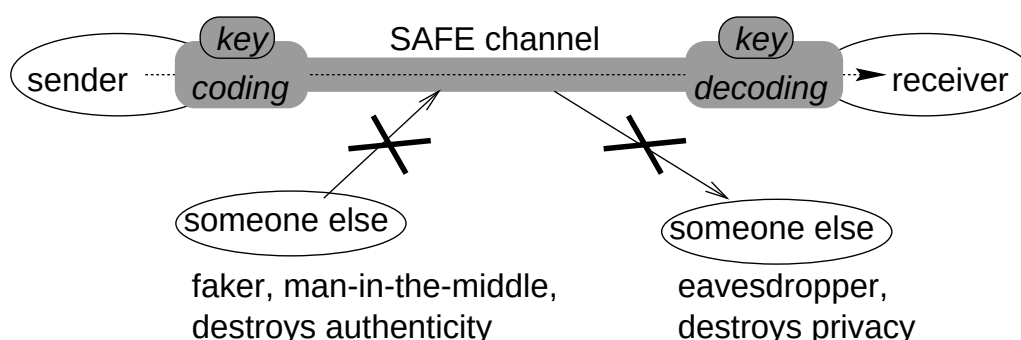
Security-Related Coding of Data

Security-Related Coding of Data — Processing data, by coding it for security reasons, ultimately means to take a sequence of bytes and to transform it in such a way that the process of transforming it back becomes very difficult for those people who are not intended to get access to the original sequence of bytes. Those people who are intended to recover the original sequence of bytes should be given means to make that re-transformation as easy and as simple as possible. This activity which involves using, analyzing and devising such processes is called **cryptography**.

Another basic mechanism, which applies cryptographic methods, is that of **authentication**: This means to analyze, devise or use methods for proving the identity of the sender of a message. Here again the conceptual basis is a controllable form of discrimination: Allowing access only to selected receivers or ascertaining that the received message was not sent by an impostor.

Basic Concepts of Security through Data-Processing

Basic Concepts of Security through Data-Processing — Cryptography locks data by transforming that data into apparently meaningless succession of bytes, the meaning is recovered only by a proper re-transformation. (Contrary to this, the access-control mechanism uses a *policy* to lock resources like hardware-services or data.)



Coding- and decoding-mechanisms are given by a coding-scheme. Such a coding-mechanism works like a locking-mechanism of a door. Many doors can have the same locking-mechanism, but different keys allow to re-use the same locking-mechanism over and over without sacrificing on security. So keys add flexibility and uniqueness to a locking-mechanism. This image can be transcended directly into coding messages with software.

As already has been mentioned, discriminating against people by processing data, takes two basic forms:

Authentication of data or files means to

1. prove the identity of the sender (a challenger is an impostor)
 2. assure that the data has been transmitted unmodified (a challenger is a forger who intercepts the communication of others)
- Thus, no third person should be able to modify or to *rewrite* the data without being noticed. Still a third person may be able to **read** the data.

Encryption of data or files means to

1. prove the identity of the sender (a challenger is an impostor)
 2. assure privacy of communication (no eavesdropper, no faker)
- Thus, no third person should be able to *read* or *effectively modify* the original data.

Social Aspects: TRUST

Trust is always
more than
identification!

Social Aspects: TRUST — Cases, where a sender can be identified by an authentication-procedure, should NOT instill trust:

In real-life, a person who shows an identity card or can reproduce a signature, which you have already seen somewhere, is not automatically allowed to enter your house. — To a certain extent, uniforms are meant to mark trustworthy people, but exactly this misconception attracts fakers.

In the same way a computer-platform should see the ability to work with electronically signed data-files as a *supplement* to its policy-based security measures. The policies for signed code should be equally carefully chosen like those policies for unsigned code. Don't grant unnecessary access rights. Furthermore, too many people with too much access rights make it difficult to trace the one that (possibly by accident) imported un-responsible code.

Concepts: Some Results of the Mathematical Theory

Concepts: Some Results of the Mathematical Theory — Cryptography and authentication rely heavily on mathematical considerations. Presenting those mathematical approaches would not necessarily support the practical application of these principles. Therefore only the relevant results of some (central) mathematical ideas are visualized below. A concrete approach to the mathematics of cryptography and authentication can be found in books on this topics or in the Internet.

Cryptography with a Secret Key (Symmetric Key)

Cryptography with a Secret Key (Symmetric Key) — Example: A text can be encrypted very easily by an algorithm that involves “shifting the alphabet”: The table below

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

can be used as an encryption engine that realizes the algorithm “a cyclical shift of the alphabet” (also abbreviated as “ROT” (for rotation)) by cyclically shifting the characters 11 places. The letter to be encrypted is identified in the upper line and substituted by the corresponding letter found in the lower line. Using the function(-table) above that implements the ROT-algorithm with the shift-index 11 (also called **secret key**) encrypts the message “THIS IS TO BE KEPT SECRET” to “JXYI YI JE RU AUFJ IUSHUJ”. Decrypting is done by exchanging the labels Input and Output in the function(-table) above. Receiver and sender need only know the algorithm name “ROT” and have to exchange the key “11” secretly. (The door's lock has the name “ROT” - this can be known publicly -, the key - not to be lost or shown - is the number “11”.)

example: secret
key

Of course, the secret-key can be found easily by trying all the 26 possibilities, when knowing the cryptographic algorithm “ROT”, but simple examples usually do not furnish the most effective functionality. Here the algorithm is relatively simple, and unfortunately also relatively simple to “break”: Encrypted data can be decrypted (here relatively effortless) without having the key!

“breaking” a
cryptographic
algorithm

Encrypting and decrypting of messages with one single key — which is known exclusively to the sender and the receiver — is called the **Secret-Key-method** (or **symmetric-key-method**) or **cipher**.

cipher =
symmetric-key-
method =
secret-key-
method

A Java program that works in the way described above is given by the following source-code (the secret key is chosen to be the number 13):

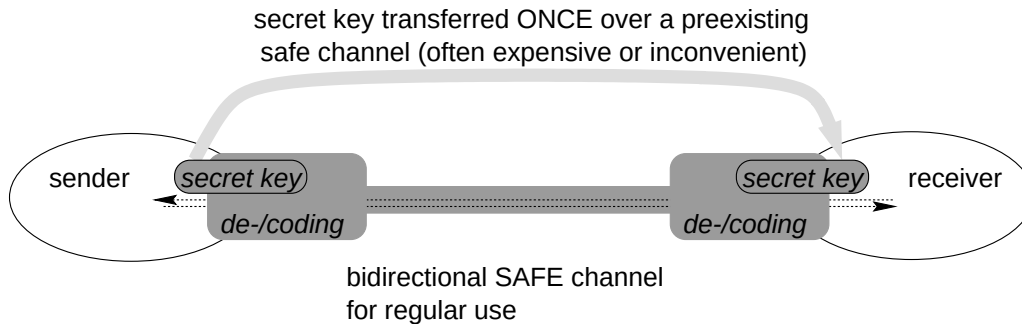
```
class ROTEncryption1
{ public static void main( String textWithoutSpaces[] )
  { byte[] clearText = textWithoutSpaces[0].getBytes(),
    encrypted = new byte[ clearText.length ];
    int clear, crypt = 0;
    for( int i = 0; i < clearText.length; i++)
    { clear = clearText[ i ];
      if( (clear >= 'A') && (clear <= 'Z')) crypt = ( (clear - 'A' + 13) % 26) + 'A';
      if( (clear >= 'a') && (clear <= 'z')) crypt = ( (clear - 'a' + 13) % 26) + 'a';
      encrypted[ i ] = (byte)crypt;
    }
    System.out.println( new String( encrypted ) );
  } }
/* A shift of 13 is the half of the 26 ring,
   therefore twice the shift reproduces the original!
*/
```

```

/*
jacobneuhau@linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity > javac ROTEncryption1.java
jacobneuhau@linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity > java ROTEncryption1 ThisIsTheEnd
GuvfVfGurRaq
jacobneuhau@linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity > java ROTEncryption1 GuvfVfGurRaq
ThisIsTheEnd
jacobneuhau@linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity >
*/

```

Secret Key Method



Problem: How to let sender and receiver share the secret-key and algorithm? Or, differently stated, how to make sure that the sender is no impostor who got the secret-key (more or less) by accident. This problem can be solved by establishing a **safe channel**. In the case the sender and receiver know each other, a safe channel may be a telephone-call during which the secret-key is exchanged as a sequence of characters by speaking and writing. Or for an additional fee, a post-office attendant may identify the sender of the secret-key when the secret-key is sent on a magnetic disk. This problem of an initial safe channel occurs in all the following security-contexts. The previous considerations were suggestions of how to solve the basic problem of BINDING REAL PERSONS TO AN IDENTITY: How can be ascertained that the owner of a key has the identity which (s)he has made public?

identity: binding person to a key

safe channel

Another problem has been hinted at during the consideration of the ROT-algorithm: How to assess the quality of an encryption algorithm? Considering an encrypted message, this means to search for methods to find the original message without the previous knowledge of the used algorithm or key ("to break the cipher"). The methods for unconventional decryption may include conventional trying or elaborate mathematical analysis.

cryptographic algorithm: "strength" = quality

The ROT-encryption method is a cipher to be broken easily by systematically trying all possible keys (as many as characters in the alphabet). When it is known that the encrypted message is written in English, there exists another approach: Get or generate a table that lists the relative frequency of single characters in English texts. Then simply analyze which letter occurs most often, this will most probably be an "e", and keep trying. Because of these relatively simple methods of breaking the ROT-cipher, it is said that the ROT-encryption-method has very little strength.

Other algorithms which up to now have been recognized as un-proportionally more difficult to decrypt are said to be of greater strength: DES (Data Encryption Standard), DESede (triple-DES), DH (for Diffie-Hellman). Their mathematical background is not treated here, the knowledge of their names suffices in most applicational contexts.

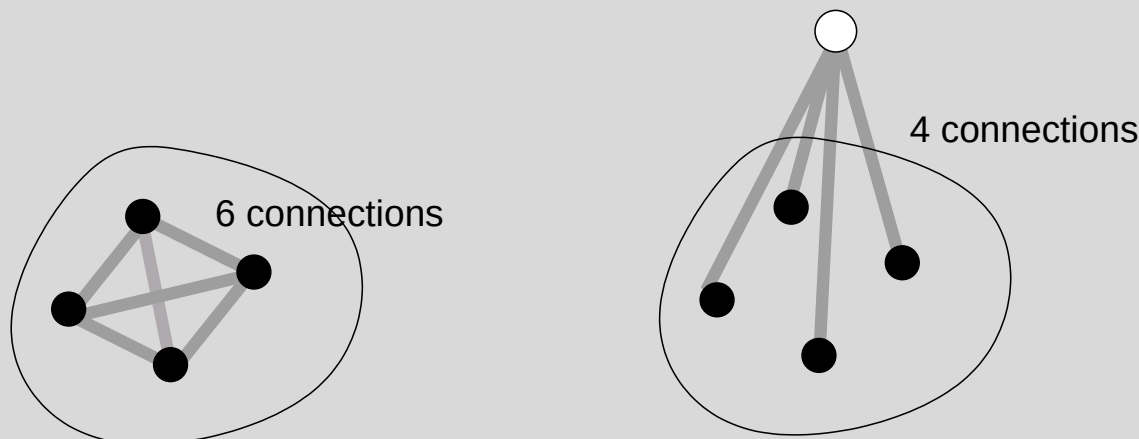
DES, DH secret key encryption algorithms

A secret-key-method called Vernam-chiffre is deemed to be mathematically 100% safe. But this method uses the safe channel extensively and thus is only economically useful for sensitive data of business- or governmental affairs. Consult the Internet or a book about cryptography on "Vernam-chiffre" for more information.

Cryptography with a Public/Private Key-Pair (Asymmetric Key)

Cryptography with a Public/Private Key-Pair (Asymmetric Key) — With a symmetric-key-method, as it has been presented, each pair of communicating instances needs a secret-key of their own. With the following method each communicating instance only needs a key-pair (associated public- and private key) of which it can distribute the public key freely. This gives an advantage that can be seen with a simple example.

Pairing-Quandary: 1. When four people ($n = 4$) want to send encrypted messages to each other, then they need six ($\frac{n \cdot (n-1)}{2}$ (combinatorics or in this simple example counting)) secret-keys when using a symmetric encryption algorithm. With an asymmetric algorithm they need only four (n) public keys. The disproportion enlarges with the number of participants.

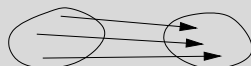


2. Consider a set containing n elements. If each element wants a (safe-)channel for communication to each other element, then $\frac{n \cdot (n-1)}{2}$ bidirectional channels are necessary. If one switchboard-entity is added, then each element needs only one safe channel to that switchboard-entity. In this case all in all only n safe channels are necessary. The larger the set, the more economically becomes the switchboard-solution.

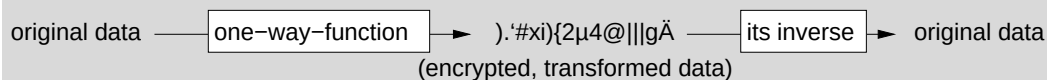
The basis for the use of public- and private keys are one-way-functions:

Mathematical concept of One-Way-Functions: For each given input, a mathematical function produces an output; much like a non-void Java method or the table of the ROT-encryption example. Mathematical functions which are called one-way-function make it extraordinarily difficult to reproduce the input from a given output. The next step involves to find two one-way-functions, so that an initial input that is processed first by one of the functions (producing some result, the encrypted initial input) and then by the other, recovers the original initial input.

The **name of an algorithm** describes the principle according to which the one-way-functions work, a **key-pair for that algorithm** fixes an individual pair of such one-way-functions that reproduce the input.



function	property: each input-value is given exactly one output-value
invertible function	add property: each possible output-value can be traced back to exactly one input-value
one-way-function	add property: it is very difficult to find the inverse-function

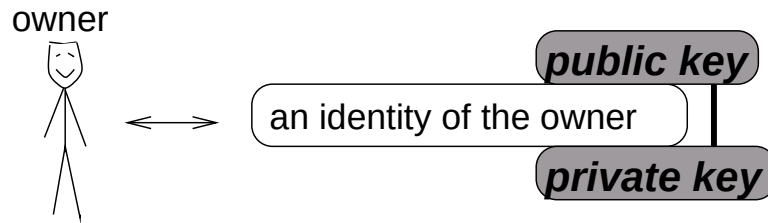


So, symmetric encryption-methods made sender and receiver share one key. Asymmetric encryption methods give each person two keys of which one can be made public (the public key, producing a one-way function) and the other is used only by the owner (private key, producing the inverse of the one-way-function). Thus, a person who wants to participate in exchanging authenticated and encrypted messages by using the key-pair-concept has to assure the following:

(key-pair) A key-pair of some given algorithm is needed for encrypting purposes.

(identity) One key of that key-pair has to be made public under some identity. People who change their identity can cause a lot of confusion. The problem of associating securely an identity to the owner of a key-pair later will prove to be imminent in key-management.

(owner) The identity has to be bound credibly to the person of the owner (also called principal or entity).

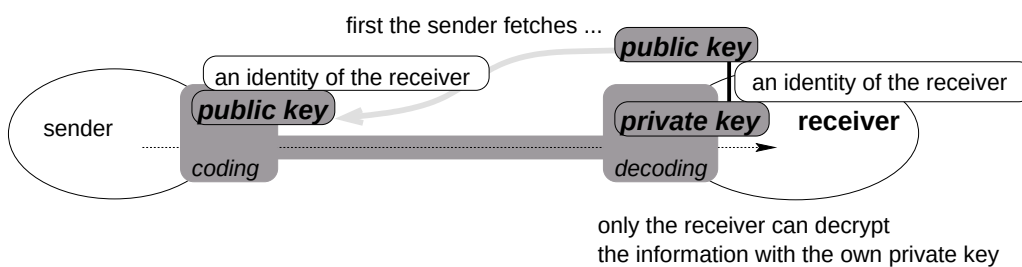


How to sent an encrypted message between persons using a key-pair algorithm? The receiver also has to generate a key-pair (The public key can be distributed freely in public, of course bound to the proper identity). The sender takes the public key of the intended receiver over a safe channel²¹⁰ and uses this public key to encrypt the message (first one-way-function applied). The receiver now uses his or hers private key (which should be kept totally private, so that only the receiver can use it) to decrypt the message (second one-way-function reproduces the original).

public key for
ENcryption,
private key for
DEcryption

Assymetric Key Method: Encryption

the public key is publicly available



Problem: Is the identity of the receiver real or fictitious? How to obtain a public key, so that one can be certain that this is the key of the presumed receiver? Or, how to ascertain that the owner of the key-pair is no impostor, someone else who pretends to be the intended receiver? These questions trigger the following one: What kind of safe channel can be used to distribute the public key or are there any alternatives? An alternative represents a “certification-process” that is mentioned later in this chapter (page 483).

Social: binding
person to identity
and thus to
public key

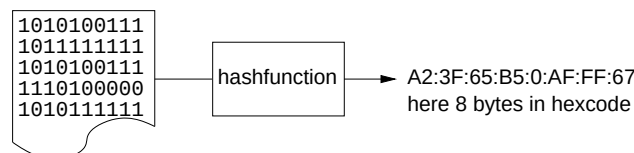
Public key algorithms are named DSA (Digital Signature Architecture), RSA (Rivist Shamir Adleman); their mathematical background is not treated in this text. A programmer may use software implementations of these algorithms without needing that background.

Public keys are often given in files encoded in the X.509-format, this format assigns the key to his owner by an “alias” and a Distinguished Name (DN) (specified in X.500) that tries to characterize the owner as globally unique. A format which is also used is the 7bit format RFC1421.

Private keys are often given in files encoded in the so-called PKCS#8-format.

Hashcode - Message-Digest - Digital Fingerprint - Checksum

Hashcode - Message-Digest - Digital Fingerprint - Checksum — Consider an arbitrary function that takes a set of data (given by a sequence of bytes like any file or array of bytes) and produces an 8 to 16 byte long result (the name “function” is mathematical term for a method that returns a result).



A trivial example of such a function may take the first bytes of the byte-sequence and return them, then any two sets of data beginning with the same sequences of bytes would produce the same result.

But there are special functions of the above kind that assure very different results even for sets of data that vary only minimally. (Of course, there may occur the case where two different sets of data nevertheless produce the same results. If the result was chosen to be only one byte long, then only 256 different results are possible. So at least after processing 256 different sets of data results will reoccur and won't

²¹⁰This assures that the public key and its identity is properly bound to the owner.

characterize the initial data any more.) But with a result 8 or 16 byte long²¹¹, functions can be found, devised as to assure that *different* initial data will most probably produce *different* 8 or 16 byte results.

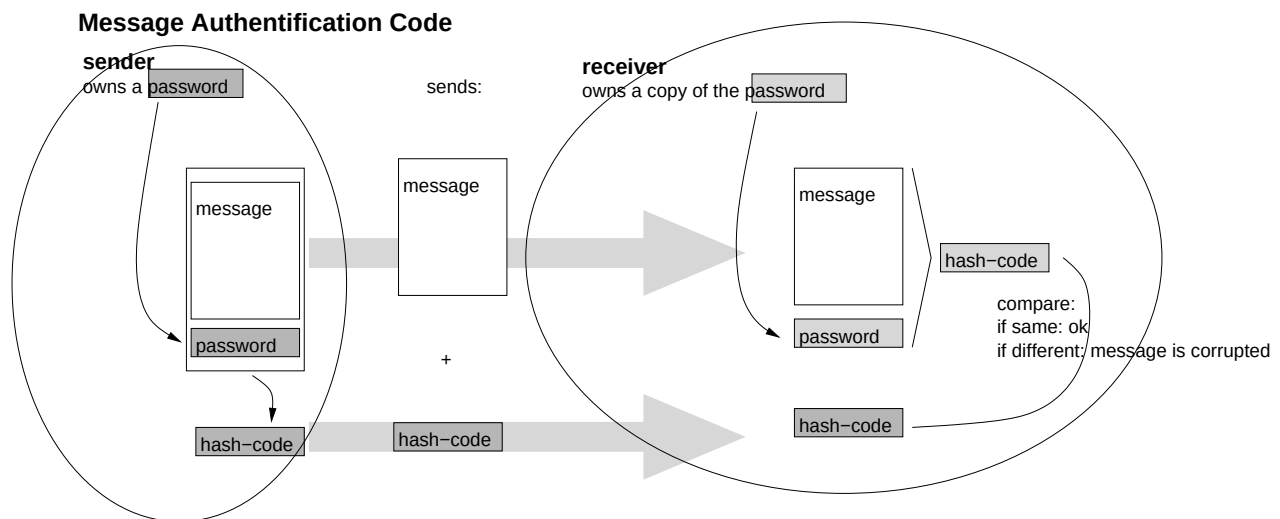
Hash-Function, Message-Digest: Hash-functions take an usually larger sequence of bytes and transform it to a little block of 8 or 16 bytes. This output-value of a hash-function is called **hashcode**, **message-digest** or **digital fingerprint**. Hash-functions produce results that, with almost absolute certainty, are different for different initial data. (The term “almost” suffices for almost all practical applications, where an unexpected occurrence of two equal results for two sets of unequal data can be tolerated.) Symbolically, short: Ascertaining $\Rightarrow =$, where the case $\Rightarrow \neq$ is impossible and the case $\neq \Rightarrow =$ may be deemed excludable, effectively assuming $\neq \Rightarrow \neq$!

For what can be used a hash-function?

The advantage of such a hash-function is that very little pieces of data can be used to detect differences in very large pieces of data. The existence of changes in versions of an electronically stored book can be detected by simply comparing the results of a hash-function. A human may write down the hashcode or memorize it, the hashcode is also short enough to be transferable by telling it another person.

Even in safe environments are used hashcodes (= digital fingerprint/message digest) to assure oneself of the integrity of a data set or for identification. In Java, each object is given its own hashcode, returned by the instance-method `java.lang.Object.hashCode()`.

Hash-values can be used as a **message authentication code (MAC)**, a method to authenticate messages:



Therefore a password is added to the message when producing a hashcode. Then the message is sent together with the hashcode but of course without the password. — The receiver, who has to know the password²¹², adds it to the message and generates a second hashcode. If both hashcodes on the receiver's side are the same, then the message is authenticated (Only the person who knows the password could have produced the message-digest for that message).

Algorithms that describe hash-functions are MD2, MD5 (Message Digest, result 128bit-block), SHA-1 (also called SHA, Secure Hash Algorithm, result 160bit-block), they do not use keys for variation (as do secret-key algorithms and key-pair algorithms).

OUTLOOK: The Java implementation of such an algorithm, as so-called `MessageDigestSPI`-class, is not treated in this text (SPI stands for Security Provider Interface). Giving a string-variable (“MD2”) to an engine-class like `java.security.MessageDigest` triggers it to select the appropriate SPI-class, from which is instantiated the actual engine-object (see the Java Provider Infrastructure on page 485).

Signing/Authenticating a Message Using Encrypted Message Digests

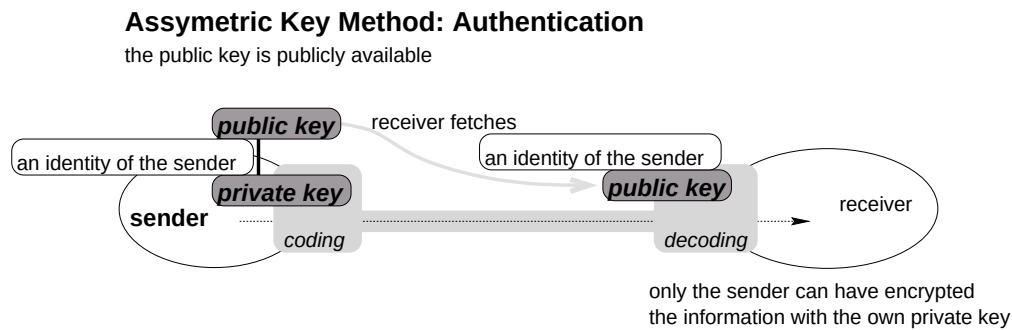
Signing/Authenticating a Message Using Encrypted Message Digests — There exists another way to implement a message authentication code (MAC), a way of making a message authenticatable within the context of a key-pair algorithm: The sender encrypts the digital fingerprint of the message with the own private key. The message remains readable and its integrity and the identity of the sender can be verified by using the public key of the sender for decrypting to get the original digital fingerprint and compare

²¹¹Eight BYTES code as much possibilities equivalent to the number of different entries that can take a Java-variable of primitive-type `long` (which is quite a lot as can be seen on page 85)!

²¹²Received over some safe channel.

it with the one generated by the receiver. The receiver gets the message un-encrypted, the encrypted message-digest and possibly the public key unless the public key has been transferred before. Conceptually this process represents just a variation of the encryption-process that encrypts a message with a private key:

private key for
signing, public
key for
authentication



Contrary to the previously introduced MAC-procedure, in the case of key-pairs NO common password is needed! So the first problem has been solved: How to assure the **INTEGRITY OF THE MESSAGE**. But another problem poses the identification of the sender: Here again the question arises of how to determine the person who holds a public key-pair to be actually the person whom this person claims to be (and not an impostor).

The standard way to solve the **PROBLEM OF THE PROVED IDENTITY** means to establish a **SAFE CHANNEL** for testing the person's identity. For cases, where establishing a safe channel for each pair of sender and receiver would become too expensive, there exists the method of **CERTIFICATION**. A central entity (called certifying authority) *signs signatures*, thus reducing the number of to be established safe channels significantly, for more see the next section.

Algorithms for digital signatures generally are composed of two algorithms: First, the algorithm of the hash-function used for generating the digital fingerprint of the message. Secondly has to be noted the algorithm used for encrypting the digital fingerprint. Commonly used combinations are SHA1 with DSA, SHA1 with RSA, MD5 with DSA, and other checksum/cipher combinations.

Securely Associating an Identity with the Owner of a Key

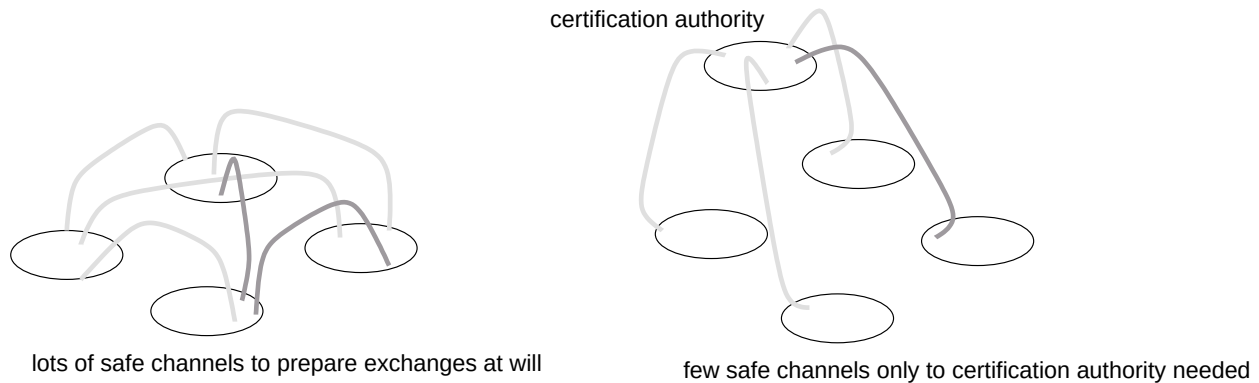
Securely Associating an Identity with the Owner of a Key — As could be seen above, before sending encrypted or authenticated messages, each pair of sender and receiver had to exchange public keys over a safe channel, thereby associating identities to real persons. Thus the identity of the actual owner of a public key has to be verified over a safe channel. A safe channel and its variations try to answer the question "How can a holder of a public key prove hers or his person matches the identity which has been associated with the public key?" There can be seen several methods for associating identity and owner of a key:

Safe Channel Once establishing a safe channel (paper-mail, telephone-call, messaging-service often proves to be impractical or too expensive) to a receiver and transferring that key. This kind of solution becomes truly impractical if the number of potential receivers, together with their geographical distance, increases.

Digital fingerprint of Key Over Safe Channel A checksum or hashcode (10 bytes) of the public key (200 bytes) may be short enough to be transferred once over the phone for comparison, but this still may prove too tedious for practical uses.

Public Key Signing (Certification) Exchanging public keys among a lot of safe channels results in the same pairing-quandary already described on page 480. To avoid this, ONE ENTITY ASSUMES AN EXCLUSIVE ROLE AMONG THE SET OF POTENTIAL SENDERS AND RECEIVERS OF MESSAGES: This special entity, usually an organization, for example a firm, establishes once a safe channel²¹³ to each member of the group and safely associates identity and owner of a given public key. During that contact the entity also distributes its public key. Then the entity **SIGNS ALL THE PUBLIC KEYS AND THEREBY AFFIRMS THE ASSOCIATION OF PUBLIC KEY AND IDENTITY FOR EACH OWNER**.

²¹³This safe channel may include just the sending of the social security number or a credit card number or, considering the other extremes, the safe channel may be realized by a visit to an attorney who checks the identity by reviewing personal documents (birth-certificate, identity-card, ...) and thus matching identity and person with much greater probability.



All others who also have the special entity's public key can now verify public keys of persons with which they never communicated over a safe channel. Such a special entity (organization, firm, bureaucratic institution) also is called **certification authority (CA)**. **Certification** is the name of the process of signing public keys to assert the identity of their holders.

Often public keys are given only in the form of a certificate, a signed public key. A simple, almost trivial, kind of certificates are those signed by their owner and called **self-signed certificates**. If there has been no certification authority to validate the identity of the key-pair owner, then public keys are often distributed as so-called self-signed certificates.

A certificate may also contain multiple signatures. Multiple signing of certificates gives so-called **key-chains**.

Certificates can be ordered by issuing a Certificate Signing Request (CSR), which, if filed electronically, uses a PKCS#10-format. Certificates can be made valid for only a limited interval of time, after this time they are not recognized anymore. Furthermore, Certificate Authorities (CAs) may regularly publish Certificate Revocation Lists (CRL), in which are listed those certificates which have been invalidated before their validity-interval had finished. The extent to which those options above are actually used, depends on practicability and usefulness: In a local, properly secured network, which can be overseen by one administrator, only self-signed certificates (=public keys) may suffice.

Variants of an Encryption Algorithm (Modes and Padding)

Variants of an Encryption Algorithm (Modes and Padding) — Many authentication- and encryption algorithms work on data-chunks of 8 bytes (64 bits), therefore those algorithms are also called **block-ciphers**. If the to-be-encoded message has not the length of a multiple of the block-length used by the algorithm, then padding procedures are necessary (one concrete procedure is given the name: PKCS5).

If the message has regularities not to be visible in the encrypted data, then different blocks may be encrypted by including information of other blocks, which supports the scrambling of regularities.

Variants of an encryption algorithm generated in such a way are called **modes of that algorithm**. (names of modes are ECB (Electronic Codebook Mode) on blocks, suitable for binary data without patterns, CBC (Cipher Block Chaining) where the previous block influences the encryption-process of the following one, it hides pattern in text or other data with regularities and needs an initialization vector, CFB (Cipher FeedBack)-mode, a variant of the one introduced before, OFB (Output FeedBack)-mode which is useful for connections with a potential for data-loss). If such algorithms are to be used for streaming data, then they may be used in a streaming-mode. These notions are used as simple parameters when configuring classes of pre-written packages for automatically doing signing and encrypting in Java.

More Concepts

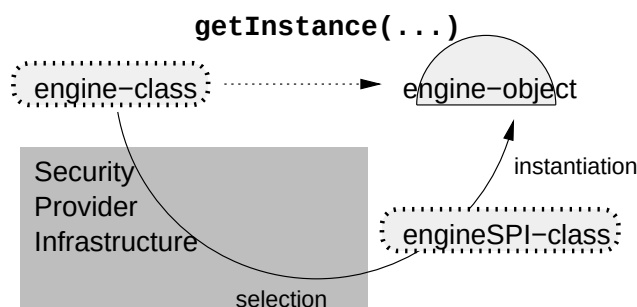
More Concepts — Steganography hides data in other data, preferably in voluminous image-, audio- and video-data. Zero Knowledge Protocols and key-agreement²¹⁴-algorithms try to fix (parts of) the identity of a person without a safe channel. Anonymizing messages also requires careful consideration of the technical background. These and other concepts are not treated further in this text.

²¹⁴A framework, a convention for such a methodology is also provided by the Java Cryptographic Extension. But this is not discussed in this text.

Java Provider-Infrastructure: Concept and Implementation

Java Provider-Infrastructure: Concept and Implementation — The Java Provider-Infrastructure is only of concern when installing new pre-written implementations of cryptographic algorithms. Or when implementing own cryptographic algorithms.

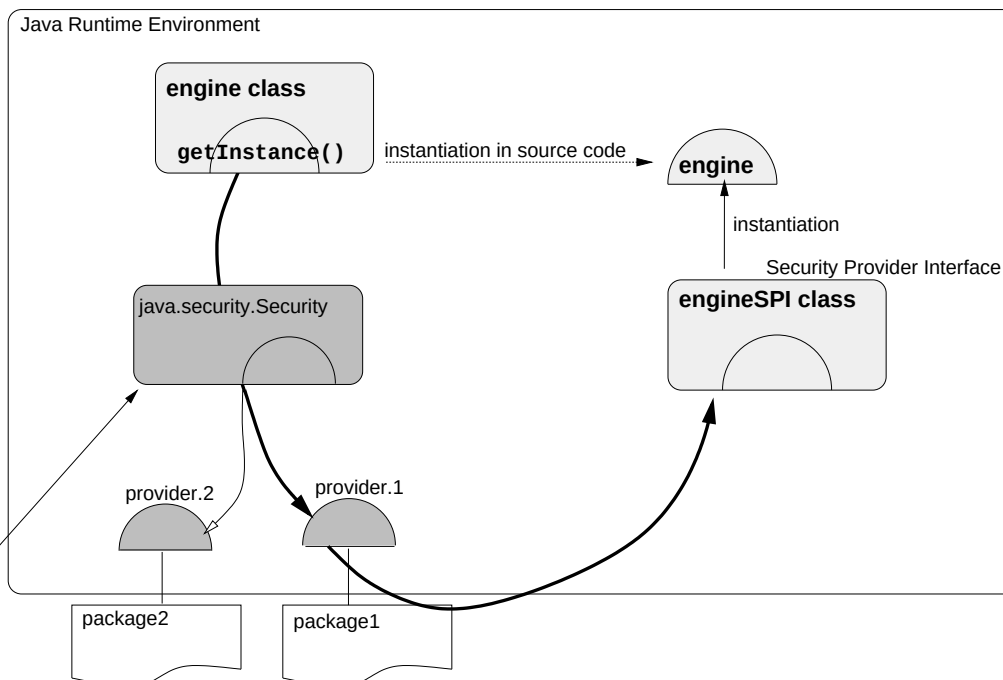
As can be seen in the graphic below, the entire Java Provider-Infrastructure is controlled by the class `java.security.Security`. When the Java Runtime Environment is started, this class consults the text-file `java.security` (introduced on page 457) for a list of provider-classes. Those provider-classes contain the implementations of the various encryption-related algorithms.



The infrastructure is activated when the programmer uses engine-classes, like those Java core-classes of the package `java.security`: `MessageDigest`, `KeyPairGenerator`, `KeyFactory`, `Signature` (and `AlgorithmParameterGenerator`, `AlgorithmParameters`) And there can be found the engine-class `CertificateFactory` in the package `java.security.cert`.

Java Provider Infrastructure

operating system



when JRE is started the file information below is automatically read into the security class and the providers are instantiated
file `$(java.home)/lib/security/java.security`

```
security.provider.1=sun.security.provider.Sun // name of default provider package
security.provider.2=<someProviderName> // name of secondary provider package
...
```

An engine-class `NameOfEngineClass` **controls** the use of an implementation of a cryptographic algorithm. The actual engine-object has to be instantiated using the factory-method `getInstance(...)` of the engine-class. The actual definition or implementation of that cryptographic algorithm is given by a Java class of name `NameOfEngineClassSPI` (SPI stands for Security Provider Interface).

For example, the engine-class `java.security.Signature` together with an engine-specifying string "SHA1withDSA", uses its static method `getInstance("SHA1withDSA")` together with the given engine-specifying string to implement the appropriate engine(-object): With this information the engine-class automatically consults the `java.security.Security`-class, which has been given a list of the registered Provider-objects. Each Provider-object gives a list of algorithm-denotations associated with a different range of implemented algorithms. A Provider-object associates name-strings "SHA1withDSA" with fully-qualified class-names like `sun.security.provider.DSA`. The list of providers is successively searched for engine-type, algorithm-implementation and possibly for the concrete provider. (The factory-method `getInstance("SHA1withDSA", "SUN")` explicitly asks for the implementation of the algorithm found in the sun-package.) The first provider that matches those specifications then automatically furnishes the implementation of the engine.

Engine-classes are classes that use the Java Provider-Infrastructure for selecting their implementation. An application-programmer usually works with engine-classes and engines. Whereas infrastructure-programming means to furnish providers and the implementations of algorithms, the Security-Provider-Interfaces (SPIs).

Engine: Engines are objects that implement (cryptographic) algorithms. Engine-classes use the Java Provider-Infrastructure to select a concrete class called Security-Provider-Interface to implement an engine(-object). As specified by the string-argument of the engine-class's `getInstance(...)`-method, the `java.security.Security`-class selects the appropriate Security-Provider-Interfaces from the `java.security.Provider`-objects.

A Provider-object basically is just a list; this list contains the names of package-members having been made identifiable by giving them a standardized name. A provider-class extends the `java.util.Properties`-class, an aggregation of (String, String)-pairs, which extends the `java.util.Hashtable`-class.

The CLASS ACTUALLY IMPLEMENTING THE (CRYPTOGRAPHIC) ALGORITHM extends a class that has the same name as the engine-class AND is postfixed with "SPI" for Security-Provider-Interface.

The following program can be used to get information about the security-providers available in the current Java Runtime Environment (JRE):

```
// Prints out actually available implementations of engine-classes.
class ListAvailableEngines
{
    public static void main( String unused[] )
    {
        java.security.Provider[] providerList = java.security.Security.getProviders();
        java.security.Provider p = null;
        for( int i = 0; i < providerList.length; i++ ) // steps provider-wise
        {
            p = providerList[i];
            System.out.println( "\n" + p + "\n" + p.getInfo());
            p.list( System.out ); // lists provider-entries
        }
    }
}
```

```
/*
jacobneuhaus@Linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity > javac ListAvailableEngines.java
jacobneuhaus@Linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity > java ListAvailableEngines

SUN version 1.2
SUN (DSA key/parameter generation; DSA signing; SHA-1, MD5 digests; SecureRandom; X.509 certificates; JKS keystore)
-- listing properties --
Alg.Alias.KeyFactory.1.2.840.10040.4.1=DSA
Alg.Alias.Signature.1.2.840.10040.4.3=SHA1withDSA
Alg.Alias.KeyPairGenerator.OID.1.2.840.10040.4.1=DSA
Signature.SHA1withDSA KeySize=1024
Signature.SHA1withDSA ImplementedIn=Software
Alg.Alias.KeyFactory.1.3.14.3.2.12=DSA
AlgorithmParameterGenerator.DSA=sun.security.provider.DSAParameterGen...
CertificateFactory.X509 ImplementedIn=Software
Alg.Alias.KeyPairGenerator.1.3.14.3.2.12=DSA
Alg.Alias.Signature.SHA/DSA=SHA1withDSA
Alg.Alias.Signature.1.3.14.3.2.13=SHA1withDSA
Alg.Alias.CertificateFactory.X.509=X509
SecureRandom.SHA1PRNG=sun.security.provider.SecureRandom
Alg.Alias.Signature.DSS=SHA1withDSA
KeyStore.JKS ImplementedIn=Software
KeyFactory.DSA ImplementedIn=Software
MessageDigest.SHA ImplementedIn=Software
Signature.SHA1withDSA=sun.security.provider.DSA
AlgorithmParameters.DSA=sun.security.provider.DSAParameters
MessageDigest.SHA=sun.security.provider.SHA
CertificateFactory.X509=sun.security.provider.X509Factory
Alg.Alias.AlgorithmParameters.1.3.14.3.2.12=DSA
MessageDigest.MD5 ImplementedIn=Software
AlgorithmParameterGenerator.DSA ImplementedIn=Software
Alg.Alias.Signature.DSA=SHA1withDSA
KeyStore.JKS=sun.security.provider.JavaKeyStore
Alg.Alias.Signature.DSAwithSHA1=SHA1withDSA
MessageDigest.MD5=sun.security.provider.MD5
KeyPairGenerator.DSA=sun.security.provider.DSAKeyPairGener...
Alg.Alias.Signature.SHAwithDSA=SHA1withDSA
Alg.Alias.Signature.OID.1.2.840.10040.4.3=SHA1withDSA
KeyPairGenerator.DSA ImplementedIn=Software
Alg.Alias.Signature.SHA1/DSA=SHA1withDSA
Alg.Alias.KeyPairGenerator.1.2.840.10040.4.1=DSA
Alg.Alias.MessageDigest.SHA-1=SHA
AlgorithmParameters.DSA ImplementedIn=Software
Alg.Alias.MessageDigest.SHA1=SHA
KeyPairGenerator.DSA KeySize=1024
Alg.Alias.AlgorithmParameters.1.2.840.10040.4.1=DSA
KeyFactory.DSA=sun.security.provider.DSAKeyFactory
Alg.Alias.Signature.1.3.14.3.2.27=SHA1withDSA
```

```
SecureRandom.SHA1PRNG ImplementedIn=Software
AlgorithmParameterGenerator.DSA KeySize=1024
Alg.Alias.Signature.SHA-1/DSA=SHA1withDSA

SunRsaSign version 1.0
SUN's provider for RSA signatures
-- listing properties --
Signature.MD5withRSA=com.sun.rsa.jca.JSA_MD5RSASignature
KeyFactory.RSA=com.sun.rsa.jca.JSA_RSAPrivateKeyFactory
Signature.MD2withRSA=com.sun.rsa.jca.JSA_MD2RSASignature
Signature.SHA1withRSA=com.sun.rsa.jca.JSA_SHA1RSASignature
KeyPairGenerator.RSA=com.sun.rsa.jca.JSA_RSAPrivateKeyPairGenerator

SunJCE version 1.2
SunJCE Provider (Implements DES, Triple DES, Blowfish, PBE, Diffie-Hellman, HMAC-MD5, HMAC-SHA1)
-- listing properties --
Cipher.DES=com.sun.crypto.provider.DESCipher
KeyStore.JCEKS=com.sun.crypto.provider.JceKeyStore
Alg.Alias.SecretKeyFactory.TripleDES=DESede
KeyGenerator.HmacSHA1=com.sun.crypto.provider.HmacSHA1KeyGe...
SecretKeyFactory.DES=com.sun.crypto.provider.DESKeyFactory
Alg.Alias.KeyFactory.DH=DiffieHellman
KeyGenerator.DESede=com.sun.crypto.provider.DESedeKeyGene...
Mac.HmacMD5=com.sun.crypto.provider.HmacMD5
Cipher.Blowfish=com.sun.crypto.provider.BlowfishCipher
Mac.HmacSHA1=com.sun.crypto.provider.HmacSHA1
Cipher.DESede=com.sun.crypto.provider.DESedeCipher
AlgorithmParameters.DESede=com.sun.crypto.provider.DESedeParameters
Alg.Alias.AlgorithmParameters.TripleDES=DESede
KeyPairGenerator.DiffieHellman=com.sun.crypto.provider.DHKeyPairGene...
KeyFactory.DiffieHellman=com.sun.crypto.provider.DHKeyFactory
AlgorithmParameterGenerator.DiffieHellman=com.sun.crypto.provider.DHParameterGe...
AlgorithmParameters.PBE=com.sun.crypto.provider.PBEParameters
Alg.Alias.AlgorithmParameters.PBEwithMD5AndDES=PBE
Alg.Alias.KeyAgreement.DH=DiffieHellman
Alg.Alias.Cipher.TripleDES=DESede
AlgorithmParameters.DES=com.sun.crypto.provider.DESParameters
SecretKeyFactory.DESede=com.sun.crypto.provider.DESedeKeyFactory
SecretKeyFactory.PBEwithMD5AndDES=com.sun.crypto.provider.PBEKeyFactory
KeyGenerator.HmacMD5=com.sun.crypto.provider.HmacMD5KeyGen...
Cipher.PBEwithMD5AndTripleDES=com.sun.crypto.provider.PBEwithMD5And...
AlgorithmParameters.Blowfish=com.sun.crypto.provider.BlowfishParam...
KeyGenerator.DES=com.sun.crypto.provider.DESKeyGenerator
Alg.Alias.KeyGenerator.TripleDES=DESede
KeyGenerator.Blowfish=com.sun.crypto.provider.BlowfishKeyGe...
Alg.Alias.KeyPairGenerator.DH=DiffieHellman
KeyAgreement.DiffieHellman=com.sun.crypto.provider.DHKeyAgreement
Alg.Alias.AlgorithmParameterGenerator.DH=DiffieHellman
Alg.Alias.AlgorithmParameters.DH=DiffieHellman
Cipher.PBEwithMD5AndDES=com.sun.crypto.provider.PBEwithMD5And...
AlgorithmParameters.DiffieHellman=com.sun.crypto.provider.DHParameters
jacobsneuhaus@linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity >
/
```

Above can be seen few entries that actually point to an engine-implementation class. Most entries begin with “Alg.alias.” and POINT TO OTHER ENTRIES, this frees the user of the engine-classes from having only one single specification-string available but, as can be seen in this case, all labels that follow the “Alg.alias.Signature.” mean the same, namely “Signature.SHA1withDSA”! The labels that follow OID (Object Identifier) are used by the government of the United States of America.

The default provider-classes are found in the JAR-file that also contains the core-class-files of the Java Runtime System (rt.jar).



To INSTALL A NEW PROVIDER-CLASS AND ITS ASSOCIATED CLASS-FILES put all those class-files or the entire package as a JAR-archive into a directory of the CLASS-PATH-variable or the ext-folder and register the provider: There are two possible ways of registering the provider:

How to install a new provider

Persistent (Once registered, the provider will be available during different computer sessions and during different lifetimes of Java Runtime Environments.) A user/administrator enters data into the java.security-file as described in the diagram above, titled Java Provider Infrastructure.

Dynamic (the provider can only be referenced in the time that remains between the registering during the program-run and the end of the respective Java Runtime Environment) A Java program calls the method java.Security.addProvider(providerName) during run-time; this changes the provider-list of the java.security.Security-class from a Java program (if a security-manager is loaded, then the appropriate permission has to be given).

The Java Provider-Infrastructure usually is not used in browsers, there the used algorithms tend to be fixed in advance.

Provider-Infrastructure: The process by which an *engine-class*’s getInstance(...)-method triggers the class java.security.Security into a selecting a class called Security-Provider-Interface from which is derived an *engine-object*, its reference finally being returned by the getInstance(...)-method. The selection-process consults lists of classes (Security-Provider-Interfaces), these lists are given by the corresponding java.security.Provider-objects. The byte-code of those classes – the Security-Provider-Interfaces also called SPI-classes – is given within JAR-files.

Finally observe, that for an application-programmer using a given Provider-Infrastructure just means the following: From the list which is given by the program above, choose the name-string that denotes the desired cryptographic algorithm to be used. Using the appropriate engine-class, invoke the static

getInstance(...)-method with the chosen name-string. This method returns the reference to an engine-object that implements the chosen algorithm.

User/Administrator: Key-Management and Signing Files

key-store

User/Administrator: Key-Management and Signing Files — A key-store can be seen as a variation of an address-book. A key-store gathers identities, certificates together with means of securely contacting²¹⁵ the associated persons or being securely contacted²¹⁶ by them. Elements of entries can be classified as follows:

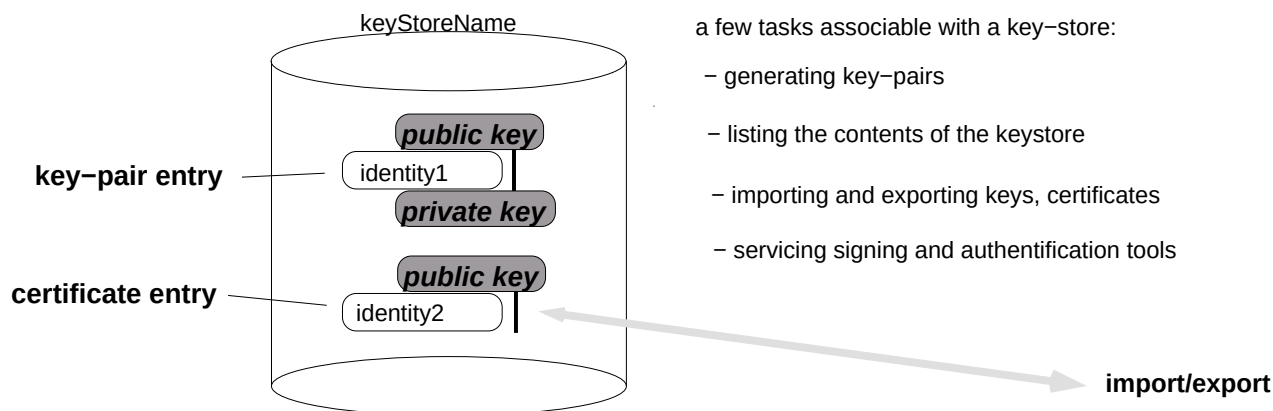
identities, in the simplest case an identity is given by a name of a person, firm or other entity. Often an identity is made unique by supplementing the name with further characterizing information, like date of birth, date of incorporation, mailing-address, fingerprint, ... Identities usually are given by aliases or Distinguished Names (DN according to the convention called X.500).

key-pair entries represent stored key-pairs together with the information about the identity of the owner.

certificate entries contain a public key together with the information about the identity of the owner PLUS the signatures of that data which affirm that the owner of the corresponding private key actually has the given identity. (certificates and certification have been introduced beginning with page 483)

(→) Own private keys are used for signing data to be sent, imported public keys or rather imported certificates (signed public keys) are used for encrypting data to be sent. (←) On receiving data, own private keys are used to decrypt data and imported certificates are used to authenticate incoming signed data. — Some key-stores provide additional functionality:

Key-Store: Entries and Services



A user or an administrator may use applications for producing key-pairs, for storing public keys and key-pairs persistently, and for retrieving keys: Such an application — a kind of incorporated data-base-like management-system — is also called **key-store**.

key-store

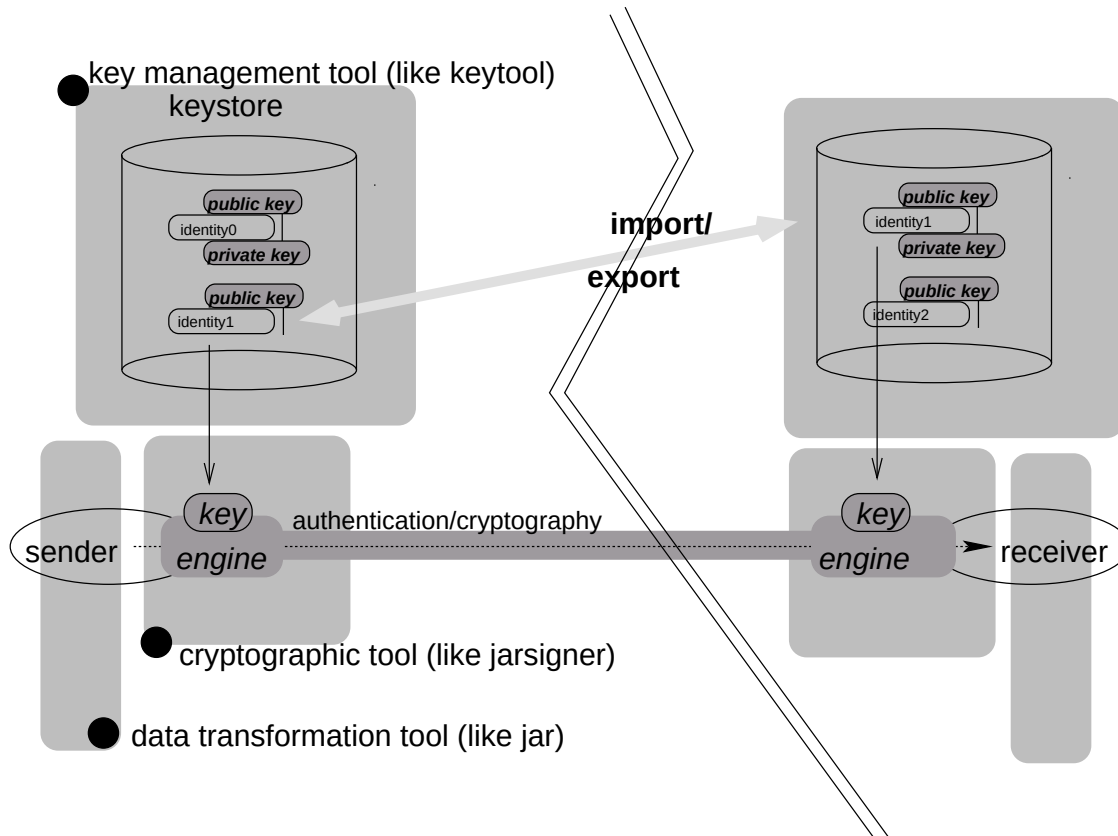
Storing a key or key-pair inside a key-store is also called **importing** the key or key-pair. — Key retrieval is also described as **exporting** keys from the store (Names of formats have been given beginning with page 481). — Then there are tools, or features of a tool, that enable the user or administrator to produce a signature of a file using their private key. — This includes the functionality to use a public key on a *signature* together with the *associated file* for verifying that signature, and thus to authenticate the *corresponding file*.

Remember that authentication means (Sender-related information is written in *italics*, receiver-related information is written in *small-caps*): First (data-integrity), the **PUBLIC KEY** and the *signature* prove that the *file* is the same *file* for which the *signature* has been produced, thus the *file* alone has not been changed during transmission! Second (identity bound to sender), presumably the *owner of the private key* who produced the *file's signature* is closely associated with the *identity* for which she or he is known.

²¹⁵For example, a public key of a person used by others to encrypt a message. Then only that person can decrypt the message using hers or his private key.

²¹⁶If the sender of a message has encrypted the message with hers or his private key, then the message can be decrypted only with the associated public key, thereby authenticating the message.

Thus the *person*, with which the PUBLIC KEY is associated, signed that file, so that person might be no impostor!



The Java Development Kit (JDK) furnishes some basic tools for working with digital signatures. The three tools are keytool, jar and jarsigner. The Java Platform API Specification [JPAPIS] offers extensive information about how to use these tools.

Java tools

The keytool-program lets a user manage a database of key-pairs and public keys (rather given as certificates). The keytool-program even allows users or administrators to *create* new key-pairs. The jar-tool, as has been mentioned on page 325, enables the user to ZIP-compress multiple files into one file and to view and unpack that file. Finally, the jarsigner-tool lets users and administrators digitally sign JAR-files and lets them verify such signatures.

Of course there are imaginable other concepts of key-stores and signing utilities. Key-stores may be distributed in safe- or local networks. Programs like PGP (Pretty Good Privacy, <http://www.pgp.org>) furnish all those functionalities within a Graphical User Interface. A Java application-programmer may write a counterpart in Java using the `java.security`-package.

Formats: Remember (page 481) that the key-pairs and public keys are associated with their owners by a short “alias” and a more explicit Distinguished Name (DN specified in X.500). JAVA’S keytool-TOOL EXPORTS AND IMPORTS KEY-ENTRIES ONLY IN THE RFC1421-FORMAT! A standard format for public keys (or certificates) is given by the X.509 standard that uses the so-called DER-encoding. Private keys often use the PKCS#8-format. Similarly there occur different types of the key-store databases. The Java keytool uses the JKS-type of database (can be set and seen in the `java.security`-file: `key-store.type=jks`). In such key-stores the private keys are stored in an encrypted form as can be checked by inspecting the key-store with a hex-editor.

How does a user or administrator create a key-pair and store it into a (newly created) key-store? An example of the commands to be issued within a terminal-window of the operating-system is given below:

```
# sharp symbol leads outcommented lines, especially outcommented output and input
## double sharp indicates pure comments

## file: keytoolCreateEntry.txt
## (unlike the printout below might suggest, the command contains NO return, it is one line!)

# [localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% keytool -genkey
-alias jacobneuhau -keypass marzia -keystore localKeystore -storepass oopenn
# What is your first and last name?
# [Unknown]: Jacob Neuhaus
# What is the name of your organizational unit?
```

```
# [Unknown]:
# What is the name of your organization?
# [Unknown]:
# What is the name of your City or Locality?
# [Unknown]: Venice
# What is the name of your State or Province?
# [Unknown]: Venice
# What is the two-letter country code for this unit?
# [Unknown]: IT
# Is <CN=Jacob Neuhaus, OU=Unknown, O=, L=Venice, ST=Venice, C=IT> correct?
# [no]: y
#
# [localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% keytool -genkey
# -alias williammartin -keypass mincemeat -keystore localKeystore -storepass oopenn
# What is your first and last name?
# [Unknown]: William Martin
# What is the name of your organizational unit?
# [Unknown]: Combined Operations
# What is the name of your organization?
# [Unknown]: Royal Marines
# What is the name of your City or Locality?
# [Unknown]: Cardiff (Wales)
# What is the name of your State or Province?
# [Unknown]: United Kingdom
# What is the two-letter country code for this unit?
# [Unknown]: UK
# Is <CN=William Martin, OU=Combined Operations, O=Royal Marines, L=Cardiff (Wales),
# ST=United Kingdom, C=UK> correct?
# [no]: y
#
# [localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro%
## file: keytoolCreateEntry.txt
```

When key-pairs have been generated and stored, then the contents of the key-store can be viewed:

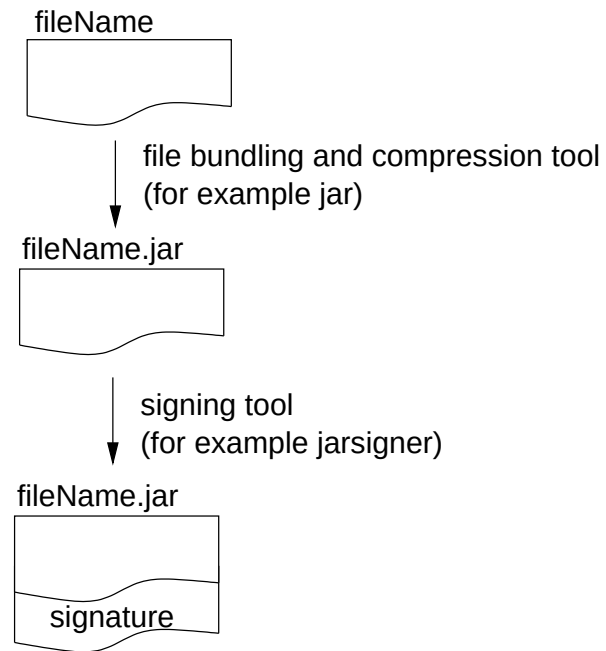
```
# sharp symbol leads outcommented lines, especially outcommented output and input
## double sharp indicates pure comments

## file: keytoolReadEntries.txt
# [localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% keytool -list
# -keystore localKeystore -storepass oopenn
#
# Keystore type: jks
# Keystore provider: SUN
#
# Your keystore contains 2 entries:
#
# jacobneuhaus, Tue Jun 19 15:08:01 CEST 2001, keyEntry,
# Certificate fingerprint (MD5): E4:46:AC:2C:A3:A2:B3:9E:0B:D0:94:8A:44:A0:A8:A9
# williammartin, Tue Jun 19 15:18:16 CEST 2001, keyEntry,
# Certificate fingerprint (MD5): 33:DE:AB:88:3C:BE:06:26:98:01:1C:29:24:2B:F9:03
#
# [localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro%
```

Then a public key may be exported from the key-store:

```
## file: keytoolExportEntry.txt
# [localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% keytool -export
# -keystore localKeystore -storepass oopenn -alias jacobneuhaus -file jneuhaus.cer
# Certificate stored in file named <jneuhaus.cer>
# [localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro%
```

By using a private key, a file may be signed. For that follow the process described in the graphic:



```

## file: SignWithJarsigner.txt
## A file with the name 'INFORMATION.txt' is given.
## First the file is transformed with the jar-tool, then the resulting JAR-file is signed:
# [localhost:~/JavaCodeFiles/JRESecurity/
    DataCodingSecurity] andreadipietro% jar -cf INFORMATION.jar INFORMATION.txt
# [localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% jarsigner
    -keystore localKeystore -storepass oopenn
    -keypass mincemeat -signedjar INFORMATIONSigned.jar INFORMATION.jar williammartin
# [localhost:~/JavaCodeFiles/
    JRESecurity/DataCodingSecurity] andreadipietro% jar -tf INFORMATIONSigned.jar
# META-INF/MANIFEST.MF
# META-INF/WILLIAMM.SF
# META-INF/WILLIAMM.DSA
# META-INF/
# INFORMATION.txt
# [localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro%

## In Windows to be run as batch-file, due to an entry too long complaint

## Extract the listed contents above by entering:
# [localhost:~/JavaCodeFiles/
    JRESecurity/DataCodingSecurity] andreadipietro% jar -xf INFORMATIONSigned.jar
  
```

Signed JAR-File

Above can be seen the contents of the JAR-file: The original data-file, and inside a subdirectory META-INF, the manifest-file which every JAR-file contains, and two files that have been generated by the signer (base-names are the first eight upper-case letters of the signer's alias):

1. The manifest-file (.MF-postfixed) lists the names of the files in the JAR-container-file, together with individually generated message digests.
2. The signer-specific, .SF postfixed file, the so-called "Signature File", lists all the files that have been signed, together with individual message digests created from the respective entries (file-name and associated message digest) in the manifest-file.
3. The second signer-specific file, the binarily encoded "(signature) block-file" (with a postfix that informs about the signing algorithm used, here ".DSA") finally signs the previous list. That "(signature) block-file" file contains the signature of the associated SF-file and a certificate of the signer in algorithm-specific formats²¹⁷.

(This three-step/three-file procedure makes it possible to sign individual member-files in a JAR-file and to make it possible to have multiple signers for arbitrarily chosen subsets of member files.)

Now verify the signed file using the certificate associated with the JAR-file AND the certificate stored inside the key-store. The precise result of the verification is indicated by the "sm" term below:

²¹⁷That means for .DSA (PKCS#7). For more about format specifications see the Java Platform API Specification [JPA-PIS] on formats of JAR-files and their manifests (online documentation of packages java.util.jar or java.util.zip) and on formats of certificates.

```

## file: AuthenticateWithJarsigner.txt
# C:\WINDOWS\JavaCodeFiles\JRESecurity\DataCodingSecurity>jarsigner
      -verbose -verify -certs -keystore localKeystore INFORMATIONSigned.jar
#
#      136 Thu Aug 02 12:31:20 CEST 2001 META-INF/MANIFEST.MF
#      189 Thu Aug 02 12:32:06 CEST 2001 META-INF/WILLIAMM.SF
#      1098 Thu Aug 02 12:32:06 CEST 2001 META-INF/WILLIAMM.DSA
#           0 Thu Aug 02 12:29:02 CEST 2001 META-INF/
# sm      2680 Wed Aug 01 15:47:52 CEST 2001 INFORMATION.txt
#
#      X.509, CN=William Martin, OU=Combined Operations, O=Royal Marines,
#                               L=Cardiff (Wales), ST=United Kingdom, C=UK
#
#
#      s = signature was verified
#      m = entry is listed in manifest
#      k = at least one certificate was found in keystore
#      i = at least one certificate was found in identity scope
#
#      jar verified.
#
# C:\WINDOWS\JavaCodeFiles\JRESecurity\DataCodingSecurity>

```

If the option `-keystore localKeystore` is omitted, then no key-store is searched and the verification is done only using the certificate furnished by the signer in the JAR-file, indicated by the “sm” term.

Additional functionalities and further options are described in the   Java Platform API Specification [JPAPIS] under the category “tools”. Rewriting the examples above for a concrete operating-system may include to use the default key-store (`.keystore`)²¹⁸ in the user’s home directory. Then the options `-keystore` and `-storepass` can be omitted in the examples above. The examples above may be revised using these options.

Access-Controlling with Signed Data

Access-Controlling with Signed Data — Java access-controlling makes it possible to grant a permission only if the code in question has been properly signed with some private key. This requires the Java Runtime Environment to know where to find key-stores with the associated public key. The user/administrator can make the Java Runtime Environment find a key-store by adding one single key-store-entry per policy-file (see page 459) outside the grant entries:

 policy file

`keystore URLofKeystore, typeOfKeystore_for_example_ "jks"`

If the Uniform Resource Locator is a relative one, then the path is taken with respect to the policy-file’s URL. This entry makes it possible to add the optional `signedby SignerAlias` entry into a grant-statement of the policy-file.

User/Administrator: How to encrypt Files

User/Administrator: How to encrypt Files — The Java Development Kit (JDK) (Java Version 1.3) or the Java Cryptography Extension (JCE) furnishes no tool for an end-user to encrypt files.

Application-Programmer: Signing and Encrypting

Application-Programmer: Signing and Encrypting — The cryptography-related tools (`jar`, `keytool`, `jarsigner`) for helping users or administrators, as furnished by the Java Development Kit (JDK) also can be programmed in Java. This would require to also use the following packages: `java.util.jar`, `java.util.zip`, and the following signature-related classes which have been given the names `java.security.KeyStore`, `java.security.KeyGenerator`, `java.security.Signature`. The section below presents an introduction to those classes that are found mostly in the security-package. Some classes (`java.security.KeyStore`, `java.security.KeyGenerator`, `java.security.Signature`) are engine-classes, with implementations already given inside provider-packages, which have been readily written by infrastructure-programmers. So this section rather focuses on how to use engine-classes than on how to write them.

²¹⁸In Unix-like operating-systems (Linux, Solaris), the prefixing period (.) makes the file of the default key-store “invisible”.

Remember that the Java Security Provider Infrastructure can be used to enrich application-programs with cryptographic- and authenticating elements (page 485).

The following program presents the use of the two engine-classes `java.security.KeyPairGenerator` and `java.security.Signature`. The class mentioned first, controls the selection of an engine for producing a `KeyPair`-object, from which can be taken the objects of type `PublicKey` and `PrivateKey`. CAUTION: Note that `Signature`-objects represent signing-engines and not the signatures of some data, the actual signatures of data are stored below as an array of bytes.

```
class SignThisFile
{ public static void main( String fileName[] ) throws Exception
  { String nameOfFileToBeSigned = fileName[0];

  // ----- Generate a key-pair.
    java.security.KeyPairGenerator theKeyGenerator =
      java.security.KeyPairGenerator.getInstance( "DSA", "SUN" );
    java.security.SecureRandom random =
      java.security.SecureRandom.getInstance( "SHA1PRNG", "SUN" );
    theKeyGenerator.initialize( 1024, random );
    java.security.KeyPair theKeyPair = theKeyGenerator.generateKeyPair();
    java.security.PrivateKey thePrivateKey = theKeyPair.getPrivate();
    java.security.PublicKey thePublicKey = theKeyPair.getPublic();

  // ----- Instantiate a Signature engine-object and initialize it with the private key.
    java.security.Signature signingEngineDSA =
      java.security.Signature.getInstance( "SHA1withDSA", "SUN" );
    signingEngineDSA.initSign( thePrivateKey );
  // -----Read in a file as a buffered stream of bytes;
  // choose the size of the buffer to be the same as the size of a cryptographic-block.
    java.io.FileInputStream fis = new java.io.FileInputStream( nameOfFileToBeSigned );
    java.io.BufferedInputStream bufis = new java.io.BufferedInputStream( fis );
    byte[] miniBuffer = new byte[1024];
    int howFull;
    while( bufis.available() != 0 )
    { howFull = bufis.read( miniBuffer );
      signingEngineDSA.update( miniBuffer, 0, howFull );
    }
    bufis.close();
  // ----- All data has been processed by the engine; now generate a signature:
    byte[] binarySignatureDSA = signingEngineDSA.sign();
  // ----- Save the signature in a file named signatureDSA:
    java.io.FileOutputStream fos =
      new java.io.FileOutputStream( nameOfFileToBeSigned + "SignatureDSA" );
    fos.write( binarySignatureDSA );
    fos.close();
  // -----Save the public key in a file.
    byte[] binaryOfEncodedPublicKey = thePublicKey.getEncoded();
    java.io.FileOutputStream keyfos =
      new java.io.FileOutputStream( nameOfFileToBeSigned + "PublicKeyDSA" );
    keyfos.write( binaryOfEncodedPublicKey );
    keyfos.close();
  } }
/*
----- The output are two additional files placed in the current directory:
INFORMATION.txtSignatureDSA
INFORMATION.txtPublicKeyDSA
All three files together, including INFORMATION.txt
have to be transferred for authentication elsewhere.
*/

/*
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity > javac SignThisFile.java
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity > java SignThisFile INFORMATION.txt
andreadipietro@linux:~/JavaCodeFiles/JRESecurity/DataCodingSecurity >
*/
```

Enter the name of the file to be signed and the program produces a signature-file and a public-key-file.

These three files have to be referenced by the program below to verify the “signed data”. Here is used a derivate class of the class `java.security.spec.EncodedKeySpec` to produce a DER-encoded-key (X.509) that is recognized by the default key-factory. A **key-factory** is an instance of the engine-class `java.security.KeyFactory` used for reversibly transforming or converting key-formats within the Java Runtime Environment (*contrary* to a `KeyPairGenerator` that produces key-pairs!). Here again THE ACTUAL SIGNATURE REMAINS A PURE ARRAY OF BYTES.

```

/* SEE ALSO the text-file INFORMATION.txt
AuthenticateSignedData.class needs the file-name of the
1. encoded public key
2. signature
3. data thus signed
*/
class AuthenticateSignedData
{ public static void main( String threeInputData[] ) throws Exception
  { String fileNameOfEncodedPublicKey = threeInputData[0];
    String fileNameOfSignature = threeInputData[1];
    String fileNameOfData = threeInputData[2];

    // ----- Read in binaries of encoded public key:
    java.io.FileInputStream fisKey =
        new java.io.FileInputStream( fileNameOfEncodedPublicKey );
    byte[] binaryOfEncodedKey = new byte[ fisKey.available() ];
    fisKey.read( binaryOfEncodedKey );
    fisKey.close();

    // -----
    // Create a public key with the KeyFactory engine and the Encoding specification class.
    java.security.spec.X509EncodedKeySpec defaultSpecificationOfPublicKeys =
        new java.security.spec.X509EncodedKeySpec( binaryOfEncodedKey );
    java.security.KeyFactory keyFactory =
        java.security.KeyFactory.getInstance( "DSA", "SUN" );
    java.security.PublicKey thePublicKey =
        keyFactory.generatePublic( defaultSpecificationOfPublicKeys );

    // ----- Read in the binaries of a signature-file
    // (this signature should authenticate the data-file):
    java.io.FileInputStream fisSign = new java.io.FileInputStream( fileNameOfSignature );
    byte[] binarySignatureOfData = new byte[ fisSign.available() ];
    fisSign.read( binarySignatureOfData );
    fisSign.close();

    // ----- Create a Signature-object and initialize it with the public key.
    java.security.Signature signingEngine =
        java.security.Signature.getInstance( "SHA1withDSA", "SUN" );
    signingEngine.initVerify( thePublicKey );

    // ----- Update and verify the data.
    java.io.FileInputStream fisData = new java.io.FileInputStream( fileNameOfData );
    java.io.BufferedInputStream bufis = new java.io.BufferedInputStream( fisData );
    byte[] miniBuffer = new byte[1024];
    int howFull;
    while( bufis.available() != 0 )
    { howFull = bufis.read( miniBuffer );
      signingEngine.update( miniBuffer, 0, howFull );
    };
    bufis.close();
    boolean authentic = signingEngine.verify( binarySignatureOfData );
    System.out.println( "data autenticated: " + authentic );
  } }
}
/*
(PREQUISITE PREREQUISITE PREREQUISITE PREREQUISITE)
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% javac SignThisFile.java
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% java SignThisFile INFORMATION.txt
(PREQUISITE END PREREQUISITE END PREREQUISITE END)

[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% javac AuthenticateSignedData.java
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% (formatting linebreak:)
java AuthenticateSignedData INFORMATION.txtPublicKeyDSA INFORMATION.txtSignatureDSA INFORMATION.txt
data autenticated: true
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro%
*/

```

If text-files can be signed, as shown above, then any file can be signed, especially class-files which contain Java byte-code. As has been mentioned before, policies can request loaded classes have their code-sources signed. Thus if “classes can be signed” that way, then one may wonder about signing individual objects within a Java Runtime Environment! This is shown in the program below using the engine-class `java.security.KeyStore` to load an existent key-store into the Java Runtime Environment and reading out the public key which is wanted for signing. Observe also that the private key can be read out relatively straightforward whereas public keys can be accessed only by first reading the `java.security.cert.Certificate`²¹⁹-object and THEN extracting the `java.security.PublicKey`-object! Also observe again that the `Signature`-object is a signing-engine, the actual signature it produces is encapsulated in the object of type `SignedObject`.

²¹⁹The other type of interface-type `java.security.Certificate` adheres to security-concepts as realized with the Java Version 1.1. These are outside the reach of this text.

```
// (PREREQUISITE) needs a key-store of name "localKeystore"
// (PREREQUISITE) with entries used below:

class SignThisObject
{ public static void main( String unused[] ) throws Exception
{
// ----- Get a keypair from a keystore file:
    java.security.KeyStore theKeyStore = java.security.KeyStore.getInstance( "JKS" );
    String fileName = "localKeystore";
    String keyStorePassword = "oopenn";
    java.io.FileInputStream fis = new java.io.FileInputStream( fileName );
    theKeyStore.load( fis, keyStorePassword.toCharArray() );
    String alias = "williammartin";
    String password = "mincemeat";
    java.security.PrivateKey thePrivateKey =
        (java.security.PrivateKey)theKeyStore.getKey( alias, password.toCharArray() );
    java.security.cert.Certificate theCertificate = theKeyStore.getCertificate( alias );
    java.security.PublicKey thePublicKey = theCertificate.getPublicKey();

// ----- Instantiate signing engine, hope that a provider supports the algorithm.
    String algorithm = thePrivateKey.getAlgorithm();
    System.out.println( algorithm ); // probably DSA
    java.security.Signature signingEngine =
        java.security.Signature.getInstance( algorithm );

// ----- Object to be signed:
    String textToBeSigned =
"The bearer of this object hereby has the right to do what has to be done.";
// ----- The signing:
    java.security.SignedObject theSignedStringObject =
        new java.security.SignedObject( textToBeSigned, thePrivateKey, signingEngine );
// ----- The verification of the signature:
    String algorithm2 = theSignedStringObject.getAlgorithm();
    java.security.Signature verificationEngine =
        java.security.Signature.getInstance( algorithm2 );

    boolean verified = false;
    verified = theSignedStringObject.verify( thePublicKey, verificationEngine );
    System.out.println( "signature of object is verified? " + verified );
} }

/*
C:\WINDOWS\JavaCodeFiles\JRESecurity\DataCodingSecurity>javac SignThisObject.java

C:\WINDOWS\JavaCodeFiles\JRESecurity\DataCodingSecurity>java SignThisObject
DSA
signature of object is verified? true

C:\WINDOWS\JavaCodeFiles\JRESecurity\DataCodingSecurity>
*/
```

As could be seen above, the engine-class `java.security.KeyStore` can be used to read any other key-store that has a defined key-store format like `jks`. Similar to the notions of a guarded- and signed object, the following JCE also introduces sealed-objects, which won't be investigated further.

How can a Java programmer read out information of a key-store? (that means emulating parts of the `keytool`-program) This is relatively straightforward when using the code of the previous examples:

```
// SEE ALSO the text-file INFORMATION.txt
// needs a pre-existent key-store of type JKS

class ListKeystore
{ public static void main( String storeAccess[] ) throws Exception
{ String keystoreFileName = storeAccess[0];
  String keyStorePassword = storeAccess[1];
  java.security.KeyStore theKeyStore = java.security.KeyStore.getInstance( "JKS" );
  java.io.FileInputStream fis = new java.io.FileInputStream( keystoreFileName );
  theKeyStore.load( fis, keyStorePassword.toCharArray() );
  String alias = null;
  java.security.cert.Certificate c[] = null;
  for( java.util.Enumeration e = theKeyStore.aliases(); e.hasMoreElements(); )
  { alias = (String)e.nextElement();
    System.out.println(alias);
    c = theKeyStore.getCertificateChain( alias );
    for( int i = 0; i < c.length; i++ )
    { System.out.println( c[i].toString() );
    }
  } } }

/*
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% javac ListKeystore.java
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% java ListKeystore localKeystore oopenn
```

```

jacobneuhau
[
  [
    Version: V1
    Subject: CN=Jacob Neuhaus, OU=Unknown, O=, L=Venice, ST=Venice, C=IT
    Signature Algorithm: SHA1withDSA, OID = 1.2.840.10040.4.3

    Key: Sun DSA Public Key
    Parameters: DSA
      p: fd7f5381 1d751229 52df4a9c 2eece4e7 f611b752 3cef4400 c31e3f80 b6512669
      455d4022 51fb593d 8d58fabf c5f5ba30 f6cb9b55 6cd7813b 801d346f f26660b7
      6b9950a5 a49f9fe8 047b1022 c24fbbab d7feb7c6 1bf83b57 e7c6a8a6 150f04fb
      83f6d3c5 1ec30235 54135a16 9132f675 f3ae2b61 d72aeff2 2203199d d14801c7
      q: 9760508f 152390cc b292b982 a2eb840b f0581cf5
      g: f7e1a085 d69b3dde cbbcab5c 36b857b9 7994afbb fa3aea82 f9574c0b 3d078267
      5159578e bad4594f e0710710 8180b449 16712308 4c281613 b7cf0932 8cc8a6e1
      3c167a8b 547c8d28 e0a3ae1e 2bb3a675 916ea37f 0bfa2135 62f1fb62 7a01243b
      cca4f1be a8519089 a883dfe1 5ae59f06 928b665e 807b5525 64014c3b fecf492a

    Y:
      3d7c5cbd df667a02 2aaacc11 113be12c 62c642ef 4bd2c451 df7e28d9 4a833a5e
      240e06f3 3ce07618 b40f41d4 fbeb09e3 42648bea 93a4c256 205800fb e333e8c5
      1884a94c cbe72e44 9ce4e70d c274f994 9438e4c4 a9507273 043be9bf 364e4655
      b326e3c1 df0a08db 2eb94fd6 6d59be09 9d1c36cd dc183c7f 7584a733 601c3a6f

    Validity: [From: Tue Jun 19 15:07:51 CEST 2001,
      To: Mon Sep 17 15:07:51 CEST 2001]
    Issuer: CN=Jacob Neuhaus, OU=Unknown, O=, L=Venice, ST=Venice, C=IT
    SerialNumber: [ 3b2f4ea7 ]

  ]
  Algorithm: [SHA1withDSA]
  Signature:
    0000: 30 2E 02 15 00 93 07 CD A9 8F 5C E5 AC D2 95 01 0.....\.....
    0010: 8F F7 78 B1 A4 DB A1 6D 6C 02 15 00 8D DC 82 18 ..X.....mL.....
    0020: 27 42 01 D4 45 E1 A3 54 C3 9B 2D F5 CD EE 51 BC 'B...E...T...Q.

]
williammartin
....
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro%
^/

```

Installing the Java Cryptography Extension (JCE)



Installing the Java Cryptography Extension (JCE) — This should become unnecessary with Java Version 1.4 and higher, when the JCE may be an integral part of the Java Runtime Environment. With Java versions of Number 1.3.x get a folder or file containing the Java Cryptography Extension²²⁰. Extract the received JAR-file into a little directory-structure that contains documentation and jarred class-files.

FIRST: Put the extracted relevant JAR-files into the `${java.home}/${lib}/ext`-directory (in UNIX-derivates for example `/usr/local/jdk1.3.0/jre/lib/ext`, in Windows `jdk1.3.0\jre\lib\ext`)²²¹.

SECOND: Register the provider with the Java Runtime System; a statical registration means to first identify the text-file `${java.home}/${lib}/${security}/${java.security}`. Then, open that file with a text-editor and extend the entries by entering the fully-qualified name of the new provider-class as seen below (to be found when entering “`jar -tf sunjce_provider.jar`” into the command-line of the terminal of the directory that contains that JAR-file.):

```
java.security
```

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.crypto.provider.SunJCE
```

The order of these entries indicates where the Java Runtime Environment will search first and where it will search next for an engine-class implementation. The international version of the SunJCE works with Java Version 1.3 and higher. The dynamical registration is demonstrated in the source-code on page 499. Names for ciphers are usually given by String-objects that follow the convention

Algorithm/Mode/Padding,

thus the full information about a popular cipher may read “DES/ECB/PKCS5Padding”, for a short introduction into modes and padding-schemes see page 484.

Using the JCE

Using the JCE — Secret key engine-classes of type `javax.crypto.KeyGenerator` produce secret keys (be reminded that classes of type `java.security.KeyFactory` together with key-specifications are intended solely to help with key-format changes!).

²²⁰A standard way is to download the file from the Sun site, for areas outside the United States or Canada, those files may be available only in a version of weakened encryption algorithms. There exists also an INSTALL-page available that helps with the installation.

²²¹Just a directory of the CLASSPATH-variable may not work!

The cryptographic engine that actually does the encryption- and decryption tasks is given by an object of class-type `javax.crypto.Cipher`²²² (an engine-class). Instances of the sub-class `javax.crypto.NullCipher` relay the data without any modification; they are useful for testing a program with the cryptographic interfaces in place but without using them.

After the engine-class `javax.crypto.Cipher` has been instantiated (by invoking the static `getInstance()`-method indicating algorithm and provider), the call of the `init(int, Key)`-method makes the `Cipher`-object ready for en-/decrypting. The integer-constant `javax.Cipher.ENCRYPT_MODE` or the integer-constant `javax.Cipher.DECRYPT_MODE` in the `init()`-method decide whether the `Cipher`-object encrypts or decrypts data. The argument of class-type `java.security.Key` takes the instance of the sub-type `javax.crypto.SecretKey`.

The name of the algorithms may include specifications like padding-schemes or streaming-modes (if a block-cipher is used for streaming data)

```
// (PREREQUISITE) Needs package javax.crypto.

class DESEncryption
{ public static void main( String unused[] ) throws Exception
{
// ----- KeyGenerator, Key and Cipher
// may take some time
    javax.crypto.KeyGenerator kg =
        javax.crypto.KeyGenerator.getInstance( "DES", "SunJCE" );
    kg.init( new java.security.SecureRandom() );
    java.security.Key key = kg.generateKey();
    javax.crypto.Cipher c =
        javax.crypto.Cipher.getInstance( "DES/CBC/PKCS5Padding", "SunJCE" );

// ----- Encryption
    c.init( javax.crypto.Cipher.ENCRYPT_MODE, key );
    byte inputData[] = "The secret message.".getBytes();
    ////////////// the secret message ///////////
    byte encryptedData[] = c.doFinal( inputData );
    byte initializationVector[] = c.getIV();
    System.out.print( "encryption finished: " +
        new String( encryptedData, 0, encryptedData.length ) );

// ----- Decryption
    javax.crypto.spec.IvParameterSpec parameters =
        new javax.crypto.spec.IvParameterSpec( initializationVector );
    c.init( javax.crypto.Cipher.DECRYPT_MODE, key, parameters );
    byte outputData[] = c.doFinal( encryptedData );
    System.out.println( "\nDecrypted: " + new String( outputData ) );
    // above byte-array to String
} }
/*
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] williammartin% javac DESEncryption.java
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] williammartin% java DESEncryption
Exception in thread "main" java.security.NoSuchProviderException: No such provider: SunJCE
    at javax.crypto.b.a([DashoPro-V1.2-120198])
    at javax.crypto.KeyGenerator.getInstance([DashoPro-V1.2-120198])
    at DESEncryption.main(DESEncryption.java:6)

->(after inserting the line "security.provider.3=com.sun.crypto.provider.SunJCE" into the text-
file "java.security":)
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] williammartin% java DESEncryption
encryption finished: 7[EÉ3ġ
                               67t
ê~oġ"U)0l
Decrypted: The secret message.
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] williammartin%
*/
```

Sources for secret-keys are instances of implementations of the engine-class `javax.crypto.KeyGenerator` (key-pairs of asymmetric encryption-algorithms use the engine-class `java.security.KeyPairGenerator`).

Infrastructure-Programming: Signing and Encrypting

Infrastructure-Programming: Signing and Encrypting — The following example will not work, because security-provider-software has to be registered with *Sun Microsystems Inc.* before being able to register provider-software within a Java Runtime System. The code below merely serves demonstration purposes

²²² A `javax.crypto.Cipher`-object encrypts data — A `java.security.Signature`-object produced the signature for a data-file. So the equivalent engine-class in the signing-process is given by the `java.security.Signature`-class!

and will not run in a Java Runtime Environment! For more on the registration process with *Sun*, see the documentation of the Java Cryptography Extension (JCE).

How to incorporate written implementations of algorithms into the Java Security Provider Infrastructure? This section shows a minimalistic implementation of a provider: First, the extensions of the classes `java.crypto.KeyGeneratorSpi` and `java.crypto.CipherSpi` furnish an implementation of the ROT-algorithm. Notice how the act of following the convention takes a large part of the entire code, in contrast to the unconventional implementation of the algorithm (already presented on page 478):

```
package ROTProviderPackage;
class ROTKey implements java.security.Key, javax.crypto.SecretKey
{ byte shift;
  public ROTKey(){ shift = 13; }
  public ROTKey( byte b ){ shift = b; }
  public String getAlgorithm(){ return "ROT"; }
  public byte[] getEncoded(){ return new byte[]{shift}; }
  public String getFormat(){ return "IEEE byte (Java)"; }
}
```

The next piece of source-code (together with the class-file) in the `ROTProviderPackage` inside the folder of the same name is:

```
package ROTProviderPackage;
public class ROTKeyGenerator extends javax.crypto.KeyGeneratorSpi
{ byte n; // number of ASCII characters to be shifted 1 to 28
  public javax.crypto.SecretKey engineGenerateKey()
  { return new ROTKey( n ); }
  public void engineInit( java.security.SecureRandom random )
  { n = 11; }
  public void engineInit( int keySize, java.security.SecureRandom random )
  { engineInit( new java.security.SecureRandom( new byte[]{1} ) ); }
  public void engineInit( java.security.spec.AlgorithmParameterSpec params,
                          java.security.SecureRandom random )
  { engineInit( new java.security.SecureRandom( new byte[]{1} ) ); }
  // public void engineInit( java.security.spec.AlgorithmParameterSpec params,
  //                          java.security.SecureRandom random,
  //                          int keySize )
  // { engineInit(new java.security.SecureRandom( new byte[]{1} ) ); }
}
```

Finally the folder named `ROTProviderPackage` contains the byte-code of the following piece of source-code:

```
package ROTProviderPackage;
public class ROTCipher extends javax.crypto.CipherSpi
{ byte shift;
  public int engineGetBlockSize(){ return 1; }
  // block-size is one byte since ASCII code and not Unicode is used
  public int engineGetOutputSize( int inputSizeOfDataInBytes )
  { return inputSizeOfDataInBytes; } // since block-size is 1byte
  public java.security.AlgorithmParameters engineGetParameters(){ return null; }
  public byte[] engineGetIV(){ return null; } // IV=Initialization Vector here not used
  public void engineSetPadding( String padding )
    throws javax.crypto.NoSuchPaddingException
  { throw
    new javax.crypto.NoSuchPaddingException( "BlockSize=1, so no padding needed" ); }
  public void engineSetMode( String mode )
    throws java.security.NoSuchAlgorithmException
  { throw new java.security.NoSuchAlgorithmException( "No modes needed in ROT" ); }
  public void engineInit( int i, java.security.Key aKey )
    throws java.security.InvalidKeyException
  { boolean encrypt = i == javax.crypto.Cipher.ENCRYPT_MODE;
    boolean definedCryptMode = encrypt | ( i == javax.crypto.Cipher.DECRYPT_MODE );
    if(aKey instanceof ROTKey & definedCryptMode )
    { byte sign;
      if( encrypt ) sign = 1; else sign = -1;
      shift = (byte)( sign * aKey.getEncoded()[0] ); }
    else throw
      new java.security.InvalidKeyException
        ("Please, ROTKey for ROTCipher or undefined cryptMode."); }
}
```

```

    public byte[] engineUpdate( byte[] input, int offset, int length )
    { return engineDoFinal( input, offset, length ); }
    public int
engineUpdate( byte input[], int inOffset, int length, byte output[], int outOffset )
    { int clear, crypt = 0;
      for( int i = 0; i < length; i++)
      { clear = input[ inOffset + i ];
        if( (clear >= 'A') && (clear <= 'Z')) crypt = ( (clear - 'A' + shift) % 26) + 'A';
        if( (clear >= 'a') && (clear <= 'z')) crypt = ( (clear - 'a' + shift) % 26) + 'a';
        output[ outOffset + i ] = (byte)crypt;
      }
      return length;
    }
    public byte[] engineDoFinal( byte input[], int offset, int length )
    { byte finalOutput[] = new byte[ length - offset ];
      engineUpdate( input, offset, length, finalOutput, 0);
      return finalOutput;
    }
    public int
engineDoFinal(byte input[],int inOffset,int length,byte finalOutput[],int outOffset)
    { return engineUpdate( input, inOffset, length, finalOutput, outOffset );
    }
    public void engineInit( int i,
                           java.security.Key aKey,
                           java.security.SecureRandom sr
                           ) throws java.security.InvalidKeyException
    { engineInit( i, aKey ); }
    public void engineInit( int i,
                           java.security.Key aKey,
                           java.security.AlgorithmParameters parameters,
                           java.security.SecureRandom sr
                           ) throws java.security.InvalidKeyException
    { engineInit( i, aKey ); }
    public void engineInit( int i,
                           java.security.Key aKey,
                           java.security.spec.AlgorithmParameterSpec spec,
                           java.security.SecureRandom sr
                           ) throws java.security.InvalidKeyException
    { engineInit( i, aKey ); }
}
/* Since no constructors have been defined,
   there exists the default constructor without arguments.
   If constructors had been defined, there should be one without arguments among them,
   otherwise a newInstance()-call from the JRE might cause trouble.
*/

/*
[localhost:JRESecurity/DataCodingSecurity/ROTPProviderPackage] andreadipietro% javac *.java
[localhost:JRESecurity/DataCodingSecurity/ROTPProviderPackage] andreadipietro%
*/

```

As an additional security-measure, suitable classes may be declared `final`, methods declared `protected` and supporting methods declared `private`.

After the code above has been compiled it is put into a folder named “demo”. This folder may be jarred, for signing, compression or easier handling purposes. Then that folder is put into the `${java.home}/lib/ext-` folder. The registration of the provider with the Java Runtime Environment is done during run-time in the following demonstration program:

```

/* (1) NEEDS package javax.crypto
*/
/* (2) NEEDS custom-written package ROTProviderPackage
   Since this source-code contains NO direct reference to identifiers
   of this supporting package,
   the classes of that package are NOT automatically compiled!
   Thus they have to be compiled previously and explicitly! SEE the file ROTCipher.java
*/
/* (3) To be fully functional the security provider had to be registered by SUN,
   since this is not the case,
   the source-code can be verified only as being compilable, the code is NOT RUNNABLE!!
*/
import ROTProviderPackage.*; //11111111 no direct reference no automatic compilation

class DemoProvider extends java.security.Provider
{ public DemoProvider()
  { super("ROTPProviderPackage", 0.0, "demonstration of a provider" ); //2222222222
    put("KeyGenerator.ROT", "ROTPProviderPackage.ROTKeyGenerator");
  }
}

```

```

        put("Cipher.ROT", "ROTProviderPackage.ROTCipher");
    /* put( engineName.algorithmName, ClassExtendsEngineNameSpi )
       --- engineName: general type of service like Signature, MessageDigest ...
       --- ClassExtendsEngineNameSpi: fully-qualified name of class
           that actually implements an algorithm for that engine/service
       --- algorithmName: name of the implemented algorithm like "DiffieHellman", "DES", ...
    */
} }

class ROTEncryption2
{ public static void main( String unused[] ) throws Exception
{
    // ----- Provider, SecretKeyFactory, Cipher
    DemoProvider theDemoProvider = new DemoProvider();
    java.security.Security.insertProviderAt( theDemoProvider, 1 );
    //java.security.Security.addProvider( theDemoProvider );
    java.security.Provider p = java.security.Security.getProvider("ROTProviderPackage");
    p.list(System.out);

    // To list all providers use the code of ListAvailableEngines.java
    System.out.println( "getting KeyGenerator ..." );
    javax.crypto.KeyGenerator kg =
        javax.crypto.KeyGenerator.getInstance("ROT", "ROTProviderPackage");
    javax.crypto.SecretKey key = kg.generateKey();
    System.out.println( "getting Cipher ..." );
    javax.crypto.Cipher c = javax.crypto.Cipher.getInstance("ROT");
    // Encryption
    c.init( javax.crypto.Cipher.ENCRYPT_MODE, key );
    byte inputData[] = "THIS IS TO BE KEPT SECRET".getBytes();
        /////// the secret message /////
    byte encryptedData[] = c.doFinal( inputData );
    System.out.print( "encryption finished: " +
        new String( encryptedData, 0, encryptedData.length ) );

    // Decryption
    c.init( javax.crypto.Cipher.DECRYPT_MODE, key );
    byte outputData[] = c.doFinal( encryptedData );
    System.out.println("Decrypted: " + new String( outputData ) );
} }

class ROTParameterSpec implements java.security.spec.KeySpec
{ byte shift;
    public ROTParameterSpec( byte shift ){ this.shift = shift; }
    public byte getShift(){ return shift; }
}
/*
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% java ROTEncryption2
...
Exception in thread "main" java.security.NoSuchAlgorithmException: Algorithm ROT not available
    at javax.crypto.b.a([DashoPro-V1.2-120198])
    at javax.crypto.KeyGenerator.getInstance([DashoPro-V1.2-120198])
    at ROTEncryption2.main(ROTEncryption2.java:15)
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro%
// reason improper package placement
// (no explicit reference to package member makes no automatic compilation,
// here the package has to compiled independently)

Exception in thread "main" java.security.NoSuchProviderException: JCE cannot
authenticate the provider ROTProviderPackage java.lang.SecurityException: Cannot authenticate
jar:file:/Users/andreadipietro/JavaCodeFiles/JRESecurity/DataCodingSecurity
/!java.security.PrivilegedActionException <<java.util.zip.ZipException: error in opening zip file>>
    at javax.crypto.b.c([DashoPro-V1.2-120198])
    at javax.crypto.b.a([DashoPro-V1.2-120198])
    at javax.crypto.KeyGenerator.getInstance([DashoPro-V1.2-120198])
    at ROTEncryption2.main(ROTEncryption2.java:30)
[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro%
// reason above

[localhost:~/JavaCodeFiles/JRESecurity/DataCodingSecurity] andreadipietro% javac ROTEncryption2.java
*/

```

Within the Java Provider-Infrastructure, this kind of registration is also called dynamical because it is done during run-time. A registration by adding a line (`security.provider.n=...`) in the `java.security`-file is also called statical- or persistent registration.

Concept: Network-Security

Concept: Network-Security — Java furnishes packages that make it possible to use the concept of a Secure Socket Layer (SSL). Secure sockets encrypt the content of outgoing IP-packets and decrypt the content

of incoming IP-packets. Another possibility means working with the HTTPS-protocol. This, as well as secure streams has not been covered by this text.

Summary: Security

Summary: Security — Quality of Service and Security are rather pervasive computing-concepts: These concepts include the choice of hardware, maintaining it properly, securing it from unauthorized access; as well as configuring the operating-system, especially with respect to input- and output-routines to an external network. Finally, a runtime-system like the one furnished by the Java Development Kit may present security-infrastructures of their own. These various levels of consideration come together with such diverse kinds of interests of people like system-programmers, Java infrastructure-programmers, application-programmers who want to use security-infrastructures without an extreme overhead of learning and finally end-users. This mix endows issues in the area Quality of Service and Security with considerable complexity.

Even on the level of the Java Runtime Environment (JRE), security configuration relies heavily on text- and other files (`java.security-file`, `java.policy-file`, `jks-keystore-files`) that have to be safe-guarded on the level of the operating-system to assure an effective overall security.

Java relates closely security-relevant issues like class-loading, access-controlling and cryptography-related areas, authentication and encryption of data:

If properly programmed, Java's secure class-loaders supply detailed information about the source of the code (`CodeSource`-objects). This can be used to assess whether to grant permissions to system's resources.

Protecting resources, like files and network-connections within the Java Runtime System (JRS) is possible by using the access-control-infrastructure in conjunction with secure class-loaders. A given policy, that collects all permissions given to groups or individual code-sources, can be analyzed by a secure class-loader to supplement the loaded class with a `CodeSource`-object and an object of type `ProtectionDomain`, that gathers a reference to the `CodeSource`-object and all the permissions given by the policy concerning the loaded class and its derived objects. Users (`policytool`) and application-programmers cast their very own perspective on this access-control-infrastructure, as has been demonstrated. Programmers who want to refine the access-control mechanism by furnishing own types of `Permission`-classes and scrutinize the programming of secure class-loaders with the production of own objects of type `ProtectionDomain` and `CodeSource`, add a new level of complexity to Java programming. This kind of infrastructure-programming usually is unseen and rightfully ignored by users and application-programmers who usually focus on solving real-life-problems.

The other area with an even richer spectrum of issues represents that of encryption: There can be considered the area of mathematics which solely considers the design and testing of cryptographic algorithms. This area has been omitted almost completely in this text, because this area alone can fill entire handbooks. Some basic concepts of encryption have been introduced: The secret-key-method that uses a single (secret-)key, the asymmetric-key-method that uses a key-pair of which one remains private and the other key is made publicly available. With these methods has been attempted to authenticate messages and to send encrypted messages which can only be read by the intended receiver. These problems also made Java supply an entire Provider-Infrastructure. On top of that infrastructure the Java Development Kit itself furnishes tools like `jar`, `jarsigner`, `keytool` for the interested end-user. The Java application-programmer may install provider-packages and use the readily implemented cryptographic algorithms. Even here, an interested Java programmer may furnish own provider-packages which requires much more knowledge about the Provider-Infrastructure and the algorithms to be implemented.

For more information refer to the Java ➡  ➡  Java Platform API Specification [JPAPIS] of the package `java.security`.

Keep in mind that the access-control mechanism of the Java runtime-system can only be as good as the security features of the underlying computer-platform, especially the operating-system. Software security deficiencies of the operating-system (free access to policy-files ...) may render the Java security-mechanisms obsolete.

The Java 2 Enterprise Edition (J2EE) uses further mechanisms, like log-in procedures or user-, super-user- and administrator-functions to refine a system's security infrastructure.

Part VII.

Single User Interface

Single User Interface — So far, interaction between an end-user and a running Java program has been confined to the use of the input-method `java.lang.System.in.readln()` (example on page 297) and the constantly reoccurring output-method `java.lang.System.out.println()`. The first method takes a keyboard-input from the top-leveled terminal-window; the second method prints strings into the terminal-window. Entering string-arrays, when issuing the “java”-command in a terminal-window of the operating-system, was another means of keyboard input, but only at the time when starting an application (page 236).

How does an (end-)user interact with a computer? This question is posed far too general to be adequately covered in this place. Nevertheless this gives the direction of thought in this part of the text, focusing on Java-related use of mouse, keyboard and monitor combined to so-called Graphical User Interfaces.

Concepts of end-user interaction have to be complemented by actually working implementations. How do such implementations fit into the computer hardware, the operating-system, a possible runtime-environment (for example the Java Runtime Environment, JRE) and the application-program?

These questions concern the basic Java Runtime mechanisms and the pre-written packages to be introduced. Furthermore the range of notions presented here, represents only a small part of the services supplied by the Java packages `java.awt` and `javax.swing`. Thus, in the limited space of this text, only central ideas are presented and exemplified by Java programs, as far as possible.

Java Foundation Classes (JFC)

AWT (Abstract Window Toolkit)	<code>java.awt</code>

Swing	<code>javax.swing</code>

Accessibility	<code>javax.accessibility</code>

Java2D	<code>java.awt.image</code>

Drag and Drop	<code>java.dnd</code>

Java Advanced Imaging (JAI)	<code>javax.media.jai</code>

The collection of packages like `java.awt`, `javax.swing` and `java.dnd` (Drag and Drop) is called the Java Foundation Classes (JFC). Of this collection only the AWT- and Swing-packages are presented in excerpts. Java2D (mostly found in the package `java.awt.image`) and Java3D give extensions of the standard toolbox (`java.awt.Graphics`) for graphics-objects, which is found in the AWT-package (AWT is the acronym for Abstract Window Toolkit). The `javax.accessibility`-package extends support for input- and output-devices. There has been omitted also any deeper consideration on (hyper-)text production (awt and swing), image-processing (jai) and their management as well as Drag-and-Drop-functionalities.

This final part of the text first introduces the, probably well-known concepts of a Graphical User Interface (GUI). Then are presented Java’s basic capabilities for putting pixel-graphics on the computer-screen, for drawing individual graphics-objects (like circles and rectangles) on the screen and finally are investigated on-screen representations of so-called Java screen-components.

The AWT-implementation of the delegation event-model is introduced, which allows a Java programmer to let an end-user interact with the Java program by using keyboard- and mouse-signalizations. At the end of this part, the proper synchronization of those signalizations together with the on-screen reactions is investigated by considering the so-called event-dispatch-thread. Then, there is given an example of a Java-desktop and finally applets are introduced as special screen-components.

None of these topics is covered exhaustively and mastering even one of those areas should require some additional experience and insight!

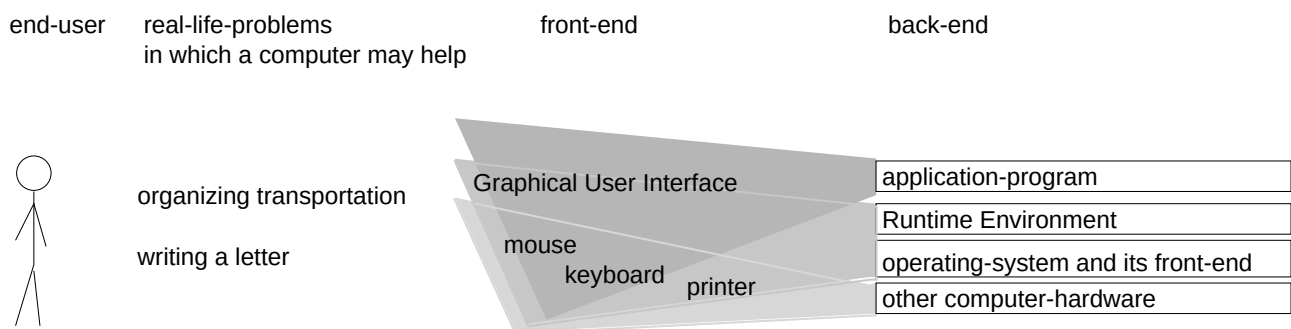
Concept: User Interfaces

Concept: User Interfaces — Users who want to go to another place do walk, take an automobile, bus, train, ship or airplane. These means of transportation help the end-user in solving the end-users real-life-problem of needing to be somewhere else. But these means or tools for solving the user's real-life-problem produce secondary, tool-related problems: Getting a raincoat, knowing how to drive a car, delving into time-tables, arranging proper connections.

In short: An end-user has a real-life-problem (transportation) and solves it with tools (leg, car, bus, train ...) ensuing tool-related problems (time-tables, delays, congestions ...). If the end-user can choose between different tools, then the user most likely will select that specific tool which promises minimal tool-related problems and a high problem-solving potential.

Computers, operating-systems (with their various front-ends: terminal-based, Graphical User Interface, mixtures of both) and especially application-programs (short: “applications”) are such tools for an end-user who hopes to be helped by them in solving kinds of real-life-problems. Here tool-related problems should be kept to a minimum too. In this general context end-user interfaces of computers are also called **front-ends**.

front-end



Ideally, the user's daily contact with a computer-system will rather be confined to standard hardware functions and to the user-interface of the application-program that intends to help in solving real-life-problems. — Non-ideally a user may be absorbed by maintaining or configuring the computer in the back-end area (hardware, operating-system, Runtime Environment, application-program).

Hardware User Interface

Hardware User Interface — The typical minimal configuration of a hardware-User-Interface is given by a SCREEN-KEYBOARD-COMBINATION. This is the case where only characters of a single character-set appear on the screen in fixed lines and end-user inputs can only come from the keyboard. This hardware-and software-setting is called a **terminal- or text-based user interface**. To make the computer do anything, usually a textual-command (taken from a command-handbook) has to be entered via the keyboard and committed with the keyboard's return-key. (see page 45)²²³

In the 1990ies standard I/O-hardware was given by a SCREEN-KEYBOARD-MOUSE-COMBINATION. On-screen appear graphical-objects like OK- or Cancel-buttons, checkboxes, selection-lists and text-fields. The mouse is used primarily for command-purposes²²⁴: Pressing buttons by positioning the mouse (-

²²³ In the year 1982 Apple Pascal managed to furnish a menubar on a text-based screen layout, by reserving the upmost line of the screen as a list of commands (e(dit f(ile ...)); entering the first letter activated the command; there the terminal- or text-based hardware configuration already had been given features of a modern Graphical User Interface. So here, the classifications “terminal-based”- or “graphical user-interface” are not meant to rigidly separate notions for definition, but rather to indicate the range of possibilities!

²²⁴ Some versions of the Windows operating-system strived to make the following possible: Each command that could be given with the mouse was associated with a keyboard-command (for example CTRL-X for “cut”, CTRL-C for “copy”, CTRL-V for “paste”) to make the software effectively independent of the usage of a mouse.

pointer) over the `Button`-object and pressing a mouse-key, or by selecting an option by clicking on a checkbox. The keyboard is used primarily for entering data in text-fields. (see again page 45)

Mouse and keyboard can be loosely classified according to which kind of task they are used:

The basic idea of using mouse and keyboard can be re-formulated as follows:

Mouse Interaction codes commands (=meta-data)

Keyboard input enters (textual) data

This basic idea is often extended pragmatically by also using keyboard shortcuts (`CTRL-C`, as an alternative to navigating the mouse-pointer through the menu: `Edit-Copy`). Concepts or ideas should give a clear starting-point for thinking and working, good ideas should be able to tolerate or to adapt to sensible contradictions.

Such a kind of user interface in which graphical-objects on the screen become relevant or even dominant over text-representation is called **Graphical User Interface (GUI)**. Re-referring to the graphic on page 45 for the third time, illustrates also that some operating-systems with Graphical User Interfaces, just by making text-based terminal-windows available, allow both kinds of user interfacing.

Revising the previous paragraphs, ascertains that end-user interaction requires hardware to closely cooperate with software that implements a Graphical User Interface.

There exists a wide VARIETY OF AUXILIARY HARDWARE: Printer, scanner, loudspeaker, microphone, trackball, trackpad (on laptop computers), pen-table, card reader, braille output, joystick, web-camera, cameras (Data-gloves or cameras that detect eye-movement are less typical but equally versatile). These means of end-user interaction are only mentioned here as an illustration of what may be still possible.

Graphical User Interfaces (of Operating-Systems)

Graphical User Interfaces (of Operating-Systems) — An operating-system is a program. This program relieves application-programmers and end-users from considering technical details of the computer hardware. The operating-system furnishes services like internal memory-management, file-storage, file-search, processing keyboard- and mouse-input. In that way, the operating-system basically lets application-programmers and users concentrate on solving their real-life-related problems. — When the computer is turned on, virtually the first program that becomes active is the operating-system. The operating-system cannot foresee all variants of new hardware that may be attached to the computer; therefore new hardware becomes functional by integrating associated software-patches into the operating-system. These hardware-related software-patches are also called **drivers**. Operating-systems may also be imaginable as a managed collection of driver-software. (Remember the sections about operating-systems and file-systems at the beginning of the text on page 43.)

Below has been compiled a table giving an impression how different operating-systems furnish differing user interfaces:

operating-system	graphical front-end	terminal-windows
BeOS	integrated	yes
Linux	KDE, Gnome	integrated
MacOS 9.x	integrated	none
MacOS X	integrated	yes
DOS	Windows 95, 98	yes
Windows ME	integrated	none

Stressing the Graphical User Interfaces in the following chapters, the next sections try to give an answer to the question “What are the common functionalities of most operating-systems with graphical front-ends?”

Graphical User Interface of Operating-Systems: Desktop - Icons - Windows

Graphical User Interface of Operating-Systems: Desktop - Icons - Windows — Operating-systems with a Graphical User Interface usually open a session by presenting a screen containing icons. **Icons** are little images used for representing the function associated with them: Text documents are represented by icons that look like sheets of papers. The program for editing such documents may be characterized by an icon that looks like a pencil writing on a sheet of paper. Icons of a printer-software may look like a little

printer. (Icons have been visualized in the graphic on page 45.) The screen background of a Graphical User Interface (GUI), which also can be seen as a container of icons, is also called **desktop**.

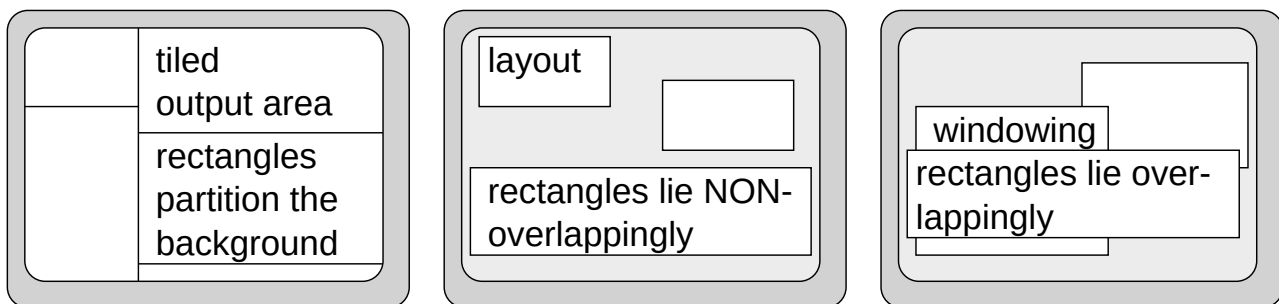
MOVING THE MOUSE-POINTER ONTO AN ICON AND POSSIBLY (DOUBLE-) CLICKING A MOUSE-KEY lets the icon open an (associated) rectangled area on the screen, a **window**. Inside such a window may appear the editable text of a document-file or the dashboard-like control-panel of some other program, for example that of a printer-controller. Such windows (sometimes also called **frames**²²⁵), or rectangles in general, can be arranged in different ways on the screen:

tiling non-overlapping and covering the plane²²⁶

layout non-overlapping

windowing overlapping

Output Area: Screen



Most operating-systems favor the windowing approach of overlapping layout. A window that **TAKES THE CURRENT INPUT FROM THE KEYBOARD OR THE MOUSE** is said to be an **activated window**. A window that is activated is also called the **top-level window** and can be easily discerned from all other currently opened windows, usually by a contrasted border. A window may be activated by moving the mouse-pointer onto it and possibly clicking once with a mouse-key.

Inside an activated window **graphical-objects or screen-components**²²⁷ (like text-fields or buttons) **on the screen are said to be on focus** if they are ready to be manipulated (to take input or to take a command). The mouse-pointer may be positioned over a text-field of the window of a document-file; on clicking of the mouse-key, a cursor may become visible and the text-field is on focus, ready to take input. A programmer can *disable* screen-components like windows or text-fields, then the window cannot be activated and the text-field may not come into focus, even if the end-user does the, usually required, mouse-clicking.

Thus windows on the screen and graphical-objects²²⁸ within windows basically can appear in three different states:

on focus/activated Screen-components or graphical objects in the state of being activated or on focus can take input, they can be operated with. (Consider a window which contains a set of screen-components; often, by multiply pressing the TAB-key of the keyboard, the end-user can rotate the focus among those screen-components in a pre-programmed cycle (focus-cycle).) Depending on the operating-system, the notions focus and activation may have slightly different meanings.

not on focus/ not activated The screen-object *can be made* to take input or *can be activated* by positioning the mouse-pointer over it and, if necessary, by additionally clicking a mouse-key. Some

²²⁵ A frame in the context of a Graphical User Interface may be confused with a packet or frame in packet-oriented network-communication. Frames in communication are the parts of a data-packet that contain data, sender- and destination-address and other management-information (page 349). Frames in the context of Graphical User Interfaces (GUIs) are screen-components which appear as graphical-objects on the screen, usually as rectangles containing information (icons or text-fields or the like), more on page 510 or beginning with page 540.

²²⁶ Due to a smaller resource-consuming overhead a tiled screen was used by the operating-system called Oberon. (<http://www.oberon.ethz.ch> or <http://www.ssw.uni-linz.ac.at/Oberon.html>) Literature: [Oberon] Niklaus Wirth, Jürg Gutknecht; Project Oberon; 1992; ACM Press and Addison-Wesley, ISBN 0-201-54428-8

²²⁷ The difference of graphical-objects and screen-components is described in the next section and on page 540. Here a graphical-object describes any application-specific graphic like a letter printed out in the window of a text-editor or a circle or an oval in the window of a graphics-editor.

²²⁸ For example: A line, circle, rectangle in a graphics editor. A text-passages high-lighted for further use.

operating-systems allow the end-user to choose the desired configuration: For example, the mouse-pointer in the area of the screen-component, makes that screen-component take input, though another window is the top-level window at the same time. Thus again, depending on the operating-system, the exact meaning of the notions “focus” and “activation” may differ.

disabled The window or the graphical-object-within-the-window is barred from taking input, primarily programmer-induced and not user-induced

Other Java-related states are listed on page 541 (Display) and on page 598 (visible/showing).

Graphical User Interface of Operating-Systems: Screen-Components

Graphical User Interface of Operating-Systems: Screen-Components — A window, an icon, an OK-button, a simple image, are graphical-objects on the computer-screen that may change their appearance: A window may change its size or its content. An icon may be repositioned and change its color upon being activated or deactivated. OK-buttons usually change their color when the mouse-pointer has been moved onto them and the mouse-key is pressed. A simple image serves just as an illustration and any user-operation on it may be disabled.

Such objects, that may change their appearance being represented by differing graphical-objects, but remain the same entity are called **screen-components**. A checkbox is also a typical example of a screen-component: Initially the checkbox is represented by an image on the screen, after positioning the mouse-pointer on it and clicking a mouse-key, the checkbox is represented by another image on-screen, signaling the choice that the user made — here two images (depicting the checkbox checked and unchecked) are part of one screen-component (the checkbox)! For more on the concept of a screen-component, see page 540.

screen-
component

OS-GUI: File-System: Opening Files and Folders

OS-GUI: File-System: Opening Files and Folders — Most of the direct contact between an end-user and an operating-system (OS) happens when storing and retrieving files from some persistent storage-device like a hard-disk. Recall the section about file-systems on page 46. In Graphical User Interfaces (GUIs) of operating-systems a single mouse-click (or depending on the operating-system, a double mouse-click) on that folder-like icon opens a window in which usually appear other *folder*-icons and possibly some *file*-icons. Naming folders and files in that folder-containment-hierarchy makes it possible to identify a file by a “path-name” like
 /usr/home/MarcAurel/writings/TheMeditations/BookOne.html (Unix/Linux/Solaris see page 47).
 (The text may be downloaded from <http://classics.mit.edu/Antoninus/meditations.html>)

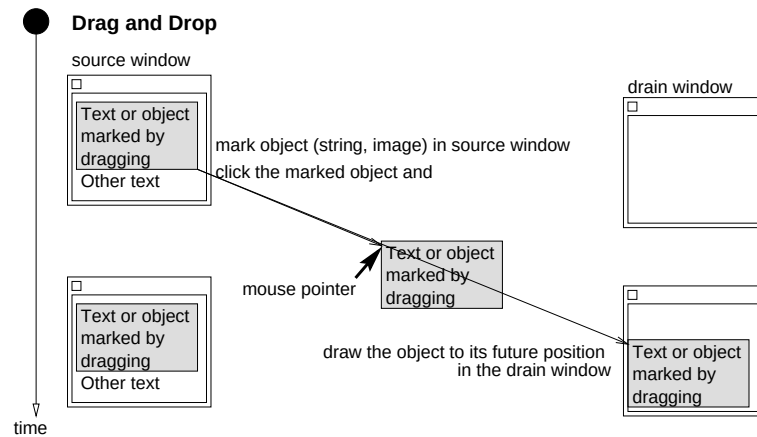
GUI: Giving Commands: Clicking - Drag and Drop (D&D, dnd)

GUI: Giving Commands: Clicking - Drag and Drop (D&D, dnd) — Simple commands in operating-systems (OS) with a Graphical User Interface (GUI) are given by mouse actions: Positioning the mouse-pointer onto one screen-component (for example an icon) on the desktop and pressing a mouse-key for a moment is also called **clicking the screen-component** (icon). Two clicks, in fast succession, represent a so-called **double-click**. Often a single click brings the screen-component into focus and a double-click opens a window associated with the screen-component. Positioning the mouse-pointer onto one screen-component, pressing a mouse-key and holding it down while moving the mouse-pointer, is called **dragging**, because this procedure usually enables the user to move screen-components over the desktop.

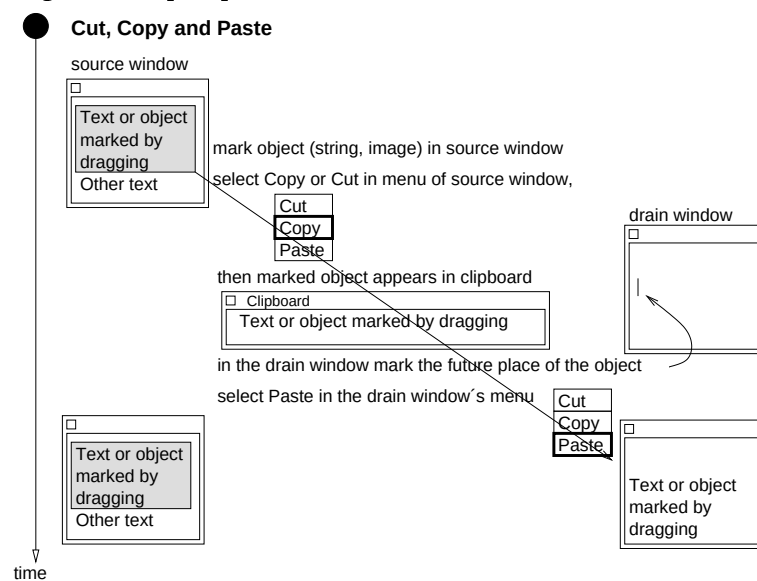
clicking

dragging

Moving the mouse-pointer onto an icon, holding a mouse-key down and moving the mouse into a desired window and letting the mouse-key go makes it possible to move the icon or rather the file or the entire folder with its contents into the destination-window. Such a dragging procedure is also abbreviated as a **drag and drop action**. If an application supports drag and drop functionalities, then dragging may be done within and between components of windows that represent a running application (marked text-passages in a text-editor or marked images in a graphics-application).



Applications that do not support a drag and drop infrastructure have to use the menu items “Cut”, “Copy” and “Paste” and a so-called “clipboard” as an intermediate storage-facility. Java supports Drag-and-Drop functionalities but this introductory text does not describe document-production with Java, neither does it describe the Java Drag-and-Drop implementation.



GUI: Giving Commands: MenuBars - PopupMenus - Toolbars

GUI: Giving Commands: MenuBars - PopupMenus - Toolbars — Commands in operating-systems (OS) with a Graphical User Interface (GUI) that cannot be coded as simple mouse actions have to be given by other means. Commands, which cannot be coded by mouse-clicking of icons on the desktop, can be activated by searching them in so-called **menus** that may appear also on the desktop (=screen). The initial part of menus to be seen on-screen is just a list of descriptive words (this initial list usually is called **menubar**). For an example of an opened menu, see the screen-copy on page 547. Menus may appear at the sides of the desktop or at the sides of a window; so-called **popup-menus** may appear anywhere if the user presses down a mouse-key while holding down a keyboard-key like the Ctrl-key. (see also the small example of an “exit-popup-menu” in the graphic following the source-code on page 610)

menu =
menubar +
item-list

A MENU TOGETHER WITH ITS SUB-MENUS REPRESENTS AN HIERARCHICALLY ORDERED COLLECTION OF COMMANDS. Positioning the mouse-pointer onto one of the items of a menubar — and in some operating-systems pressing the mouse-key permanently or once — opens a second list of sub-items from which can be chosen a command or option by positioning the mouse-pointer over one's choice and releasing the mouse-key. This means that choosing a command often is done by dragging the mouse over the menu and possible sub-menus.

A menu given by a simple lists of commands — that means without sub-menus — often is called **toolbar**. Toolbars often use graphical, iconlike representations of the commands. A menubar (File Edit View ...) from which can be chosen several menus, alongside with a toolbar which represents commands as iconlike symbols (arrows, cross, ...), is shown in the browser-window on page 370.

GUI: Windows as Containers of Screen-Components

GUI: Windows as Containers of Screen-Components — As already indicated, and as it may be already known from practical experience, a window (also called frame) may not only contain icons of folders and files. Some types of windows can be used as containers for screen-components like OK- or Cancel-buttons, checkboxes, selection-lists and text-fields. Exactly those windows are used as single user interfaces for the actual application.

Applications may be started by clicking the application-icon itself or one of its data-file-icons. If, for example, the application is an Internet-browser and the computer is an Apple computer, then simply double-click the icon which represents an HTML-document to view the document in the browser-window. Or double-click the browser icon and select the file to open. The software for helping to solve real-life-problems, the so-called application-software, usually blends into this background furnished by the operating-system and its Graphical User Interface (the so-called system-software).

Graphical User Interface of Application-Software

application

Graphical User Interface of Application-Software — **Applications** are the programs that help end-users in solving their real-life-problems. Examples are: Text-Editors (problem: Writing a letter without constantly rewriting it when minor editing takes place like on a sheet of paper), FAX-programs (problem: sending symbolic (text) or non-symbolic information (pictures) over a telephone-connection used for voice-communication), Internet-browsers (problem: getting information), encryption and signature-management software (problem: hiding information from a third person or authentication) and even — to a certain extent — the file-system of an operating-system (problem: storing and finding files — electronically). A bookkeeping-program or properly configured spreadsheet relieves the accountant from tediously adding numbers. Non-application programs like much of the operating-system, driver/filter software, networking-protocol-stacks and to a certain extent de-/compression programs support the applications but usually are not immediately noticed by the user as related to her or his real-life-problem. (Remember also the first introduction of the notion “application” on page 38.)

HOW TO CHOOSE AN APPLICATION?

- (REAL-LIFE-PROBLEM and SOLUTION STRATEGIES) First the end-user, or the person who lets that program install, should have a thorough understanding of the problem. What do I want to do, sending FAXes or emails, writing a report or a book, computing a balance for a small firm or a huge firm?
- (REAL-LIFE-PROBLEM-DOMAIN MATCHES APPLICATION-DOMAIN) **So, when a real-life--problem arises, a properly chosen standard application should help in solving it faster and more effectively.** Ideally, the end-user or the person who lets that program install, have a thorough understanding of the problem and WHY the specific application was chosen to solve it!
- (APPLICATION-RELATED PROBLEMS TO BE MINIMAL) “How can I make the application do what it is programmed to do?” Many applications, for a beginner even a simple text-editor, may force the user into reading a large handbook that documents the application’s functionalities. Even the user who has a thorough knowledge of the problem-domain may be (unnecessarily) forced into the intricacies of the handling procedures of that specific text-editor. It benefits the end-users if that **overhead of learning to use the application can be kept as small as possible.**

User-interface-programmers and day-to-day users face the problems arising from the last item. User-interface-programmers are in the position to control two services the Application User Interface (AUI) can furnish:

(**TOOL-TUTORING**) A good program lets the end-user concentrate on the problem which the end-user wants to solve, not on learning how to use the program.

Even for advanced text-editors it proves difficult to credit the various aspects and levels of knowledge of end-users in text-formulation and -design. Nevertheless for those end-users, who have a moderate knowledge of application-independent functions (Cut-Copy-Paste), there are intuitively usable text-editors that enable users without much tutoring to effectively produce a text.

(**PROBLEM-TUTORING**) Ideally the program could tutor the user in the given problem-domain (commented spell- and grammar checking, giving templates for letters and reports, giving hints and proposing solutions, even on-the-fly explanations of what the program is capable), but a really effective problem-oriented tutoring on the side of applications is scarcely found.

Design of Graphical User Interfaces (GUIs)

Design of Graphical User Interfaces (GUIs) — In the introduction to this chapter a (Graphical) User Interface was presented as a means of stressing the real-life-problem-orientation of the application to the end-user. On the basis that the problem-domain is well-comprehended by the user, **intuitive access to a user interface** means easy use of the application, ideally untutored by other means. Therefore, programmers of efficient user-interfaces usually try to incorporate as much backgrounds of potential users as possible:

sensory Small print on a computer-screen strains most people's eyes. Unorganized positioning of screen-components within some window may promote confusion.

knowledge A professional typographer may be more aware of the potential or the deficiencies of a text-editor application than a high school student.

culture Many Arab-people read and write from right to left and practice a right-to-left (←) watching-habit. Furthermore, remember the remarks on page 259.

On the programmer's side the **overall style of the design** often gives a good impression of the user-friendliness: Being used to analyze problems clearly, respecting options, granting flexibility without losing the strategic line, helps a lot when designing a user-interface. Often even users with little background-knowledge can perceive the results of such an approach.

For a closer look on general principles search for "Human Interface Guidelines" (HIG) on the Internet-sites of the producers of operating-systems, respectively their front-ends (www.apple.com, www.sun.com, www.ibm.com, www.kde.org, www.gnome.org). (for a list of operating-systems search in Yahoo)

The overall style of an human interface (colors, shape and general design of screen-components, even a consistent background of audio signals) is also called "Look and Feel" of the interface, operating-system or/and application. Abbreviations often found for the term "Look and Feel" are LAF, LNF. Java's capabilities to integrate "Look and Feel"-choices into the programming of Graphical User Interfaces (GUIs) are not covered by this text.

This text concentrates on (Java-related) programming-libraries (Application Programming Interfaces (API)) that contain pre-programmed elements for designing a User-Interface (UI). The text does not cover explicitly necessary influences on end-user interface-design that come from areas like psychology, culture, and real-life-problem-structuring. In that sense the text solely covers the basics of the programming mechanics of an end-user interface! UI = User Interface

Concept: Programmer's View of User-Interface (UI)

Concept: Programmer's View of User-Interface (UI) — Various ways of structuring source-code have been introduced so far: Procedural programming with methods. Type-extension with object-orientation, concurrency with multithreading or even component-programming (Java-Beans). More related to the top-down approach of modularizing, end-user-related program-code may be structured according to the Model View Controller (MVC) concept:

Program-Structure: Model View Controller (MVC)

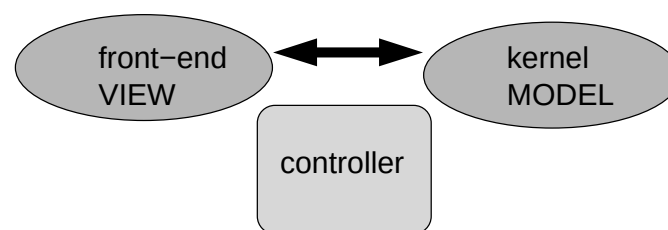
Program-Structure: Model View Controller (MVC) — The following concept intends to ease programming by giving the programmer a guideline of how to structure the source-code: Often the source-code of a **larger application with a (Graphical) User Interface** can be divided into several parts with basically different functions:

Program-parts called front-ends concern themselves with all what an end-user sees of the actual application. As previously stated, this **FRONT-END**, often a **GRAPHICAL USER INTERFACE** or **VIEW**, is designed for simplicity and intuitiveness of use. Usually the **VIEW** or front-end manages all end-user-related input/output activities.

The application does its problem-related work in the part of the program called **KERNEL**. The kernel of the program handles a data-**MODEL** of the real-life-problem, which the application intends to ease solving.

The communication between the front-end (**VIEW**) and the kernel (**MODEL** being the solution-helper for the real-life-problem) often is organized by the front-end. But especially in applications with a larger amount of source-code, the communication between front-end and kernel may be organized by a separate entity in the source-code: the **controller**.

Structuring source-code of an application
that uses a Graphical User Interface (GUI)



The notions front-end, kernel, model, view and controller may differ somewhat in other programming perspectives, but their essence has been described above.

So, PARTS OF SOURCE-CODE OR OBJECTS exhibit Model, View and Controller behavior under the following circumstances:

Model-Code contains the core of the application, the part that does the actual data-processing. This part of the source-code contains the knowledge and expertise of the programmer concerning the real-life-problem which the application intends to help solving. This part of the application even may reside on a different platform (for example in the case of a database-access). The model-part of the code produces the results for solving a real-life-problem. The model-part does data-storage and data-manipulation.

View-Code manages all user-related input and output. This code solely services the user and primarily attempts to keep the end-user's interface as intuitive as possible. Here the kernel's results are

presented to an end-user ideally in a way that lets the user concentrate on the real-life-problem to be solved without minding the computer or the application itself. This part of the source-code constructs the end-user-interface and the input/output routines.

The problem-centered approach of the MODEL-part and the end-user centered approach of the VIEW-part of the program sometimes can be helped in communicating: The communication-related parts may be treated separately (just like protocols between message-senders and -receivers are developed for a clear division of concepts, see page 274):

Controller-Code manages the communication between MODEL- and VIEW-parts of the program, the kernel and the front-end. Controller code, for example, may reformat data when exchanged and makes it possible that the two other parts of the programming task – model and view – can be handled as independently as possible. This part of the source-code connects and transforms the input/output properly between the VIEW- and MODEL-parts. The controller realizes the input/output-infrastructure.

If kernel- or front-end parts of the code contain reusable objects or elements, then the controller-part often covers the part of the code that contains the least reusable parts of the entire code.

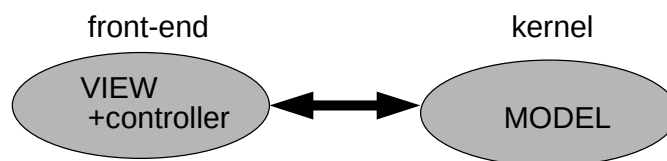
Example: (Image- or Text-Editors) End-users can view and edit images and texts on the computer-screen. The representation of images and texts within the source-code of the image- and text-editors differs significantly from what can be seen on-screen. The parts of the source-code that manage the internal representations of images (bitmaps or graphical-objects) and texts (tree-structures) are the MODEL-parts. The parts of the source-code containing the routines for transforming the internal representations into bitmaps which can be displayed on the current screen(-hardware) constitute the VIEW-part of the source-code. Finally, the parts of the source-code that make these two parts interact according to the end-users mouse-clicking and menu-commands are called CONTROLLER-part of the image- or text-editor's source-code.

Example: (Scrollbar/Slider) On a concrete virtually microscopic level consider a scrollbar or slider screen-component, as depicted in the graphic on page 547. An end-user using the mouse may drag the rider of that component through a range of values, typically 0, 1, ..., 99. The screen representations of that screen-component, the images used to give the end-user the impression of a slider and their drawing routines, together with the mouse- and keyboard-listening routines are part of the VIEW-section of the source-code. The variable that stores the value of the slider constitutes the MODEL-part of the source-code. Finally there may be found a part of the code that uses the information of the listening routines to change the value of the model-variable, and takes that value to the drawing routines of the VIEW-part for updating the screen-image of the rider. This part of the code is called the CONTROLLER.

The Model View Controller (kernel/front-end) concept may be used to structure the source-code of an application with a (Graphical) User Interface. But often, especially with smaller pieces of source-code, this procedure may introduce unnecessary complications just like the object-orientational programming may be unsuited for smaller programs. Therefore the concept should be seen as a guideline and suggestion; ideally the programmer alone has to decide whether there lies an advantage in applying these principles!

The Model View Controller (MVC) concept often is applied simply in the front-end/kernel version, where the front-end takes over the controller functions. In these cases the front-end part is also called **delegate** or **UI-object** (UI for User Interface).

delegate,
UI-object



Examples: (**operating-system, OS**) Most operating-systems with a Graphical User Interface consist of a kernel that handles hardware-access (communication with drivers, Random Access Memory organization and more) and processor-management (multi-threading, scheduling) and a front-end for the user that furnishes icons of files and folders as well as mouse-pointers, dialog-windows and menubars.

Example: (**Swing**) In Swing (a Java package for screen-components), the Swing-components use a user interface/delegate approach: Each JButtonModel-component (model) is associated with a JButtonUI component (delegate). Both are accessed by a JButton-class. This intricacy becomes only of concern to those who want to change the VIEW of the button. (These programming tasks are not covered in this text.)

What relation has the Model View Controller concept to the source-event-drain concept? The latter concept introduces one-way-communication. The MVC-concept needs two-way-communication. The MVC-concept may use source-event-drain structures for internal communication. And the VIEW-part of the MVC-concept clearly is highly specialized for managing the interaction with an end-user.

What relation has the Model View Controller concept to the client-server- or peer-to-peer-communication? The latter concepts solely consider who initiates a two-way-communication. The programmer should be free, in using the MVC-concept, to implement other concepts if they serve the cause.

This concludes the present introduction of overall concepts in programming single-user-interfaces. At this point it is possible to skip the following sections with their image-considerations and to continue with the sections on screen-components (start on page 540).

Screen Output

Screen Output — The chapter on screen-output introduces three types of graphic-related notions: Images, graphical objects and screen-components. Furthermore, this part of the text also presents prototypes of Java programs which anticipate simple event-driven functions. The event-communication is more systematically approached in the following chapter about event-communication (beginning with page 572).

(Text and Images in Communication) First, there may be formulated one fundamental observation: Humans recognize image-information fast (like that of music and odor). One reason may be that images (like music and odors) usually are perceived in a highly non-symbolic approach, sparing some mental

decoding process. But even pictograms  leave much room for context-dependent, cultural or even individual freedom of interpretation. Contrary to that, using text or speech gives a less direct means of communication. Text or speech usually leaves less freedom for interpretation, and therefore can be used in a more precise way. Text can be interpreted by using a relatively clear set of rules, beginning with a dictionary or a grammar-book.

(Bitmaps) A first attempt in representing images as sequences of bytes is given by so-called **bitmaps**. In this case images are represented as a grid of image-points, so-called pixels, short for picture-element. A pixel holds the grayness- or color- and transparency-information of the image-point it represents. If a picture-element is (partially) transparent, then the background of the image can be seen, which allows rectangled images without a visible border to contain and exhibit shapes differing from that of an

rectangle  or to even let them have a diffusive border.

USING THE AWT- AND SWING-CLASSES, THE DRAWING OF AN IMAGE CAN ONLY TAKE PLACE IN A SO-CALLED SCREEN-COMPONENT²²⁹; therefore the following text on images anticipates the use of screen-components and delays the discussion of their features (beginning with page 540).

(Graphical-Objects) Constructing images as a structured collection of single points often can be considered impractical. Simple diagrammatic drawings can be realized by using shapes like ovals (special case: circle) and rectangles (special case: square) within a given coordinate-system. Software-packages called graphics-tools or graphic toolboxes (graphics-library or graphics context) furnish those functionalities to application-programmers. In Java, a graphics-toolbox is given by native code referencable by an identifier of an abstract type²³⁰ `java.awt.Graphics` or by the Java2D package extensions (mostly found in the package `java.awt.image`) or the Java3D packages. The latter two are not discussed in this text.

In the case of graphical-objects, converting the shapes and associated coordinates into a bitmap, for on-screen-display or printing, is called **rendering**. In the case of the native graphics toolboxes this rendering is done by native software and not by the Java software.

Rendering usually takes more time than copying a bitmap onto the screen. Therefore rendering an image directly onto the screen may let the end-user have the impression of slowness. One method to avoid that impression is to render the image within the computer's memory and to copy the resulting bitmap into the screen's memory; this is called **double buffering** (treated on page 535).

(Screen-Components) Screen-components are entities represented on the screen by (usually various) images: Buttons, lists and windows are more than single, simple images, they provide the basis for differing reactions to mouse-clicks and keyboard-signals. For example, different images may show different states of a button-screen-component: The states “non-focused”, “focused” and “clicked” of the same screen-component may be represented by showing three different images. — Especially in Java, there are now two basically different concepts of how to implement screen-components: The **AWT-approach** reuses components and event-infrastructures that are native to the platform, whereas the Swing-approach tries

²²⁹ A Java programmer may desire to freely use the entire screen as a painting area, without respecting the desktop of the operating-system. This may be possible by putting onto the desktop a window of maximal size and by drawing onto this screen-component (for an example see page 610). OR, beginning with Java Version 1.4, by using a special Java package-extension for drawing directly onto the desktop, namely the full-screen exclusive mode API of the AWT-package.

²³⁰ The platform's basic graphic capabilities are accessible for the Java programmer through pieces of native code, which are controlled from the Java Runtime Environment (JRE) by using factory-like-methods that return references of abstract class-type. These Java-references make it possible to access native code by using the specifications given by an abstract class. Remember also the paragraphs following page 184.

to be as independent as possible from platform-specific aspects. The latter, the **Swing-approach**, is consequently given more room in this text.

Abstract Window Toolkit (AWT): 1. AWT like Swing — a programming-library, an → Application Programming Interface (API) for implementing a → Graphical User Interface (GUI). Contrary to Swing, which uses only simple native drawing routines, the AWT reuses the native code of the platform-specific → screen-components.
2. The notion AWT also includes the files that implement this AWT-API and are part of the Java Runtime System (JRS).

Concept: Computers and Images

Concept: Computers and Images — The following is intended to give only a really superficial impression of the wide field of handling images in computing (still-images, animation, three-dimensional spaces, image-scans, photos, video cutting, video production). Here are presented some general notions and finally two basic examples of image-bitmaps in Java.

Programmers who want to write screen-components with total freedom of graphical design and programmers who want to load and display images (for example of GIF- or JPEG-type) have to consider closely how images are manipulated byte- or bitwise.

An object-oriented programming-language may furnish a type or class for storing and manipulating image-information. Just like in many programming-languages, the String-class furnishes an elementary type for textual data.

(Coding) Especially image-data can be coded in many forms depending on convention or usefulness. There are file-formats for storing images on persistent storage-devices like hard-disks (TIFF). There are types or classes for representing image-data in the computer's Random Access Memory (ARGB-model). Compressed image-information becomes useful for transferring the image on floppy-disks or over busy networks (GIF or JPEG). Images coded in the Postscript-language (PS or EPS) are human readable and independent of output-device (printer, screen).

(Processing) As can be seen above, the same image-information may be usefully represented in different formats (compressed for transfer or storage, uncompressed for being easily brought on-screen). Format changes that apparently do not change the visible information for the end-user may be called **coding-transformations**. Image-coding-transformations for purposes of output are called **rendering-operations**. Manipulation of the image which may be noticeable to the end-user may be called **content-transformation**. Multiple steps of processing may give an entire **process-chain**. Adding information or extracting information even may give a **processing-graph**.

A processing-graph may be activated on special request (pull) or may produce the result on its instantiation (push). These are the same triggering options as presented by the graphic on page 274, here only the denotations have been changed:



The Java AWT image-processing (Image, ImageProducer, ImageConsumer, ImageObserver) follows the PULL-concept deemed useful for Internet-browser type of image-loading: The consumer (or sink) activates the loading, transformation or display of the image, but may have to wait for the results. Though there are ImageFilters possible which act like sink-source combinations (for cropping or resizing images) the raw image-data is not automatically available within the initial AWT-context.

The java.awt.image-package contains many of the Java2D classes that intend to remedy this lack of accessibility. (BufferedImage, ColorModel, ColorSpace, Raster, SampleModel, DataBuffer) It reuses the initial AWT-PULL concept (ImageProducer, ImageConsumer, ImageObserver).

The package javax.media.jai (Java Advanced Imaging package) introduces a variety of classes for image-description (RenderableImage, RenderedImage) that implement a PUSH-concept and can make an image fully and extensively editable. (Not covered by this text.)

The following text only addresses basic aspects that may become of some relevance within the fundamental screen-components of a Graphical User Interface (javax.swing, java.awt).

(GUI and Images) Images are a fundamental part of the front-end of any Graphical User Interface, images are the standard means for any output of data. Although on-screen, even characters of a text are

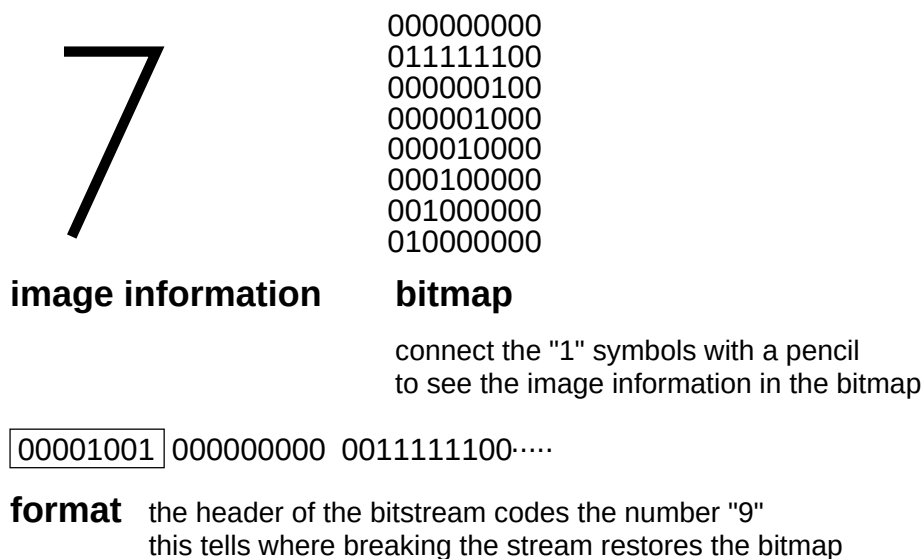
given as (parts of) images, they are nevertheless perceived as symbols with a clear meaning that can be grasped by an often unintuitive decoding-mechanism (dictionary and grammar book). Thus, images are used to represent such diverse entities as the character of a written language (glyph), screen-components and pure graphical representations.

As an example reconsider labels and checkboxes and all those windowing-objects which occur as images on the screen: A label is just an rectangular image (this image may be transparent in some areas of the rectangle and therefore seems to exhibit another shape), nevertheless a label is represented by just one image. A checkbox-component has an image for its unchecked state and an image for the checked state, both can be displayed depending on the state of the corresponding screen-component.

Graphical User Interfaces usually are constructed from such pre-written components, like those pre-written components which may be taken from a library like `javax.swing`.

The Concept of Image-Formats

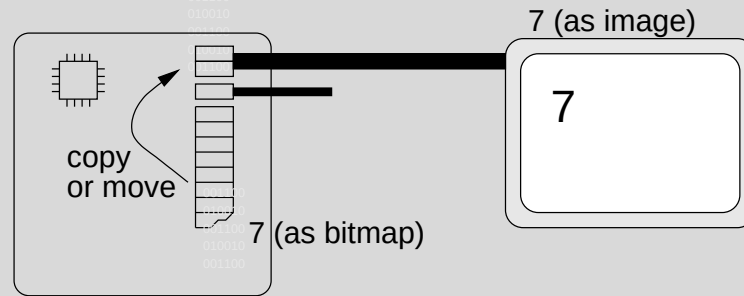
The Concept of Image-Formats — Remember that data on computers is stored as arrays of bytes. A format tells how to interpret such byte-sequence, a type even furnishes means for manipulation²³¹. Early computer-systems stored unicolored, (black and white) images, initially called bitmaps. These bitmaps bitmap encoded the entire picture:



So, files, streams or simple arrays of bytes, storing or transferring image-data, have to give additional information for restoring the original bitmap. This information is called the encoding, the class or the **format of the image-file**. Again, an image-file is a succession of bytes which encodes the image-information. Well-known formats (encodings of image-data) are GIF and JPEG, they also describe two ways of compressing image-data. Programmers and even end-users may be confronted with many other formats (PNG, TIFF, ...). To make the image manipulable on the computer or visible on-screen, the program has to transform the format into a form that is manageable or displayable by the runtime- and/or operating-system.

²³¹Formats or encodings have been introduced beginning with page 33. Types occurred first on pages 73 and 83.

Display 1 (How does image-information appear on-screen?) As has been already indicated in the graphic on page 28, the Random Access Memory (RAM) may occupy only parts of the processor's address-space. (Usually the processor's address-space limits the volume of later RAM-upgrades.) The other part of the address-space of the processor may be occupied by read/write memory of hardware interface-cards for outside communication. Especially a graphics-card may occupy address-space which is reserved for the bytes that represent the screen-output-information. Written into this area, properly formatted bitmaps directly create an image on-screen. Thus, images given as hardware-specific bitmaps, written into the proper address-space appear on the computer-screen. This kind of drawing usually is reserved to drivers on the level of the operating-system. Continued in Display 2 (page 535).



(Compression Formats) Up to now most examples of Java source-code and data were fragile to modifications: Almost any random change of (source-)code lets it become unfunctional. Almost any byte, any token is assigned a definite meaning in the programming-context. Standard code usually uses the memory very efficiently, therefore its sensitivity to changes.

But images, audio-clips and movies generally do not mind small changes. Usually the information they convey is highly redundant. That means omitting suitable content doesn't change the meaning intended to be recognized by the user.

These sections consider such non-symbolic data (images) and its transformation to characteristic formats and human readable forms. Compression algorithms seek ways to filter out the redundancies in image- or audio-files and thereby make them smaller, this is useful to increase transfer-speed through data-lines.

As an example, consider an image with very few colors stored with the direct-color-model. Converting that image to be stored using the indirect-color-model reduces the number of bytes noticeably without compromising on the original information of the picture-data. (Both models are presented in the next sections.)

Another example presents the Run Length Encoding (RLE) used by FAX machines: The document to be faxed is digitized into lines (of black- and white-points), then the RLE-algorithm, successively traversing the points of each line, notes only the points where a change from black to white does occur, or a change from white to black does occur, and then sends the distances between those changes. Therefore a FAX with little unbroken print is sent faster than a FAX with complex and diverse black and white structures (see the bitmapped image of the "7" in the previous paragraphs).

Still another example gives the ZIP-compression-algorithm, which is furnished within a pre-written package (java.util.zip) in the Java Runtime System (JRS).

The examples above described **non-destructive methods of data-compression**. **Destructive compression** can only be used on data that encodes non-symbolic information (audio, video), because during the compression, hopefully unimportant, data gets lost. The methods of the destructive compression are to be closely chosen according to what kind of information is supposed to be conserved: Algorithms for destructive compression are used in producing GIF- and JPEG-formatted images. The GIF-formatted images have better quality and lesser size if the original image has few colors and simple shapes. The JPEG-images are useful for photos with many color tints and unsharper shapes. A relatively well-known destructive compression format for audio-files is given by the MP3-format.

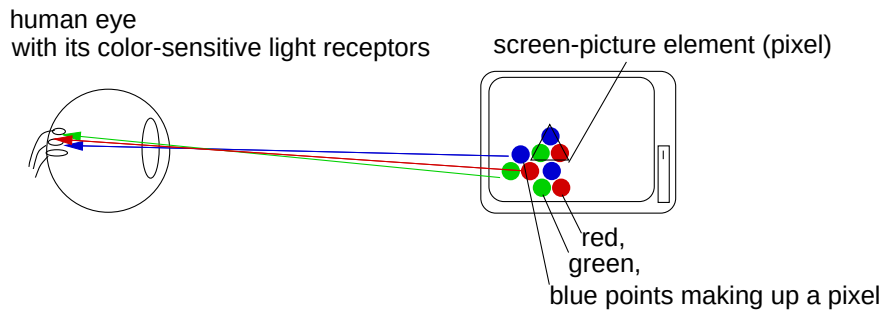
Concept: Colors

Concept: Colors — Today, images appearing on-screen or even on printers usually come in color. How can that be made possible? The human eye sees color only because it has nerve endings that are sensitive to red, green and blue light²³². In this context, white light can be produced by mixing equal intensities of red, green and blue light. Therefore, to recreate the impression of color, it suffices to produce mixtures of red, green and blue light. Exactly this is done on the color-screen: An average color-screen consists of picture elements which themselves are made up of three color-points: red, green and blue. Such a picture

²³²That human sensitivity has no constant and absolute quality: If you stare a longer time at a bright red object and then close your eyes, you may see a green image of that object. There are even people who may have a deficiency in recognizing colors.

element of the screen can code one color-point of an image by specifying the intensity (0.0 to 1.0) of each color-channel: The triple (0.5, 0.5, 0.7) represents one colored point of an image²³³.

(To reduce the cost of the printing of this text, the images below are not colored. The differing colors have been represented as gray shadings. Please, use color pencils to give the pictures their proper coloring.)

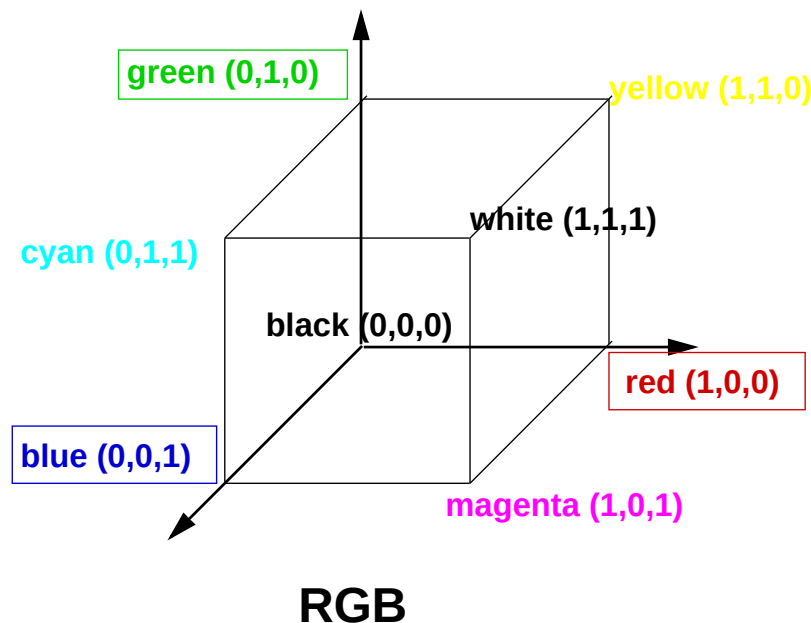


The RGB Color-Model

The RGB Color-Model — In case colors are described as mixtures of red, green and blue light adding up to white, if properly mixed, then the colors are called to be described in the **RGB-model** (RGB stands for Red, Green and Blue):

RGB
color-model

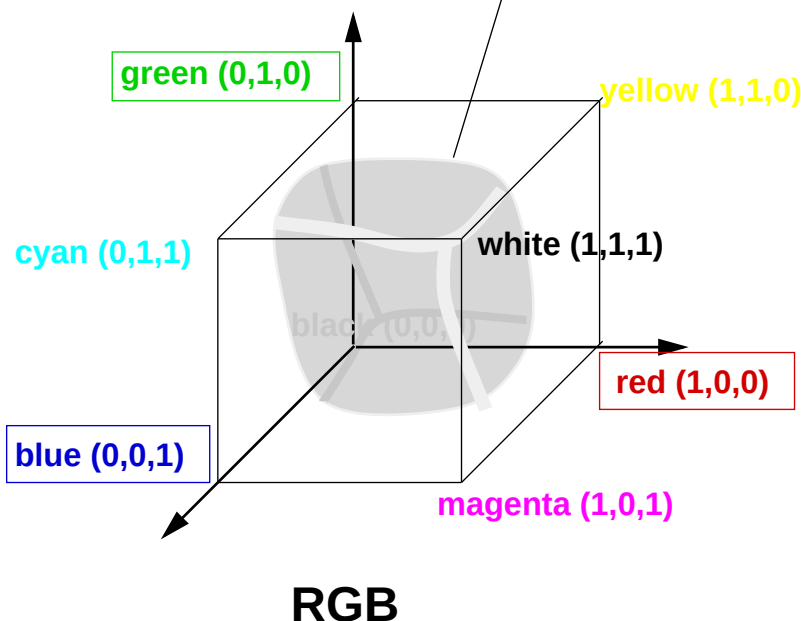
RGB-(Red-Green-Blue) Color Model



Since the parameterization of the RGB-model makes the colors of highest intensity (=1) add up to white, THIS MODEL IS USEFUL FOR DESCRIBING COLORS OF LIGHT-SOURCES LIKE COLOR-POINTS OF A SCREEN-PIXEL. Unfortunately brightness of a red, green or blue light-emitting point on the screen often depends non-linearly on the voltage signal of the screen's hardware and may not fully span the idealized range of the color-cube above. The transformation to be done to represent the colors properly on the screen is called the **gamma-correction**. The subspace of the color-cube, that is actually representable, is called **color-space** of the output-device.

²³³ Light can be described as a mixture of electromagnetic waves of differing wavelengths and differing intensities. The visible wavelengths are found in the interval from 400 to 800 nanometers, fractions of $10^{-6}m$, less than a micrometer, a millimeter reduced by the factor thousand. Light characterizable by a small subinterval of higher intensity is perceived as colored. This opens a more refined view on describing colors than the one presented in this text. But further considering here the wavelength-intensity description of light and color would not advance ideas of the Java Programming Environment.

Color-Space of Output-Device (actually printable colors)



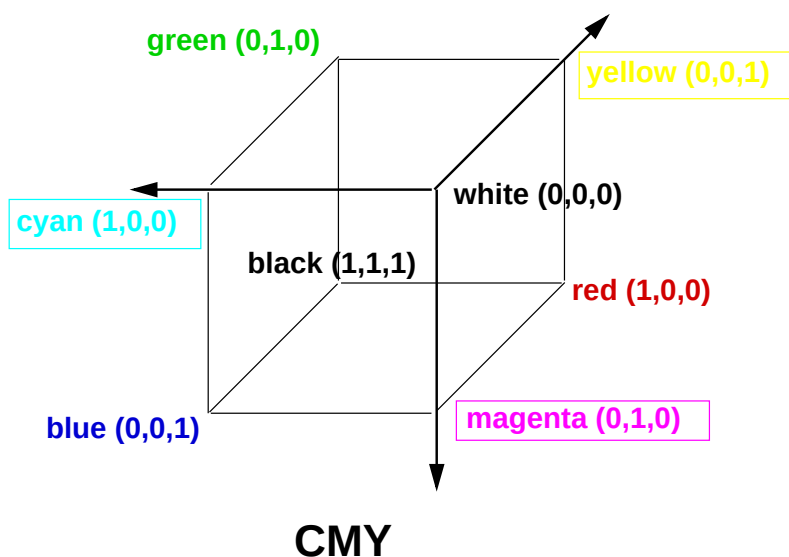
The CMY Color-Model

The CMY Color-Model — CONTRARY TO THE SCREEN-COLORS WHICH ARE EMITTED FROM A LIGHT-SOURCE, IN PRINTING THE COLORS EMERGE FROM REFLECTING LIGHT AND ADDING MORE PRINTING COLOR MEANS REDUCING REFLECTIVITY, THUS REDUCING THE OVERALL REFLECTIVITY. THE SAME CAN BE SEEN WHEN FILTERING LIGHT: ADDING RED, GREEN AND BLUE FILTERS PRODUCES A (NEAR) BLACK RESULT.

The color-cube above can be re-parameterized in various ways. A simple reparameterization, that emulates the filtering of colors, gives the **CMY color-model** (CMY stands for Cyan, Magenta and Yellow). In the CMY color-model the colors add up to black when mixed:

CMY-(Cyan-Magenta-Yellow) Color Model

coordinates differ from RGB! - the color cube remains the same!



In this choice of parameters the colors of highest intensity (=1) add up to black. Observe that the color distribution on the cube has not changed, only the parameterization has changed!

The CMYK Color-Model solves the Printing-Problem: $C+M+Y \neq \text{Black}$

The CMYK Color-Model solves the Printing-Problem: $C+M+Y \neq \text{Black}$ — Analogously to the problem of a gamma-correction on screens, there exists a similar concrete technical problem in printing: Mixing real printing-colors does not reduce the reflectivity to null — black. Therefore, for purely practical reasons, the CMY color-model is extended by a black channel:

The CMYK-model contains an additional black component to compensate the deficient blackness of a mixture of printed cyan-, magenta- and yellow colors. So, the black K channel is only added because the colors used in the printing-process on paper cannot produce a pure black, they also cover only a subspace of the color-cube. This explains why the CMYK-model is mainly used in context of printing. Again the coordinates of a color have to be appropriately transformed to yield a proper result on the output device.

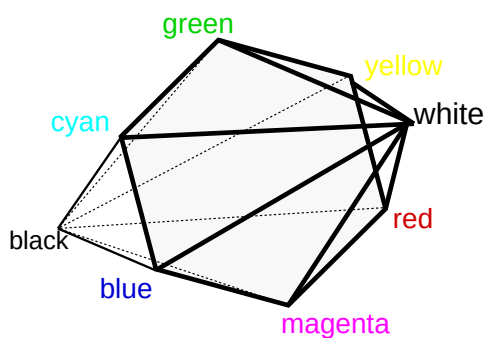
Two other very similar color-models are often used in computing contexts:

The HSV and HSB Color Models

The HSV and HSB Color Models — Deformations of the color-cube above along the black and white axis yield the HSX color models, where the placeholder "X" stands for L/B and V:

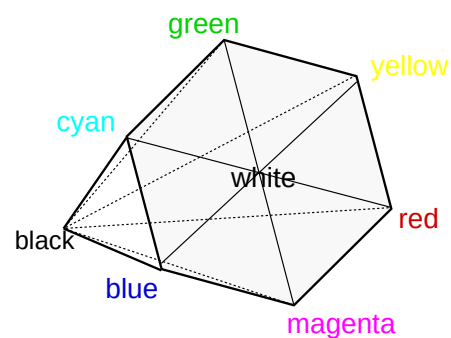
the color cube with eight corners is deformed so that:

six of eight corners are in one plane



HSB = HSL

seven of eight corners are in one plane



HSV

H stands for “hue” and means the gradual change of colors from red, yellow, green, cyan, blue, magenta back to red:  The colors are often presented arranged as a circle, instead of occurring as a hexagram, as shown in the graphics above. “Hue” has a RAINBOW sort of quality.

S tells about whether the full color-information is provided (“deep full red”) or only fractions of it have been added to white (“only a light tint of pink”). The character “S” stands for “saturation” and gives a value that describes “how far that color is separated from” an even admixture (adding up to white) of all the colors. In the diagram above, this is the parameter that varies along the line between the center of the circle (or hexagram) and the point on the circle chosen by the value of hue. The saturation of a color is diminished when a color DISSOLVES into white, gray or black.



B=L stands for EVEN admixtures of all other colors until white or black is reached. This value represents a position along the black- and white axis in the graphics above.

V stands for “value”. The HSV color-model just may be seen as a variation of the HSB color-model.

Two Basic Image- and Color-Formats

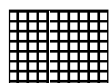
Two Basic Image- and Color-Formats — Images that contain any continuous changes of color and intensity, often are stored rasterized into picture-elements, each picture-element furnishing the entire color-information in terms of a triple of numbers of some color-model. A technical device doing exactly this to pictures, which are given on paper, is called scanner.

The digitized pictures can be stored in different ways inside a computer. A simple way to store bitmaps has been indicated in the graphic on page 521.

Direct Color-Model: ARGB

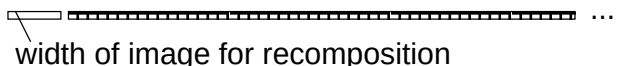
Direct Color-Model: ARGB — The pixels of an image may be stored just as an array together with the width of the image in pixels.

Format: Direct Color Model



picture, digitized into pixels

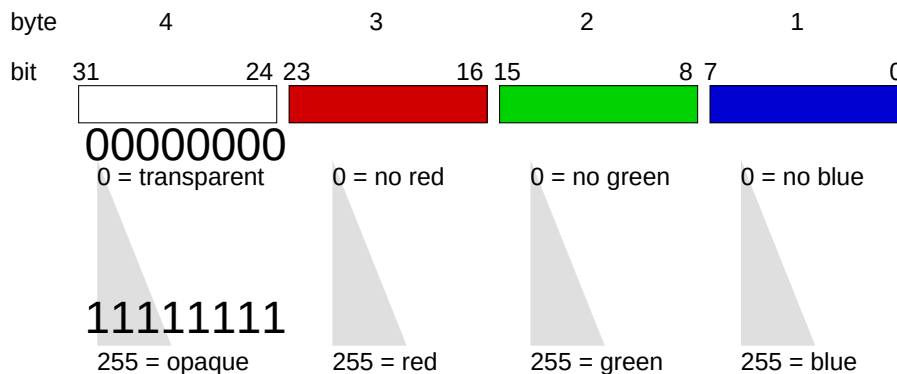
transformed into an array of pixels



Ignoring further format considerations, consider the coding of a single picture-element (pixel): The default color-model for pixels, which Java (`java.awt.image.DirectColorModel`) uses for storing (digitized) images, is given by the ARGB-model: Each pixel is represented by four bytes, stored as an integer-variable of the Java primitive-type `int`. The `int`-value has no meaning as a number itself, but each byte of its four bytes, which are taken to code a value from 0 to 255, is interpreted differently as follows: The most significant byte shows how transparent the pixel is (The historical denotation for transparency is "alpha", therefore the "A"). The rest of the bytes store color intensities in the RGB order (therefore the name "ARGB-model").

A Pixel, represented in the ARGB model:

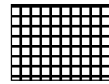
a single (32 bit, 4 byte) int value:



Indirect Color-Models

Indirect Color-Models — Another conceptual way of storing colors means to take a list of the colors that appear in the picture. The color-information, which each pixel needs, is then given by an index of that color-list. This procedure is also called index color-model and Java furnishes a class of that kind: `java.awt.image.IndexColorModel`. As a general rule, an index color-model becomes useful when there are very few colors used in an image. (Using four byte per pixel gives the absolute freedom of the choice of color. But, if there are found less than 265 different colors in a picture, then this freedom gives an overhead of three unnecessary bytes per pixel, when neglecting the overhead of the color table of the index color-model.)

Format: INDirect Color Model



picture, digitized into pixels

transformed into an array of pixelcodes



width of image for recomposition AND
table with used colors and color-codes

Java: Transparency, Partial Transparency and Opaqueness

Java: Transparency, Partial Transparency and Opaqueness — Consider an image, color-coded with an alpha-channel. Then any output processing (screen, printing) can determine how intensely background color-information may be admixed to the pixels of the image. A zero alpha-value for a pixel triggers the use of any available background- or default color-information discarding the current pixel color, thus making the pixel fully transparent. A maximal alpha-value (255 if coded as a byte) represents opaqueness, the pixel color remains as given, no background-information is used. Any alpha-values in-between trigger the weighted mixture of the color-information, thus attaining a partial transparency.

Apart from the basic image-formats which have been prepared by the classes `java.awt.image.IndexColorModel` and `java.awt.image.DirectColorModel`, Java furnishes a type that solely codes color-information, including an alpha-value (Java Version 1.2 and higher): `java.awt.Color`. And Java furnishes a reference to system-related colors: `java.awt.SystemColor`.

The ranges of the color-channels vary within the values 0 to 255, this also means one byte per dimension of the color-cube that has been introduced previously on page 523. Adding one byte for the alpha-value for determining the partial transparency (zero for glass-clear, 255 for opaque).

0 =
GLASS-CLEAR

In Java, the AWT screen- and Swing-components use `java.awt.Color`-objects to specify color-values. **Partial transparency** of pixels requiring an alpha-value can be used – where supported – for custom AWT- or Swing screen-components.

Java: `java.awt.Image`

Java: `java.awt.Image` — As indicated in the context of PUSH/PULL image-processing on page 520, Java uses the class `java.awt.Image` for referencing images. The `Image`-class is declared abstract, nevertheless references of the type `Image` are available by calling factory-like-methods. WHY? Because references of type `Image` usually point to platform-specific representations of images and NOT to Java objects²³⁴. Naturally, when referencing image-data outside Java's domain, the possibilities for image-manipulation are somewhat restricted: As already mentioned, the `Image`-class initially has been designed for loading images from the Internet into a browser-like environment and displaying them, not for handling fully-developed image-manipulation.

references of
abstract
class-type
`Image`

Thus identifiers, referencing image-data, may describe their data as being of abstract `Image`-type. How to take image-data and turn it into images on-screen? Screen-components use the triggerable²³⁵ methods `paint()` or `paintComponent()` (Swing) which import a reference of (again) abstract type `java.awt.Graphics`. Here another case of a reference, typed as an abstract class, is presented: An identifier of type `Graphics` references the native software, which in turn can be used to draw directly on the screen-area given by the associated screen-component: The Java Runtime Environment lets references of this type access native code that implements the functionalities of the abstract `Graphics`-class. These functionalities include various methods for drawing on-screen inside the component's screen-area.

A prototype for such an application is given by the “`ViewImage`”-class:

```
import java.awt.Toolkit;    // abstract, direct access to platform-specific resources.
import java.awt.Image;      // abstract class,
                           // for referencing (native) image-code (JPEG, GIF).
```

²³⁴ Here the abstract `java.awt.Image`-class permits a Java-identifier to access to the native image-data. The reuse of the modifier `abstract` for referencing native code has been introduced on page 184 and is further exemplified on page 551 for other AWT-classes.

²³⁵ So-called triggerable methods have been introduced on page 127.

```

import java.awt.Graphics;    // abstract class, references native code to draw on-screen.
import javax.swing.JComponent; // abstract class,
                               // anonymously extended to a class, instantiating a DRAWING-SURFACE.
import javax.swing.JFrame;   // non-abstract class,
                               // DESKTOP-WINDOW containing the drawing-surface for display.
/* Identifiers of the class-types given by the first three abstract classes
   have access to native code or data.
   The fourth abstract class is anonymously extended and
   anonymously instantiated in the code below.
*/

class ViewImage
{
    public static void main( String imageFileName[] )
    { String fileName = imageFileName[0];
    // ----- read image from current directory
      final Image anImage = Toolkit.getDefaultToolkit().getImage( fileName );
      System.out.println( "Image reference: " + anImage );
    // ----- display image
      JFrame screenArea = new JFrame( fileName );
      screenArea.getContentPane().add
      ( new JComponent()
        { // ^ drawing-surface: given as anonymous class and as anonymous object
          public void paintComponent( Graphics drawingTools )
          { drawingTools.drawImage( anImage, 0, 0, this );
          } }
      );
      screenArea.setSize(200,200);
      screenArea.setVisible(true);
    } }

/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > javac ViewImage.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > java ViewImage TestImage.jpg
Image reference: sun.awt.motif.X11Image@133796
(Ctrl-C in terminal-window)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput >
*/

```



The command `java ViewImage name.jpg` — issued in the terminal-window of the native operating-system — shows the image-contents of the JPG-file *name.jpg* (located in the current directory) in a frame on the screen. Access to the platform-specific form of the image is obtained by a factory-like-method (`getImage( ... )`) in the same way as is obtained (`getDefaultToolkit()`) a reference of the class-type `java.awt.Toolkit`, which gives access to various platform-specific resources. Although the `Toolkit`-class has been declared abstract, the static (non-abstract!) method `getDefaultToolkit()` makes it possible to return a valid reference of type `Toolkit`!

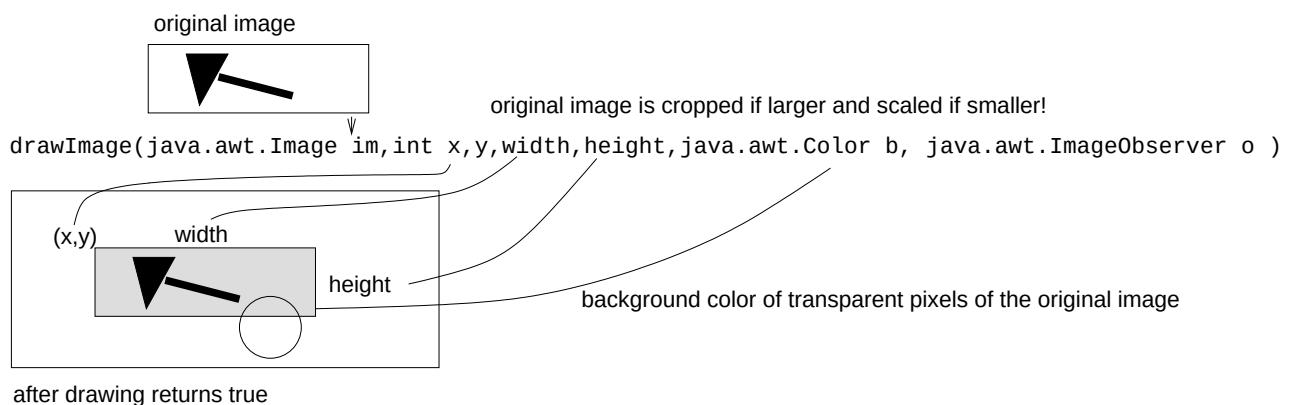
Images can be displayed only on a drawing-surface-object of type `java.awt.Component`, `java.awt.Container`, `javax.swing.JComponent` or sub-classes of these. A `javax.swing.JFrame`-object furnishes the screen-area with a content-pane onto which a screen-component with a customized `paintComponent()`-method is added that actually draws the image.

The following table collects various methods which return a reference of type `java.awt.Image` pointing to a native (=platform-specific) image-entity. Even if those native image-entities are called Image-objects,

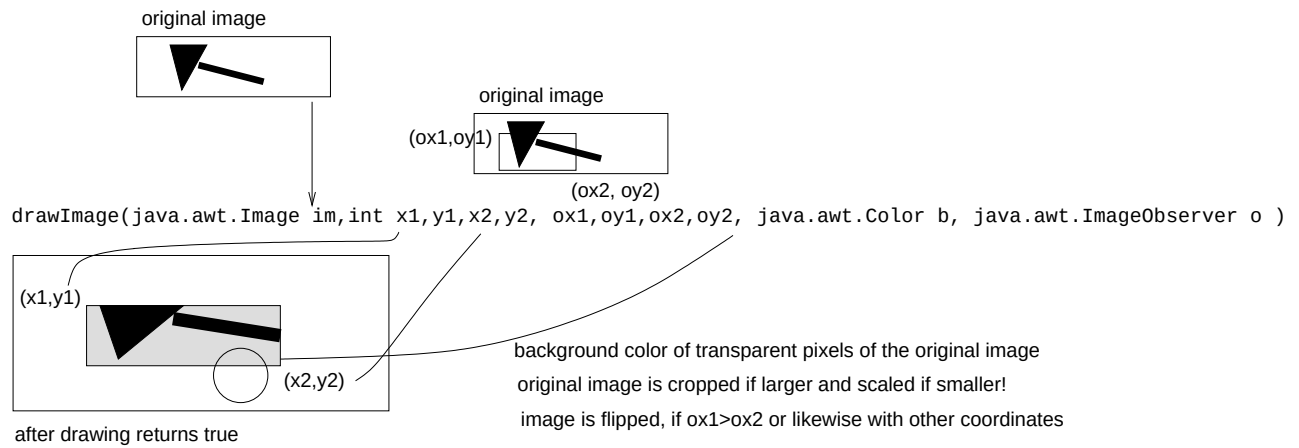
they are NOT Java-objects! The corresponding arguments in these factory-like-methods²³⁶ are names of image-files, Uniform Resource Locators, or arrays of bytes containing image-information in specific formats:

class	method-signature	method functionality
java.awt....		
Component	Image createImage (width, height) Image createImage (ImageProducer)	Loads, caches, - produces an empty image in CPU's memory. - Produces an off-screen image when doing double buffering and background drawing. See also within the table beginning on page 553.
Toolkit	Image createImage (byte [])	Transforms a byte-array of GIF, JPEG or XBM format into a native image-object.
applet.Applet	Image getImage (URL) Image getImage (URL, String)	Returns a reference of an empty image for future loading as needed. If loading has already taken place, just gets the reference.
Toolkit	Image getImage (String)	same as above
Graphics	boolean drawImage (Image,...,ImageObserver)	Loads, caches, - puts an image in the computer's memory. - Asynchronously/non-blocking within a new thread. - Returns true if image is already loaded. Since the image is taken within a graphics context it is ready for display.
Component	boolean prepareImage (Image,...,ImageObserver)	Loads, caches, - puts an image in CPU's memory. - Asynchronously/non-blocking within a new thread. - Returns true if image already loaded.
Toolkit	boolean prepareImage (Image,...,ImageObserver)	same as above

The following graphical displays intend to give an impression about different possible uses of the overloaded method `drawImage()` within the abstract class `java.awt.Graphics`. This also illustrates some of the limited possibilities for manipulating an image of class-type `java.awt.Image`:



²³⁶(Factory-methods return a reference to a Java-object of a regular class-type.) Factory-like-methods return a reference to a native entity which is given an abstract Java class-type, see also the gray box on page 185. (Or factory-like-methods return references of pre-existent Java objects, as an example consider the instance-method `Object.getClass()`.)



Typical image-observers are screen-components or progress-bars. They implement an `imageUpdate()`-method that is called by the Java Runtime Environment (JRE) each time it notices a change in the image's processing-status. This concludes the cursory look at the initial AWT-PULL-model of imaging with `Image`, `ImageObserver`, `ImageProducer` and `ImageConsumer` classes/interfaces. For more, including an `imageUpdate()`-method in action, see page 606.

Java: Simple Bitmapped Images

Java: Simple Bitmapped Images — In this section two image-bitmaps are made displayable on-screen. This gives just a demonstration of what can be done within the aforementioned AWT-PULL-model of imaging; for a more effective and explicit coding of image-data see the Java2D programming environment (mostly found in the package `java.awt.image`) and see the description of Java Advanced Imaging (JAI). In the Java source-code, references of the `Image`-type can be used to scale the referenced image and to retrieve information about the image. (The Java2D programming interface contains the non-abstract `Image`-extension `java.awt.image.BufferedImage` which gives more access to the image-data.)

The following program produces an image with pixels of a random ARGB-color-coding. And here a `java.awt.Image`-class wraps a Java-specific `java.awt.image.MemoryImageSource`-object, instead of a platform-specific piece of (native) image-data!

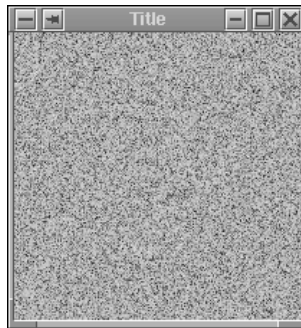
```
import java.awt.Graphics;           // abstract, method-calls are mapped onto native code
                                   // for drawing on-screen.
import java.awt.Image;             // abstract class gives references to (native) image-code.
import java.awt.image.MemoryImageSource; // An ImageProducer,
                                   // produces an image from an integer-array.
import javax.swing.JComponent;     // abstract, anonymously extended to a class,
                                   // instantiating a DRAWING-SURFACE.
import javax.swing.JFrame;         // DESKTOP-WINDOW containing the drawing-surface for display.

class HelloBitmap
{
    public static void main( String unused[] )
    {
        final int width = 200,
                  height = 200,
                  firstPixel = 0,
                  line = width; // number of pixels per horizontal line
        final int [] pixelArray;
        /* final because createPixelArray requires it,
           otherwise another thread may change values during computation.
        */
        // ----- production of byte-array
        pixelArray = createPixelArray( width, height ); // defined below
        // ----- drawing-surface
        JComponent drawingSurface = new JComponent() // inner class
        {
            Image image;
            {
                // non-static initializer of inner class
                image =
                this.createImage(new MemoryImageSource(width,height,pixelArray,firstPixel,line));
            }
            public void paintComponent( Graphics tools ) // or only paint()
            { tools.drawImage( image, 0, 0, null );
            }
        };
    }
};
```

```
// ----- drawing on-screen
    JFrame screenArea = new JFrame( "Title" );
    screenArea.getContentPane().add( drawingSurface );
    screenArea.setSize(200,200);
    screenArea.setVisible(true);
}

// ----- production of byte-array
static int [] createPixelArray( int width, int height )
{ int pixelArrayLength = width * height;
  int [] pixelArray = new int [ pixelArrayLength ];
  for( int i = 0; i < pixelArrayLength; i++ )
  {
    pixelArray[i] = (int)( (java.lang.Math.random() - 0.5) * 2 * 2147483647 );
  }
  // to cover the range of int-values randomly
  return pixelArray;
} }

/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > javac HelloBitmap.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > java HelloBitmap
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput >
*/
```



Remarks: Variables of primitive-type `int` determine the size of the image in numbers of pixels. An `int`-array (`pixelArray`) takes the image-data. The image-data is produced as an array of randomized values in a separate static method (`createPixelArray()`).

Then the drawing-surface is defined: The basic component in the type-extension-hierarchy of Java Swing-objects, namely `JComponent`, is chosen to be the drawing-surface (Later, to put the drawing-surface-object on-screen, the drawing-surface is added to a screen-area derived from `JFrame`. `JFrame`-objects are one out of three types (`JFrame`, `JWindow`, `JDialog`)²³⁷ that can be placed directly on the desktop of the current operating-system's user-interface. Those objects then act as containers for other components to be displayed.)

The drawing-surface is defined inlined, as an inner class of anonymously extended type `JComponent`, immediately instantiated. Within this object, the image to-be-painted is produced in a two-step-process: 1. By transforming a given byte-array into a `MemoryImageSource`-object, referenced by an identifier of class-type `Image` (within the non-static initializer). 2. By calling the `drawImage()`-method within the `paintComponent()`-method of the `JComponent`-object. The `paintComponent()`-method of Swing-based screen-components is called by a part of the Java Runtime Environment that is responsible for the drawing on-screen, the Java programmer only needs to override it as done above (Like the `main()`-method of applications, Java programmers do not call the triggerable²³⁸ method `paintComponent()` from source-code.). AWT-based screen-components use their triggerable counterpart, the `paint()`-method to draw onto their surface.

The next program produces a ring-like, partially transparent structure on the screen:

```
import java.awt.Graphics;           // toolbox for drawing.
import java.awt.Image;              // abstract class gives references to images.
import java.awt.image.MemoryImageSource; // an ImageProducer,
                                     // produces an image from a given integer-array.

import javax.swing.JComponent;      // used as drawing-surface.
import javax.swing.JFrame;          // furnishes screen area.
```

²³⁷ A fourth type, `JApplet` produces objects that appear in browser-windows embedded in HTML-texts, much like images embedded into a text.

²³⁸ So-called triggerable methods have been introduced on page 127.

```

class HelloBitmapARGB
{
    public static void main( String unused[] )
    { final int width = 200,    // final because createPixelFormat() requires it,
      height = 200,    // otherwise another thread may
      firstPixel = 0, // change values during computation.
      line = width;    // number of pixels per horizontal line
      final int [] pixelArray;
    // ----- production of byte-array
      pixelArray = createPixelFormat( width, height ); // defined below
    // ----- drawing-surface
      JComponent drawingSurface = new JComponent()
      { Image image;    // line above: anonymous class - non-anonymous object
        {                                     // non-static initializer
          image =
this.createImage(new MemoryImageSource(width,height,pixelArray,firstPixel,line));
        }
        public void paintComponent( Graphics tools )// or paint() with java.awt.Component
        { tools.drawImage( image, 0, 0, null );
        }
      };
    // ----- drawing on-screen
      JFrame screenArea = new JFrame( "Title" );
      screenArea.getContentPane().add( drawingSurface );
      screenArea.setSize(200,200);
      screenArea.setVisible(true);
    }

    // ----- production of byte-array
    static int [] createPixelFormat( int width, int height )
    { int pixelArrayLength = width * height;
      int [] pixelArray = new int[ pixelArrayLength ];
      int blue, green, red, alpha, x, y, value, div = width * width + height * height;
      for( int i = 0; i < pixelArrayLength; i++ )
      { x = i % width;    // x-coordinate of pixel = remainder when lines subtracted
        y = i / width;    // y-coordinate of pixel = number of lines
        value = (int)(255 * java.lang.Math.sin( java.lang.Math.PI * (x*x + y*y) / div ));
        red = value;      // a red ring
        green = 0;        // no green
        alpha = value;    // 0 = image completely transparent, only background visible
                          // 255 = only image, no background
        blue = 255-value; // ring gets blue by the amount of vanishing red.
      }

      /* The amounts of alpha and red are chosen to change according to the value-function,
        then the color-information is coded into an integer-value
        and stored in the array of pixels:
      */
      pixelArray[i] = Transform.intFromBytes(blue, green, red, alpha);
      // for the 'Transform' utility-class, see below
    }
    return pixelArray;
  } }

/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > javac HelloBitmapARGB.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > java HelloBitmapARGB
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput >
*/

```



The utility-class named “Transform” essentially furnishes two methods: First the method “intFromBytes()”, which takes four bytes, checks them for being values from zero to 255 and then putting them, highest byte first, into a single value of primitive-type `int`.

big-endian

Consider an ordered list of bytes, call the byte on one end of the order the “most significant byte”, and call the byte on the other end of the order the “least significant byte”. If those bytes are reordered for processing (for example for transfer over a network), then “most significant byte”-first is also called “big-endian”-order or “network”-order. A reordering with “least significant byte”-first may also be called “little-endian”-order. The byte-order, used by a specific platform, can be seen in the printed output on page 254.

The second method “bytesFromInt()” breaks an `int`-value down to its four bytes and returns them as an array:

```
class Transform
// uses int-variable to store an array of four unsigned byte-
// values (used by ARGB color model)
{
    static int intFromBytes( int one, int two, int three, int four ) // int range 0..255
    { int [] handover = {one,two,three,four};
      // since array-constants can be used only in initializers
      return intFromBytes(handover);
    }
    static int intFromBytes( int [] b ) // that's b[0],b[1],... least significant first
    // the work is done in the return clause the rest are machinations to avoid errors
    { int [] intArray = new int[4];
      boolean smaller, larger, right;
      for( int i = 0; i < b.length; i++ )
      { smaller = b[i] < 0;
        larger  = b[i] > 255;
        right = !smaller & !larger;
        if(smaller) intArray[i] = 0;
        if(larger)  intArray[i] = 255;
        if(right)   intArray[i] = b[i];
      }
      return intArray[3] << 24 | intArray[2] << 16 | intArray[1] << 8 | intArray[0];
    }
    static int [] bytesFromInt( int n )
    { final int BIT_MASK = 255; // binary = 00000000 00000000 00000000 11111111
      int [] b = {0,0,0,0};    // Cannot be written
                                // in one line return {(byte) n & BIT_MASK, ...};.
      b[0] = n & BIT_MASK;      // Extracts, then
                                // shifting (see page 95) and extracting:
      b[1] = (n >>> 8) & BIT_MASK;
      b[2] = (n >>> 16) & BIT_MASK;
      b[3] = (n >>> 24) & BIT_MASK;
      return b;
    }
}
```

Observe that the proper length of the array has not been checked in the method “intFromBytes(int b [])” above; this may result in throwing exceptions, therefore an appropriate test could be included into this piece of source-code.

The procedures above are the Java-codings of transforming the color-information of one pixel in the ARGB-model into one single `int`-value, back and forth. This is one of the few places in this text in which the role of individual bits has been made explicit²³⁹.

Mask, Bit-Mask: A constant of primitive-type and integer-value; which within high-level programming is to be used as representing a sequence of individual bits. Bit-masks often occur in contexts where information has been stored economically as single bits in bit-maps (image-data, filtering redundancies). Bit-masks are used to extract boolean-information with logical operations. An application-programmer usually works with bit-masks given as constants within classes or objects and uses those constants as arguments in methods (see also the note below the table on page 553).

This text uses bit-masks for explicitly processing individual bits only in the source-code above. There, variables of primitive-type `int` have been used for storing information bitwise in four one byte blocks ($4 \times 8 = 32$) giving a 32bit-block, the `int`-variable.

OUTLOOK: For handling information bitwise, without being restricted to a block-size, consider the Java Collection class `java.util.BitSet` which is not covered by this text.

²³⁹Remember the section on coding beginning with page 33. And reconsider the IP-subnet mask on page 358, another kind of mask.

Concept: Graphical-Objects

Concept: Graphical-Objects — Image-information may be stored in different formats as a sequence of bytes. Generally, bitmap-like formats can easily be displayed on-screen by the computer, but the storage of bitmaps usually takes a lot of memory. Therefore data given in bitmap-format often is transformed by a compression algorithm into another format better suited for storage or network-transfer (coding-transformation). The transformation-processes take computational resources (processor-time, bus-time) but eventually reduce the number of bytes necessary for encoding the image-information.

There exists another way of producing compressed images: Using formulas, a graphical-object can be described most economically; although, when the image is to be shown on-screen, the pixel-information has to be re-calculated each time.

For example, consider the mathematical formula $((x - x_1)^2 + (y - y_1)^2 = c_1$ within some coordinate-system, this formula codes a circle of radius $\sqrt{c_1}$, with its origin at the point given by the coordinates (x_1, y_1) .

This approach can be coded into readily written commands (`drawCircle( x1, y1, sqrt(c1) )`) for describing graphical-objects like lines, polygons, circles and curves. Human-readable²⁴⁰ commands of that kind are found as members of the `java.awt.Graphics`-class or the Postscript-language.

Graphics given by:

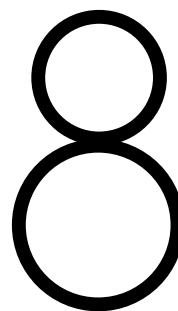
Pixels

```
000111000
001000100
010000100
001111000
010000100
010000010
001000100
000111000
```

```
pixelAt(0,0)=0
pixelAt(0,1)=0
. . . .
```

memory consuming, but
fast to render, fast to print

(graphical) Objects



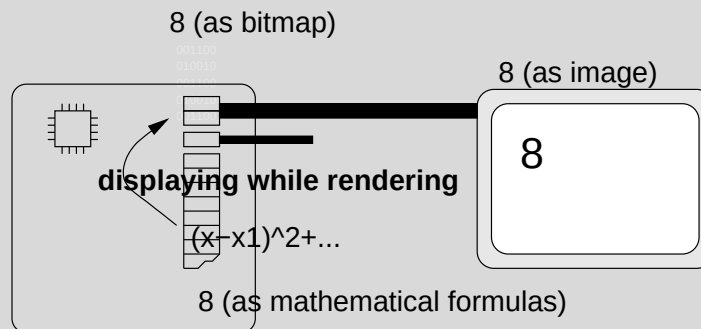
```
(x-x1)^2 + (y-y1)^2 = c1
(x-x2)^2 + (y-y2)^2 = c2
thickness = 5 points
```

consumes little memory, but
needs time to be rendered

Images, characterized by few continuous changes of pixels, for example text-fonts, may be stored as a set of simple mathematical formulas. The mathematical way of describing image-elements takes very little memory-space; but mathematical formulas need more processor-time to be transformed (“to be rendered”) to a rasterized picture for drawing it on-screen or to be printed by a printer onto paper. (Remember the gray box about displaying a bitmap on the computer’s screen, on page 522.)

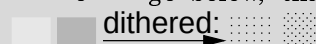
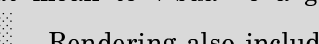
²⁴⁰Although those commands are human-readable, often comfortable what-you-see-is-what-you-get applications help to produce files containing such commands; in a way enabling the end-user to graphically edit the picture.

Display 2 (**Rendering: Original Image to Displayable Bitmap**) Image-information may be encoded binarily in formats not suitable for direct display, due to compression algorithms or through a human-readable description-language (like Postscript). The process of transforming an image-representation to a displayable- or printable bitmap is called **rendering an image**.



If such a “rendering-operation” is not done for output purposes, but just as a transformation of formats (coding-transformation), then this change of encoding is also called **PixelGrabbing** or **Rasterization**.

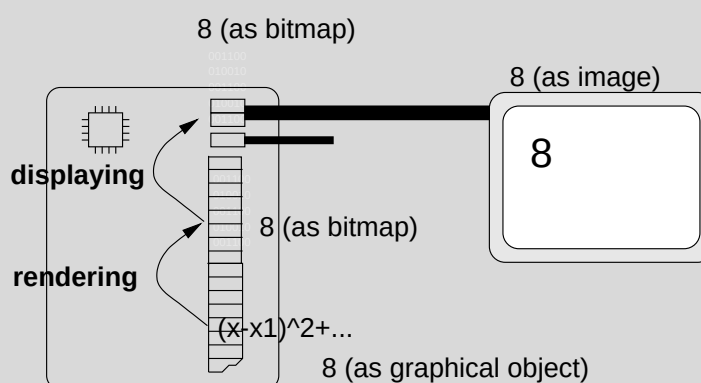
The special case of rendering an individual color- or gray-point by using combinations of other color-points or different densities of black-points on some output-device, is called **dithering**. The points in the image below, that mean to visualize a gray-scale, are an example of a dithering operation

 **dithered:**  . Rendering also includes the suitable transformation of coordinate-systems as indicated in Display 3 below.

Rendering images usually becomes a processor-intensive task, and thus a relatively slow process. If the image is rendered while being displayed, as shown in the graphic above, then the end-user may see the time-dependence of the rendering-process by a slowly-appearing image on-screen. There exists a trick, that does not save processor-time, but nevertheless gives the end-user the impression of relatively instant availability of pictures: Rendering images into the Random Access Memory (RAM) and then just copying the bitmap into the screen-display area; this avoids the visualization of the rendering-process on-screen and after an hopefully inconspicuous delay, the image appears almost instantly:

Display 3 (**Double Buffering**) RENDERING AN IMAGE — THAT MEANS TRANSFORMING THE ORIGINAL IMAGE-DATA INTO A DISPLAYABLE BITMAP-FORMAT — USUALLY TAKES MUCH MORE TIME, THAN JUST SHIFTING A BITMAP INSIDE THE ADDRESS-SPACE OF THE PROCESSOR. On-screen rendering often shows the end-user how the image is actually constructed and how long it takes to finish it. To give an end-user the *impression* of instantaneous availability of the image, another strategy is used: The image is rendered into an area of the Random Access Memory (RAM). After this relatively slow rendering-process, the bitmap is copied (faster than rendering) into the display area of the processor’s address-space.

"Double Buffering"

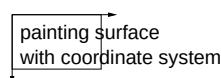


Double Buffering means to finish rendering before copying the bitmap into the display area. Rendering larger images can take a lot of processor-time, so that the end-user notices the associated delay. DOUBLE BUFFERING ONLY CONCEALS THE SLOWNESS OF THE RENDERING-PROCESS BY MAKING THE DELAYING-PROCESS NOT VISIBLE ON-SCREEN.

In some cases it may be useful to show the rendering-process: When loading images over a “slow” network or a hard-disk, the parts of the image that already have been received may be displayed, so that the user can judge if a complete download of the image is desirable.

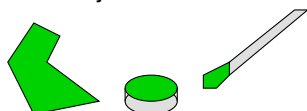
In other cases of delayed reaction of the computer-system, a user may be happy when a clocklike-cursor or a moving progress-bar informs about the state of the transforming- or loading-process. For more see the section about monitoring-processes beginning with page 605.

Painting Graphical-Objects



Painting Graphical-Objects — The basic part of painting-contexts for graphical-objects in languages like Postscript or Java are painting-surfaces given by coordinates: The equivalent to a real canvas. To paint a graphical-object, a programmer needs a **painting-surface (or space) with coordinates in which to paint**. Within this coordinate-system the programmer places the graphical-objects, for example rectangles, text and other shapes. Screen-components of the `javax.swing`-package use special objects for painting: The content-pane (`getContentPane()`) of type `java.awt.Container`. (Java screen-components of the `java.awt`-package furnish no special surface except the drawing-surface `java.awt.Canvas`.)

graphical objects and tools for their manipulation

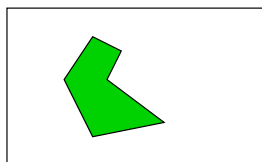


Languages like Java or Postscript give a surface with a coordinate-system and furnish a repository of methods for describing and manipulating a wide variety of **graphical-objects**. Such a repository usually is called **toolbox, graphics-library** or **graphical context**. In Java, such graphics-toolboxes become acces-

sible by obtaining references (of abstract class-type `java.awt.Graphics`) to native code: Either by getting those references as arguments of triggerable methods, see the paint-methods below, or by calling factory-like methods²⁴¹. A typical drawing-method of such graphics toolbox²⁴² can be taken to be `drawRect(x, y, height, width, empty)` which draws a rectangle using the indicated parameters. — Thus the equivalent to a painter's color-jar and a pencil is given by a `java.awt.Graphics`-toolbox with its drawing-methods. This toolbox establishes the direct contact between the Java Runtime Environment and the drawing routines of the operating-system. That contact is realized by the Java commands used for producing shapes and colors. — The process of actually painting on the canvas with pencil and color has its equivalent in overriding a triggerable method:

```
javax.swing.JComponent.paintComponent(java.awt.Graphics graphicsToolbox)
java.awt.Component.paint(java.awt.Graphics graphicsToolbox)
```

rendered result on screen or on printer



The `paint()`-methods above are called automatically by the Java Runtime Environment if the associated screen-component is set to be visible on-screen. Furthermore, object-painting-contexts furnish methods that take surface- and graphical-objects and produce arrays of pixels ready for display on-screen. Such methods that transform image-formats into a displayable bitmap-like format are called **rendering-methods**. Rendered images are ready for screen-display or the standard printing-process. A system's call of Swing's

`paintComponent()`-method (or AWT's `paint()`-method) activates such rendering-processes.

Display 4 (Generic Coordinates and Pixel-Coordinates) Often the coordinate-system may be called **output-independent coordinate-system** because it has no relation to the size or distortions of the actual output on-screen or printer. The drawing-surface is not realized as an array of pixels until the actual rendering-process takes place! If necessary, the programmer has to keep the unit length the same on an output device.

Screen and printer are standard two-dimensional output-devices with a limited size of pixels, their resolution is generally given in units of dots per inch (Typical screens may have a FIFTY OR 72 DOTS PER INCH (DPI) resolution, printers around 600 TO 1400 DPI and more, in professional-printing resolutions over 2000 DPI are standard).

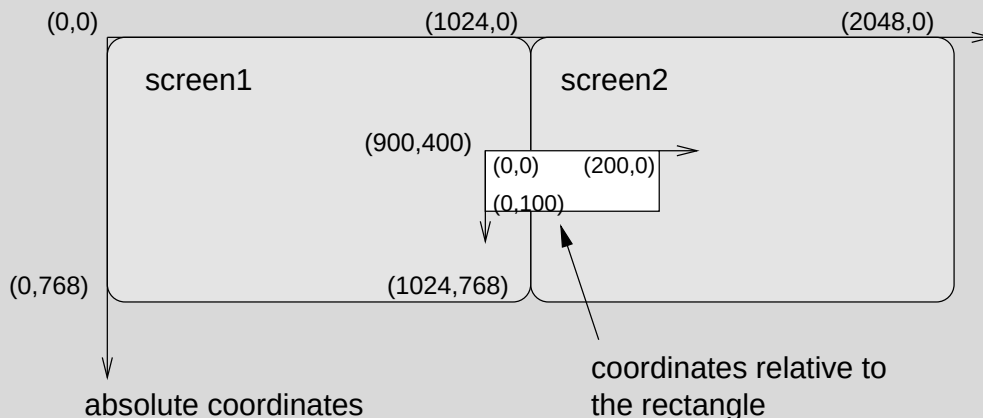
In preparing or rendering images for those output-devices the output-independent coordinate-systems have to be transformed into pixel- or device-specific coordinate-systems. That means: The output-independent coordinates of any non-pixel-oriented painting-method have to be transformed to the **pixel-coordinate-system**.

²⁴¹Factory-like methods have been introduced on page 185.

²⁴²Graphical-toolboxes (`java.awt.Graphics`, `java.awt.image.Graphics2D`) are not to be confused with collections of functions that are given by the native platform in Java: `java.awt.Toolkit` (page 551)

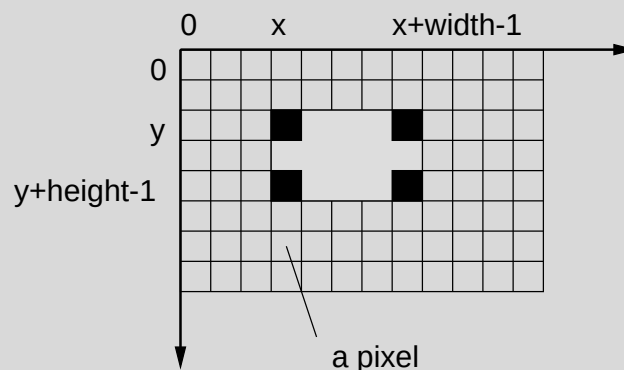
Even the simple pure pixel²⁴³-coordinate-systems of most devices have some common basic properties to be noted:

Display 5 (Pixel-Coordinates) The positioning of a window or any other rectangle on the screen usually is done by using a set of pixel-coordinates. Below is shown a screen-coordinate-system encompassing two screens. The units of that system are given in screen-points or pixels:



Of course, a second screen has to have a proper hardware connection and be supported by the operating-system of the computer-platform to be usable that way. With respect to a given coordinate-space, **locations of rectangles** are always given by the upper left hand corners of the rectangles. A rectangle may define its own little coordinate-space within its area, a **coordinate-system relative to that rectangle**. Most components have the shape of a rectangle or are enclosed by a rectangle. Components exhibiting another shape usually take the space of a rectangle, their shape is given by transparent pixels in the remaining part of their defining rectangle.

coordinates and pixel positions:



Coordinates are usually given a chime-like count, beginning with zero.

The three main notions in painting with graphical-objects are: a space with (generic) coordinates, methods to position and manipulate graphical-objects in that space and finally methods that transform that space

²⁴³Screen-points have a structure, unlike the mathematical notion of a point, therefore the name “pixel” (short for picture-element). If each screen-point can take only two states, namely bright or dark, then the point’s state can be coded with one bit only. Usually screens allow gray or colored imaging. Color imaging automatically means that one point or pixel is made up of three different colors - red, green and blue. The number of colors representable by these three basic colors depends on the capability of the screen’s hardware and on the settings of the operating-system of the computer-platform. Some often used data has been compiled below:

Bits per color	Bits per pixel	Color range:	screen with resolution of 1024 times 768 pixels (many 17” monitors) needs memory
	1	Bright/Dark	1024*768 bit / 8 = around 100 kB
2	6	64 colors (2 ⁶)	around 500kB
4	12(1.5 byte)	4096 colors (2 ¹²)	around 1MB
		millions of colors:	
8 (1byte)	24 (3bytes)	(millions means 16.777.216 = 2 ²⁴)	1024*768*3 byte/1024 = 2304 kB = 2.25MB

A typically available line of pixel-coordinates may encompass 32000 pixels with 72 dpi, this means a length of 900 inches or about 20 meters. Of course, this describes the range of coordinates before rendering, rendering only takes place for coordinates inside the screen’s area!

and the associated graphical-objects into a displayable form - mostly arrays of pixels - the rendering-methods.

Java: Object-Painting on Screen-Components

Java: Object-Painting on Screen-Components — Beginning with Java Version 1.1, any screen-component can be used as a painting-surface; previously, painting was possible only on screen-components of type `java.awt.Canvas`. Screen-components instantiated from classes out of the `javax.swing`-package use a drawing-background, named content-pane, of class-type `java.awt.Container` (see page 566). Here are given three simple examples for drawing graphical-objects: Text, rectangles or other polygons in Java. As done before, the handling of screen-components is anticipated; the three examples here present variants of introducing drawing-surfaces within Java source-code:

- First the drawing-surface is given by an object derived from a class which extends one of the classes `javax.swing.JComponent` or `java.awt.Component`.
- The second case produces the drawing-surface in the same way, except that the class of the drawing-surface itself contains the `main()`-method which controls the entire process.
- In the third case the drawing-surface is given as an object instantiated from an anonymous inner class.

The first program draws a line and a string: (The screen-component, to be found on the desktop, is instantiated within the `main()`-method as an instance of the `JFrame`-class, sized and set to be visible.) The actual drawing-surface is a screen-component of its own, for which a customized `paintComponent()`-method contains the commands producing the graphical-objects: A line and a string.

```
import java.awt.Graphics;      // toolbox furnishes graphical-objects
import javax.swing.JComponent; // used as drawing-surface
import javax.swing.JFrame;     // screen-area for display
import java.awt.Component;     // (for TESTING an AWT-version of this program)

class HelloObjectDraw
{
    public static void main( String unused[] )
    { JFrame screenArea = new JFrame( "Title" );
      screenArea.getContentPane().add( new DrawingSurface() );
        // (TEST omit .getContentPane())
      screenArea.setSize( 200, 60 );
      screenArea.setVisible( true );
    }
}
class DrawingSurface extends JComponent
{ // (TEST works the same with java.awt.Component)
  //(TEST use paint() with Component instead of ---- paintComponent() with JComponent)
  public void paintComponent( Graphics tools )
  { tools.drawLine( 10, 10, 30, 30 );
    tools.drawString( "Painted Hello!", 30, 30 );
    System.out.println
      ( "paint in event-dispatch-thread?" +
        javax.swing.SwingUtilities.isEventDispatchThread()
      );
  }
  //(The CHECK above, is treated in the context of event-handling beginning on page 586.)
}
/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > javac HelloObjectDraw.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > java HelloObjectDraw
paint in event-dispatch-thread?true (window resizing with mouse triggers automatic repainting)
paint in event-dispatch-thread?true
paint in event-dispatch-thread?true
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput >
*/
```



Who calls the Swing-specific `paintComponent()`-method? The `paintComponent()`-method is called by the `paint()`-method which itself is called by the Java Runtime Environment (JRE) each time the screen-management feels that it is necessary to do so. Thus the pair `paint()` and `paintComponent()` is given by triggerable methods. (These methods are triggered by a call from the JRE, whereas the `main()`-method is triggered by a terminal-command of the user.) More on these “paint”-methods can be found in the table beginning with page 559.

Next, the program is given as a class that extends the drawing-surface, this class is instantiated in the `main()`-method. An oval and a string are drawn.

```
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;

class HelloObjectDrawExtending extends JComponent
{
    public void paintComponent( Graphics tools )
    { tools.drawOval( 30, 0, 60, 100 );
      tools.drawString( "Painted Hello!", 30, 30 );
    }
    public static void main( String unused[] )
    { JFrame screenArea = new JFrame("Title");
      screenArea.getContentPane().add( new HelloObjectDrawExtending() );
      screenArea.setSize(200, 100);
      screenArea.setVisible(true);
    } }
/*
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% javac HelloObjectDrawExtending.java
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% java HelloObjectDrawExtending
^C (Ctrl-C)
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway%
*/
```



Finally, the drawing-surface is given as an anonymous inner class of the source-code. A rectangle and a string are drawn.

```
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;

class HelloObjectDrawInner
{
    public static void main( String unused[] )
    {
        JComponent drawingSurface = new JComponent()
        {
            public void paintComponent( Graphics tools )
            { tools.fillRect( 30, 30, 100, 20 );
              tools.drawString( "Painted Hello!", 30, 30 );
            }
        };
        JFrame screenArea = new JFrame( "Title" );
        screenArea.getContentPane().add( drawingSurface );
        screenArea.setSize(150, 100);
        screenArea.setVisible(true);
    } }
/*
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% javac HelloObjectDrawInner.java
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% java HelloObjectDrawInner
(Ctrl-C)
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway%
*/
```



Bitmaps and graphical-objects now have been introduced. Within the Java2D context (mostly found in the package `java.awt.image`) and the Java Advanced Imaging context (package `javax.jai`), the Java language furnishes a wide range of further methodologies for producing and transforming pictures. Here, within the context of describing the technical side of single-user-interfacing, imaging represents only a necessary part of the introductory consideration.

Java: Painting-Methods Fixing the View of Screen-Components

Java: Painting-Methods Fixing the View of Screen-Components — Before considering screen-components in general, remember their introduction around page 510, and reconsider their use as desktop- and drawing-surfaces as demonstrated in the previous pieces of Java source-code. The `paintComponent()`-method of a Swing-component (or the `paint()`-method of an AWT-screen-component) contains or calls all customized imaging capabilities of the screen-component, as defined by the Java programmer. The view (=graphical appearance on-screen) of that screen-component can be changed further, only by extending the given class and by overriding the method `paintComponent()` (AWT, `paint()`) thus defining a new class and new types of screen-components. Therefore, keep in mind, that THE STANDARD GRAPHICAL REPRESENTATIONS DEFINED IN THE `paintComponent()`-METHOD (AWT, `paint()`) CHARACTERIZES THE ON-SCREEN APPEARANCE OF THE SCREEN-COMPONENT. If the `paintComponent()`-method (`paint()` in AWT-components) lacks the flexibility of allowing a change of the graphics displayed, then a programmer may substitute the screen-component by a screen-component of another type: For example, a label-object “OK” may be exchanged for a label-object printing out “Running”.

The painting-methods of screen-components fix their on-screen appearance

Concept: Screen-Components

Concept: Screen-Components — Remember that sequences of bytes which *pixelwise* encode given image-information are called bitmaps. Image-information also can be given in an encoded form (compressed or given as mathematical formulae) so that producing displayable or printable bitmaps requires a transformation process (rendering). For example, elementary geometrical constructs (rectangles, ovals, ...) or character-fonts may be given by sets of parameterized mathematical formulae, those graphical entities are also called graphical-objects. Commands of some language (Java) and/or commands taken from a toolbox (`java.awt.Graphics`, `java.awt.Graphics2D`), may render such image-information to be displayed.

Screen-components use images, but screen-components are much more than simple images: Everyday life with a Graphical User Interface (GUI) lets end-users encounter - even as part of one screen-component - different graphical-objects with different functionalities and different relations: A label, a scrollbar, a button (a “key” on the screen that can be “clicked with the mouse”), editable text-fields or entire windows that contain editable text-fields, planes for drawing graphics, control-panels containing buttons, a checkbox to be activated by a “mouse-click” and to be deactivated by another one, an alert-window opening and blocking the computer until the desired choice is made (also called modal screen-component, sometimes called dialog-window), the window of an application allowing to work on an image inside, by using commands choosable from a menubar, and other graphical-objects with enhanced functionalities of that kind. Virtually each piece of Java source-code of the following sections is accompanied by a printed image showing the related screen-component.

Graphical-objects having the capability to interact with the end-user are more than just images, for images are displayable graphical-objects and nothing more. What is the difference between an image and a **(screen-) component** of the kind above? Remember also the first characterization of screen-components in this text on page 510.

- (image) A screen-component uses **an image or images as a part of its on-screen representation**. A screen-component may be visible in different ways using its differing images. For example, a label uses a text, a checkbox needs a checked image and an unchecked image-version.
- (input, interaction and communication with end-user) A screen-component **may take input from mouse and keyboard**. The screen-component can exist in different states (activated=on focus, no focus, disabled) for interaction, see page 509. An enabled "OK"-button can register mouse-clicks, or a text-field can be made sensitive to keyboard-input. Screen-components are allowed to bear an entire message-infrastructure, for communication with an end-user.
- (containment) A screen-component **may contain other screen-components, giving a containment-hierarchy**. Dialog boxes may contain text labels and "OK"- and "Cancel"-buttons. This example already shows a simple containment-hierarchy.

The roots of such containment-hierarchies are windows on the operating-system's desktop. Programmers write code that determines how to make windows visible on-screen and how to enable those windows to take inputs. End-users have to make a window ready for input (from keyboard or mouse) by clicking on it or by moving the mouse-pointer over it. This is also called **activating the window**. (page 509)

Now, the previously mentioned characteristics (image, communication, containment-hierarchy) of screen-components are refined. In Java, pre-written classes let the programmer instantiate objects, the screen-components. Software-libraries (`java.awt`, `javax.swing`) furnish these pre-written component-classes, which can sort their methods according to the following classification:

Java's screen-components are objects

- A screen-component has to be displayable on the screen, what can be seen is also called "view" of the screen-component. Often screen-components appear as part of another screen-component, one view being embedded in another view; then the embedded view only should be visible if the embedding view is visible: For example, a checkbox inside some control-panel can be seen only, if the control-panel, in which it is located, is visible on the screen. (**Display**)
- A screen-component, or rather the view of a screen-component, takes a specific place on the screen. (**Position/Size**)
- The screen-component may use colors, often system-specific ones; furthermore, if containing text, the screen-component may use a font. (**Appearance**) In some cases the appearance should depend on the cultural context of the user, and in other cases the appearance changes due to time-delayed data-loading-processes, for example with images and other multimedia-files loaded from a network. (**Appearance-time-delayed**)
- Many screen-components can interact with the user through mouse-clicks and keyboard-inputs or screen-components help to organize printing and network-access. (**I/O-state/Focus/Listening**)
- Another striking observation indicates that many screen-components, for example the one representing a control-panel, contain other screen-components, for example those representing checkboxes and labels. (**Relative Positioning/Grouping/Layout**) This containment-relation can lead to an outright containment-hierarchy (**Containment-Hierarchy**).

The word (screen-)“component” describes a general functional part of the Graphical User Interface (GUI) and bundles the basic ideas about graphical display and interaction with keyboard and mouse-events. To get a better overview of the diversity described above, the properties have been regrouped again:

- (Display) Is the component **ready for display**? Has the computer-platform been notified to draw the (view of the) component onto the screen? An affirmative answer to such a kind of question is also restated in Java-related words as “The screen-component is **visible**.” Can the component be seen by the end-user, even if it is necessary to remove overlaying screen-components by dragging or window-closing mouse-clicks? Can the end-user find the screen-component on the screen? A “Yes” can be restated in Java-related words as “The component is **showing**!” — If the container-component vanishes from the screen, so do the components inside the container (-component). So, one may discern between screen-components that are - *nonvisible*, - *visible* and - *showing*. Components that are set **nonvisible** never appear on the screen; whereas **visible** components are ready to appear on-screen or have been triggered into the computational process that makes them appear on the screen. Some visible screen-components, like labels or buttons appear on-screen only if their container appears on the screen (is visible and showing). If the top-level container is set visible, then the process is started to make that top-level container and all other contained components appear on-screen (showing) so that the end-user can find them. (See also the gray box on page 598)

(Position/Size) Where is a screen-component, or rather its graphical-representation, **positioned** and **sized** on the screen. If the screen-component exists within another screen-component, like a label within a frame, then the position may be fixed relative to the coordinate-system of the enclosing frame.

(Appearance) Screen-components use (possibly system-specific) **colors**, sometimes different ones in foreground and background, the text is presented in specific **fonts** and sizes; time- or color-information depend on the **locale** the computer is used in. Even the **cursor** may take different forms. (In some cases, a programmer may consider it useful to let the end-user manually update the display, especially when the end-user is allowed to paint on the component.)

(Appearance-Time-Delayed) **Resource-consuming**²⁴⁴ **images** are the standard example of cases where the screen-component gets some of its information successively and possibly so slow to let the user experience a perceptible waiting-time. Since the use of images is so widespread, screen-components may be given some special methods to handle the loading and the displaying of images. This may include methods which let the end-user see the loading-process by drawing everything that has been loaded directly onto the screen. Or by loading "in the background" and drawing the whole image at once onto the screen. (double buffering, meaning off-screen rendering) While the background-loading proceeds, the end-user may get informed about how many percent of the loading already has taken place.

(I/O-state/Focus/Listening) A screen-component may take mouse-clicks and keyboard-inputs, also called **events**, and use them or let them be of use for further processing. The programmer can permit or forbid end-user interaction with the screen-component by **enabling or disabling the screen-component**. If the screen-component is ready to interact with the end-user, then the **component is said to be activated or on focus**. The term "activated" is used primarily for frames and windows which are showing directly on the screen's desktop, whereas checkboxes or text-fields (within some activated window), which can take input, are called to be "on focus".

Can the screen-component be given focus by a mouse-click or is it blocked until some other window is closed or worked through? Screen-components capable of blocking all other screen-interactions are called *modal screen-components*.

A screen-component can do its own processing or delegate the processing to so-called handler-methods located in other parts of the programming-environment; the latter may be part of source-code-structuring reflecting the Model View Controller concept (MVC) or the Model-Delegate (front-end/kernel) concept.

(Reflection) A screen-component may be given a name by the programmer. The screen-component may **yield information** about its name, position/size, appearance and any other aspects of its state. Hereby a screen-component may make use of conventions or functionalities of its Run-time Environment: An example gives the Java-Beans-model in Java.

A checkbox may represent a sort of an *elementary*, non-container-like, screen-component, because it should not sensibly contain any kind of other screen-components. A button may sensibly only contain a label or an icon. Some of the examples above, like windows and control-panels, are screen-components meant to contain other screen-components, such screen-components are also called **(screen-)containers**.

An example of screen-components of the container-kind and of the elementary-kind, is given by the browser-window on page 370: The window contains a toolbar, a container itself containing several buttons, which are rather elementary screen-components. The window also contains a menubar, which itself contains menus that are currently not showing, but if needed they can be made showing, by a user's mouse-click! This example gives an idea of an entire containment hierarchy, involved in producing such a working browser-window.

So, a (screen-)container can be seen as an extension of a screen-component, characterized by the ability to add and to remove other screen-components. A screen-component is also a container if it has some additional properties:

(List of Children/Layout) First there may be given an - often numbered - list of "child"-components held by the container. Then there may be found methods allowing to add and to remove individual components from that list.

Grouping multiple screen-components inside a container can be done by using usually pixel-coordinates, this is called **absolute positioning**. Another, more comfortable way of position-

enabling or
disabling the
screen-
component

modal screen-
component

container

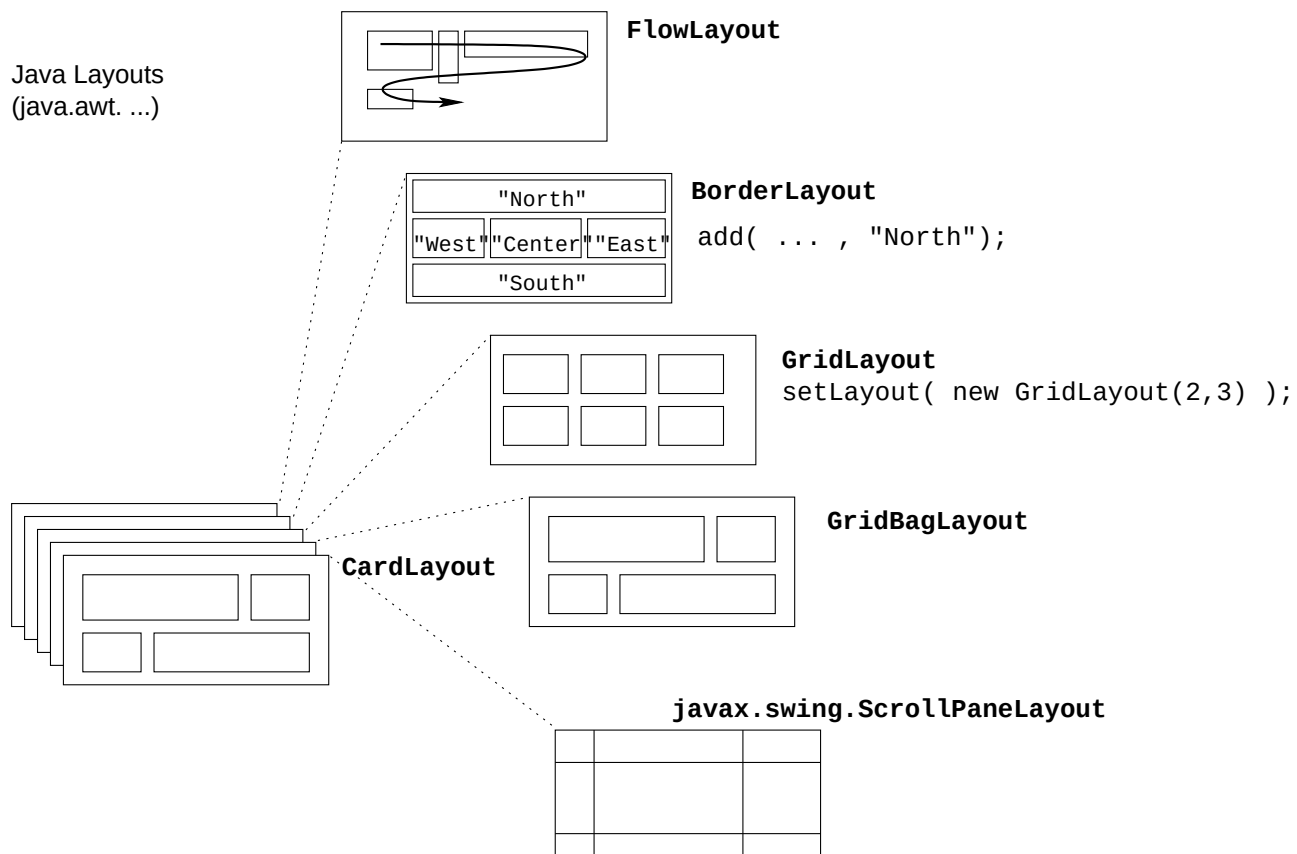
absolute
positioning or
layout-managing

²⁴⁴Resource-consuming means for example: — Needing a lot of memory. — Needing a lot of CPU-time to be rendered. — Being large enough to cause delays in network-transfer that are noticeable by the end-user.

ing child-components inside a container may be possible by using so-called **layout-managers**. Layout-managers implement a scheme for automatically grouping the child-components inside a container. Examples for layout managing are given in the graphic below.

(Containment-Hierarchy) Containers may contain other containers which lets emerge an entire **hierarchy** of containment relations, especially parent-child-grandchild relations.

In the graphic below are sketched the main functional ideas of layout-managers. Such layout-managers automatically determine the sizes of the contained-objects and compute the appropriate arrangement according to their given layout-strategy:



Since containers may contain containers, some screen-components may exhibit a full-fledged **containment-hierarchy** like the one already exemplified with the browser-window above.

containment-
hierarchy

The container that contains a component is also called **parent of that component**. Any component higher in the containment-hierarchy is called **ancestor**. A component inside a container is called **child** of that container. Any components lower in the containment-hierarchy are called **descendants of the referenced container**.

Compare this to the containment-hierarchy of files in a Hierarchical File System (graphic on page 47) and the extension-hierarchy (class-hierarchy) of classes in the object-oriented approach (graphic on page 135). This hints at the importance of tree-like hierarchical structures in general application-programming (Text-editors usually organize their text-model (following the Model View Controller concept) using a tree-structure).

Now it becomes possible to give the specifications of the Java screen-components (in their `java.awt` and `javax.swing` variants) in a more systematical way. For an approach that stresses the basic features of the libraries `java.awt` or `javax.swing` continue with the section about heavyweight screen-components on page 548. The next section presents examples of basic fully-functional screen-components:

Java: Basic Swing Screen-Components

Java: Basic Swing Screen-Components — To get a more direct impression of concrete user-interfacing, the following section presents prototype programs anticipating some basic concepts (Frames for example are introduced around page 561). Thereby is given an introduction to pre-written screen-components – just by presenting some concrete examples:

The following program puts a label inside a frame. Setting size and visibility puts the frame-component on the screen:

```
import javax.swing.JFrame;
import javax.swing.JLabel;

class LabelDemo
{
    public static void main( String unused[] )
    {
        JFrame aFrame = new JFrame( "title" );
        JLabel aLabel = new JLabel( "This is the text of the label." );
        aFrame.getContentPane().add( aLabel );
        aFrame.setSize(300,100);
        aFrame.setVisible(true);
        aFrame.setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE );
        // DISPOSE_ON_CLOSE, HIDE_ON_CLOSE from the javax.swing.WindowConstants interface,
        // implemented by javax.swing.JFrame.
    }
}
/*
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% javac LabelDemo.java
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% java LabelDemo
^C (Ctrl-C)
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway%
*/
```

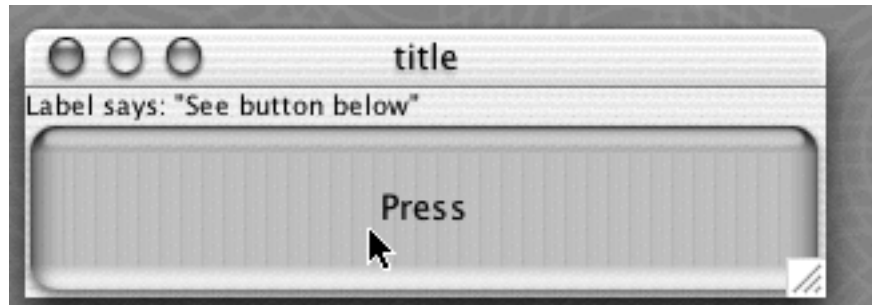


The next program adds a JButton-component to the previous setting. If the end-user clicks this button with the mouse, then – since action-listeners (here derived from a class “AButtonActionListener”) have been registered – the Java Runtime Environment (JRE) instantiates an event-object (of class-type given by the identifier java.awt.event.ActionEvent) and automatically forwards references of that event-object to the handler-methods (actionPerformed()) of the registered listeners. For more on event-handling, consider the text around the graphic on page 573.)

```
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.event.ActionListener; // interface,
// signaling the implementation of an actionPerformed()-method.
import java.awt.event.ActionEvent;

public class ButtonDemo
{
    public static void main( String unused[] ) throws Exception
    {
        JFrame aFrame = new JFrame( "title" );
        JLabel aLabel = new JLabel( "Label says: \"See button below\"" );
        JButton aButton = new JButton( "Press" );
        aFrame.getContentPane().add( aLabel , "North" ); // border-layout is default
        aFrame.getContentPane().add( aButton , "Center" );
        aFrame.setSize(300,100);
        aFrame.setVisible(true);
        ActionListener al = new AButtonActionListener(); // observe the implicit type-cast!
        aButton.addActionListener( al );
    }
}
class AButtonActionListener implements ActionListener
{
    public void actionPerformed( ActionEvent e )
    {
        System.out.println("button has been pressed");
    }
}
/*
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% javac ButtonDemo.java
```

```
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% java ButtonDemo
button has been pressed
button has been pressed
button has been pressed
button has been pressed
^C (Ctrl-C)
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway%
*/
```



The next program uses the data of an array presented to the end-user as a list. Here the class with the `main()`-method is reused as the window-screen-component containing the list-component (same principle as applied in the program on page 539). Thus the filling of the content-pane, the sizing and the setting of the visibility, can be done within the constructor of the top-level screen-component. Here too, a listener-class registers with the list-component; this enables the end-user to select list entries and to communicate the selection to the Java program.

```
import javax.swing.JComponent;
import javax.swing.JWindow;
import javax.swing.JList;
import javax.swing.JScrollPane;
import javax.swing.event.ListSelectionListener; // interface,
// signaling the implementation of a valueChanged()-method.
import javax.swing.event.ListSelectionEvent;

class ListDemo extends JWindow
{
    public static void main( String data[] )
    {
        if( data.length == 0 ) data = new String[]{ "0", "1", "2" };
        JList alist = new JList( data );
        alist.addListSelectionListener( new SomeListSelectionListener() );
        JScrollPane aScrollPane = new JScrollPane( alist );
        new ListDemo( aScrollPane );
    }
    ListDemo( JComponent display )
    {
        getContentPane().add( display );
        setSize( 200, 100 );
        setVisible( true );
    }
}

class SomeListSelectionListener implements ListSelectionListener
{
    public void valueChanged( ListSelectionEvent e )
    { if( e.getValueIsAdjusting() == false ) return; // nothing new
      Object[] selected = ( (JList)e.getSource() ).getSelectedValues();
      for( int i = 0; i < selected.length; i++ ) System.out.println( selected[i] );
    }
}
/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > javac ListDemo.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > java ListDemo ONE TWO THREE
THREE
TWO
(Ctrl-C in terminal-window)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput >
*/
```



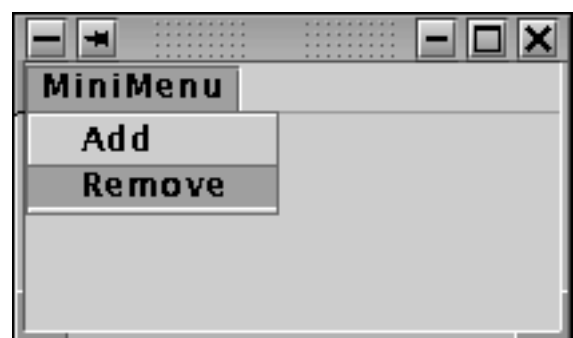
Menus are a variant of an usually two-dimensional list. Unless menu-items are explicitly disabled, menus and sub-menus contain single items which are to be selected by the end-user. Menus often present structured collections of commands which thus are available to the end-user.

Below just one class acts as a top-level screen-component (extends `JFrame`), as a container holding a menubar (`getRootPane().setMenuBar(...)`), as an event-listener (implements `ActionListener`) implementing a handler-method (`actionPerformed()`) and as a program (`main()`-method) which thus offers a `main()`-method, externally triggerable by issuing a `java`-command inside some terminal-window:

```
import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import java.awt.event.ActionListener; // interface,
// signaling the implementation of an actionPerformed()-method.
import java.awt.event.ActionEvent;

class MenuDemo extends JFrame implements ActionListener
{
    public static void main( String unused[] )           // main()-method
    { new MenuDemo(); }

    MenuDemo()                                           // constructor
    {
        JMenuItem menuIncrement = new JMenuItem( "Add" );
        JMenuItem menuDecrement = new JMenuItem( "Remove" );
        menuIncrement.addActionListener( this );
        menuDecrement.addActionListener( this );
        JMenu aMenu = new JMenu( "MiniMenu" );
        aMenu.add( menuIncrement );
        aMenu.add( menuDecrement );
        JMenuBar aMenuBar = new JMenuBar();
        aMenuBar.add( aMenu ); // or more menus
        // getContentPane().add( display ); // in Swing, different for menus (page 566):
        getContentPane().setJMenuBar( aMenuBar );
        setSize( 200, 100 );
        setVisible( true );
    }
    public void actionPerformed((ActionEvent e) )
    { System.out.println( ((JMenuItem)e.getSource()).getText() ); }
}
/*
williammartin@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > javac MenuDemo.java
williammartin@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > java MenuDemo
Remove
(Ctrl-C)
williammartin@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput >
*/
```



Finally, there is presented a demonstration of a progress-bar coupled to a slider via a registered event-listener. Any change, which the end-user produces by dragging the rider of the slider, reaches the handler-method (`stateChanged()`) of the listener (the “SliderProgressBarDemo”-class itself) and triggers the (visible) change of the progress-bar.

```
import javax.swing.JWindow;
import javax.swing.JSlider;
import javax.swing.JProgressBar;
import javax.swing.event.ChangeListener; // interface,
// signaling the implementation of a stateChanged()-method.
import javax.swing.event.ChangeEvent;

class SliderProgressBarDemo extends JWindow implements ChangeListener
{
    int min = 0, max = 100, init = 0;
    JSlider slide = new JSlider( JSlider.HORIZONTAL, min, max, init );
    JProgressBar progress = new JProgressBar();

    public static void main( String unused[] )
    { new SliderProgressBarDemo();
    }
    SliderProgressBarDemo()
    {
        slide.addChangeListener( this );
        // Default layout is BorderLayout.
        getContentPane().add( slide, "North" ); // different for menus
        getContentPane().add( progress, "South" ); // different for menus
        setSize(200,100);
        setVisible(true);
    }
    public void stateChanged( ChangeEvent e )
    { progress.setValue( slide.getValue() );
    }
}
/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > javac SliderProgressBarDemo.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > java SliderProgressBarDemo
(Ctrl-C in terminal-window)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput >
*/
```



The programs above are demonstrations of screen-components readily furnished in packages (`javax.swing`, `javax.swing.event`, `java.awt`, `java.awt.event`) together with the Java language. The related event-handling will be scrutinized a few sections later (especially beginning with page 571). This understanding of screen-components together with the event-handling mechanism, as demonstrated above, makes up the core of single-user-interfacing.

All these programs in this section used classes with a first letter “J” which denotes the type of “Swing”-components. The “Swing”-type of screen-components gives the end-user more service (tooltip, look and feel) than another, older type of screen-components: The “AWT”-type of screen-component. The programs above can be rewritten to use the “AWT”-type of screen-component by:

1. Omitting the “J”.
2. Omitting the method-call `.getContentPane()`.
3. Substituting `javax.swing` by `java.awt`, especially in the `import`-statements.
4. a) “LabelDemo” ready. 4. b) “ButtonDemo” ready. 4. c) In the “ListDemo”-class substitute `ListSelectionListener` by `ItemListener`, `ListSelectionEvent` by `ItemEvent` and `valueChanged()` by `itemStateChanged()`, finally within the so-called handler-method omit the first line and substitute

`getSelectedValues()` by `getSelectedObjects()`. 4. d) “MenuDemo” ready. 4. e) In the “SliderProgressBarDemo” substitute `JSlider` AND `JProgressBar` by `Scrollbar` and change the parameter-list of the non-empty `Scrollbar`-constructor to `(Scrollbar.HORIZONTAL, init, max-min, min, max)`, for that of the empty one use `(Scrollbar.HORIZONTAL)`. This produces the programs with “AWT”-equivalent screen-components.

Lightweight-/Heavyweight Components — Java: AWT versus Swing

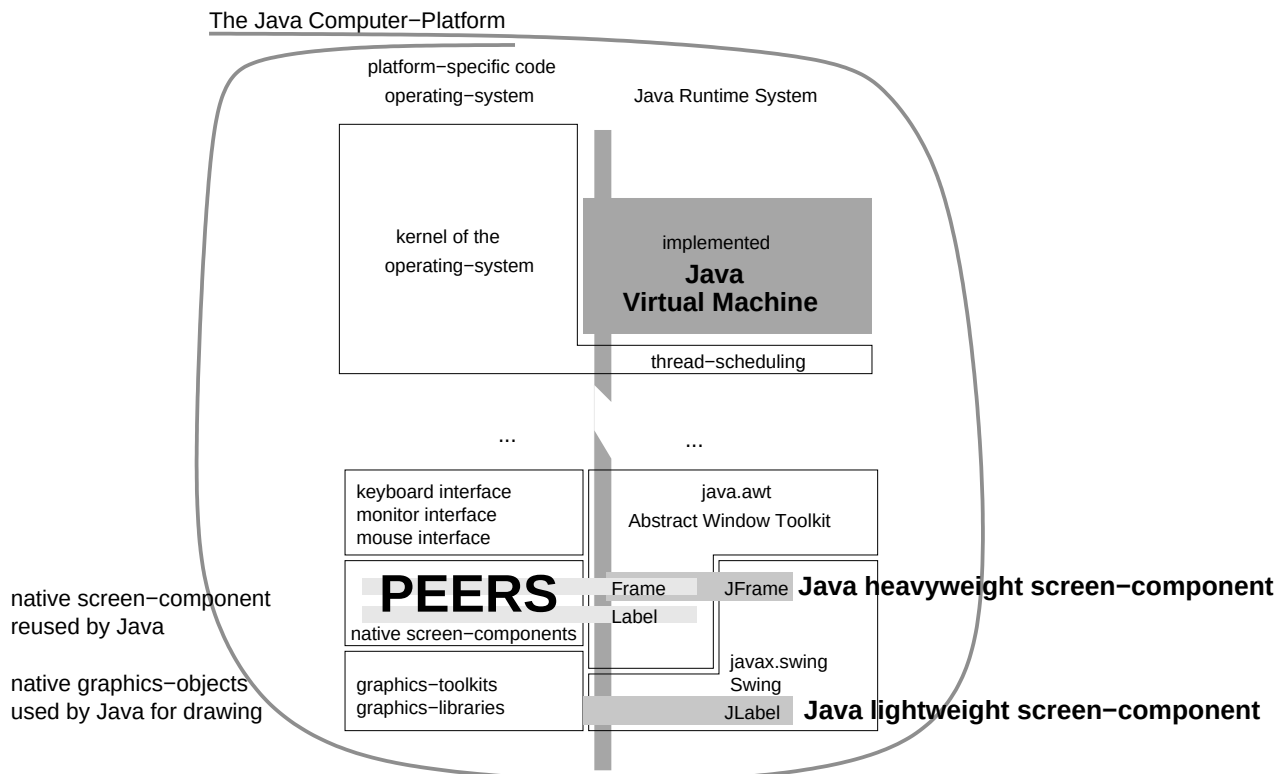
Lightweight-/Heavyweight Components — Java: AWT versus Swing — Painting images or components directly on the screen in the context of an operating-system means that the screen-drawing-resources of the operating-system have to be used. This already has been indicated in the graphic on page 53. In the vicinity of that graphic, the text describes how the Java Runtime Environment (JRE) runs platform-independent code. There already has been indicated what has been exemplified in the previous section(s): An integral part of the JRE furnishes channels of communication between the native platform and the Java program, this includes keyboard- and mouse-communication as well as furnishing a way to put graphical information on the computer’s screen, not to forget Internet-sockets and other Input/Output-services.

Written in native code for differing computing-platforms, thus implemented within differing operating-systems, the Java Runtime System (JRS) offers a platform-independent computing-environment striving to keep the Java-code fully-portable. This includes keeping on-screen results of drawing-commands as predictable as possible. The differences of the operating-systems in graphics- and event-management conflict with the necessity for Java to consistently receive all platform-events and to draw uniformly on-screen. Therefore, a properly chosen interface is needed:

One strategy (**Java’s AWT approach**) attempts to reuse the platform-specific screen-components, their containment-hierarchy and event-infrastructure. This requires the system-programmers who create the Java Runtime System to associate `Checkbox`-objects of the Java Runtime System with checkboxes of the native platform and so on. The associated native screen-components are called **peers**²⁴⁵, these are eventually seen on-screen controlled by the Java Runtime Environment. Ideally in that way the Java programmer’s source-code remains platform-independent. But as experience shows, the graphical output proves to be, not only platform-dependent, but also difficult to use. Java screen-components which reuse screen-component-software of the native operating-system (this includes basic event-management), are called **heavyweight screen-components** or **peer-based components**.

Remember the Java computer-platform as presented on page 53. Heavyweight components can be clearly visualized in that context:

²⁴⁵AWT: The Java Runtime Environment internally uses references of abstract type `java.awt.Toolkit` reusing native code and Java interfaces, named `java.awt.peer.InterfaceNamePeer`, to directly access associated native screen-components.

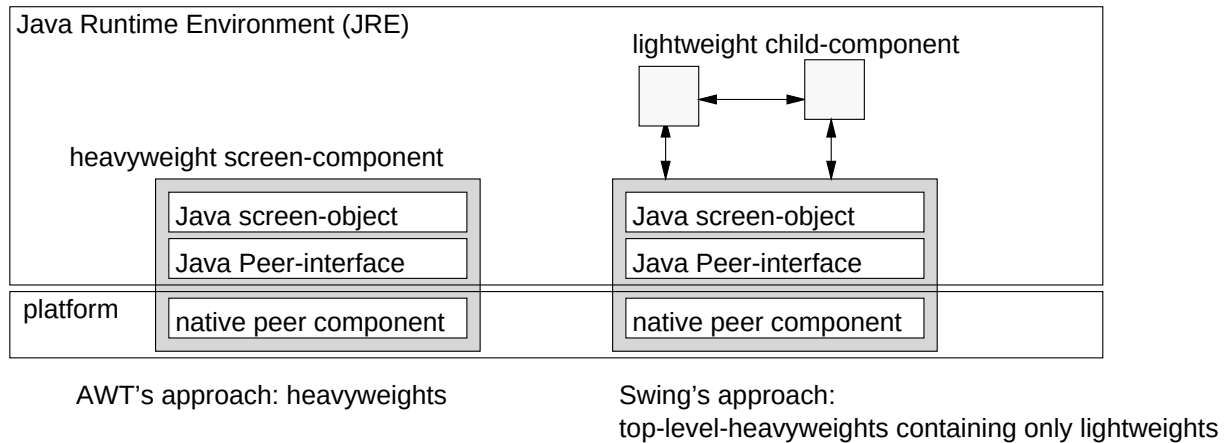


The other strategy (**Java's Swing approach**) means to rely as much as possible on the Java Runtime System's own graphic- and event-management resources and to keep the necessary contact with the underlying platform to a minimum. This means painting the screen-components with elementary graphics-commands of the native operating-system and getting event-information unprocessed by the native platform. So, the Java Runtime System takes over many services originally provided by the Graphical User Interface of the native operating-system. Apart from using Java's platform-independent source-code, this also guarantees a better control of output-representation. The Swing-idea requires rewriting the Java screen-components purely in Java, nevertheless the background, on which they are painted, is given by only a few Java heavyweight components (JFrame, JWindow, JDialog and JApplet)²⁴⁶. Screen-components WITHOUT a native counterpart (peer) of the same function, are called **lightweight screen-components**. In the standard Java Runtime Environment, painting on-screen is possible only within heavyweight containers²⁴⁷. So except for the previously mentioned four heavyweight components, all other Swing components are lightweight and painted on these heavyweights. Lightweight components in a (Java) Runtime System use the drawing resources of a heavyweight container and its event-management, so that native drawing- and event-management is kept free of using additional native components. Especially lightweight components can use the Java Runtime System's event-management-scheme to communicate with each other (this makes event-handling on this level also platform-independent). Except for basic drawing and hardware-related event-communication, Java Swing and the Java Runtime Environment handle all the tasks of the operating-system's GUI.

²⁴⁶The first three heavyweight classes (JFrame, JWindow, JDialog) produce objects with views which can appear directly on the desktop of the operating-system. The last one (JApplet) produces objects embedded in HTML-text-files viewable in browser-windows or windows generated by appletviewers.

Embedded here means: HTML-text-files contain references to Java applet class-files. On displaying the HTML-text-file, formatted in the browser-window, this reference is followed automatically by the browser, thereby establishing an Applet Runtime Environment. Within this environment the browser triggers the instantiation of the applet-object as a screen-component; finally this screen-component is shown on-screen in the reserved space of the formatted HTML-text.

²⁴⁷With Java Version 1.4 the "full-screen exclusive mode API" allows a programmer to paint directly on-screen.



In Java, lightweight screen-components can take any shape within the boundary of the heavyweight background. This is possible because lightweight components may have a graphical representation with transparent pixels and thus let the background shine through. Where supported (Java Version 1.2 and higher), color and partial transparency can be set by the class `java.awt.Color` and a method `javax.swing.JComponent.setOpaque(true/false)`.

In Java, heavyweight screen-components always are opaque and the top-level ones give the graphical background for any components they contain. **CAUTION:** Mixing heavyweight and lightweight components (within one common heavyweight background) and letting them overlap makes the lightweight component being overlapped by the heavyweight.

How to discern heavyweight components from lightweight components? — *Non-abstract AWT*-screen-components are heavyweight. Class-extension preserves the state of being a heavyweight component. But the class of a lightweight-component can be extended to a class of a heavyweight-component. (For example see, the diagram of class-hierarchy on page 564.) All Swing-components are lightweight EXCEPT those that are derived from AWT-components: `JFrame`, `JApplet`, `JWindow`, `JDialog`. Then, as a last resort to identify a screen-component as light- or heavyweight, there are still the methods

```
java.awt.Component.isLightweight() // and
javax.swing.JComponent.isLightweightComponent()
```

Swing → AWT which return a boolean answer.

Changing Java programs from using Swing screen-components to using AWT screen-components requires steps listed as three points below:

1. Substituting `javax.swing.J...` with `java.awt....`,
2. omitting the `getContentPane()`-method-call and
3. substituting the `paintComponent()`-declaration by a simple `paint()`-declaration

Additional changes, as those described at the end of the previous section, transform these programs into pure AWT-programs which then generally use only heavyweight components.

Heavyweight-Code: 1. In some computing-environment with a layer-model, heavyweight-code represents a mere interface to the same functionalities furnished by a lower-level. Heavyweight-code directly reuses or adapts lower-level code of the same kind. The lower-level counterpart of a heavyweight piece of code is also called peer-code.

2. (Java) Basically the same as above. Heavyweighted Java-code in a Java Runtime Environment has mapped most of its methods directly to native methods on the current computer-platform. Unseen by the application-programmer, a Java heavyweight screen-component communicates with the associated native screen-component via a Java interface `InterfaceNamePeer` to be found in the `java.awt.peer` package.

Lightweight-Code: 1. In some computing-environment with a layer-model, lightweight-code realizes its functions on its own level as far as possible and only uses elementary functionalities furnished by a lower-level out of necessity. Lightweight-code tries to attain maximal independence from lower-level code. Only if unavoidable, lightweight-code uses few and elementary functionalities of lower-level-code.

2. (Java) Java-code called lightweight follows the guidelines described under Point 1.

Observe that on the level of drawing graphical-objects onto the screen, Java uses references of abstract²⁴⁸ type `java.awt.Graphics` to give the Java programmer direct access to native libraries that furnish graphical functionalities, as shown in the next section. This fixes about the level on which lightweight Java screen-components have to be constructed.

Platform and Java Runtime Points-of-Contact

Platform and Java Runtime Points-of-Contact — Several packages and classes support the management of screen-components in Swing and AWT. Among others, note especially the classes `java.awt.Graphics`, `java.awt.Graphics2D`, `java.awt.GraphicsEnvironment` (page 569), `java.awt.print.PrinterJob` and the utility-class `java.awt.Toolkit`. These classes are declared abstract because they are not implemented under their original name as a file of Java byte-code. So, for these classes there exists no file of Java byte-code under the original class-name suffixed by `.class`. Instead, these classes interact directly with the platform's operating-system, they are implemented by platform-specific code (usually as a C++ program compiled for the native platform). For example, the class `java.awt.Toolkit` is implemented — depending on the platform — as:

```
sun.awt.win32.MToolkit (Java1.0)
sun.awt.windows.MToolkit (Java1.1 and Windows95/NT)
sun.awt.motif.MToolkit (Solaris/Motif)
sun.awt.macos.MToolkit (Apple Mac OS)
```

A Java programmer obtains a reference of class-type `Toolkit`, pointing to this non-Java code, by calling the factory-like-method `java.awt.Toolkit.getDefaultToolkit()` or the instance-method of the same kind `java.awt.Component.getToolkit()`. Since the implementation of these classes is not given by Java byte-code within the Java Runtime Environment (JRE), these classes have been declared abstract! Therefore, references to code accessible by these classes cannot be produced by constructors: In this special case these methods are called **factory-like-methods**, THE REFERENCES THEY RETURN factory-like-method POINT TO A NATIVE ENTITY, NOT TO A JAVA OBJECT within the Java Runtime Environment. (Remember also the remarks on this use of the modifier `abstract`, found on page 184 and on page 527.)

Then, why are heavyweight screen-components with native peers declared non-abstract? Because they can be compiled to byte-code, but this byte-code uses references of an associated abstract interface-type, thus accessing the native peer-object, as this has been depicted in the graphic of the previous section.

Java: Basics of Screen-Components

Java: Basics of Screen-Components — This text does not present all the details of core Java packages, those details often depend on the current Java version and may change. For more extensive and up-to-date information consult more specialized sources, like the ➡☐ ➡☐ Java Platform API Specification [JPAPIS] or the appropriate handbooks. This section here focuses on presenting the main ideas that govern the use of the classes given by the Java packages AWT (`java.awt`) and Swing (`javax.swing`).

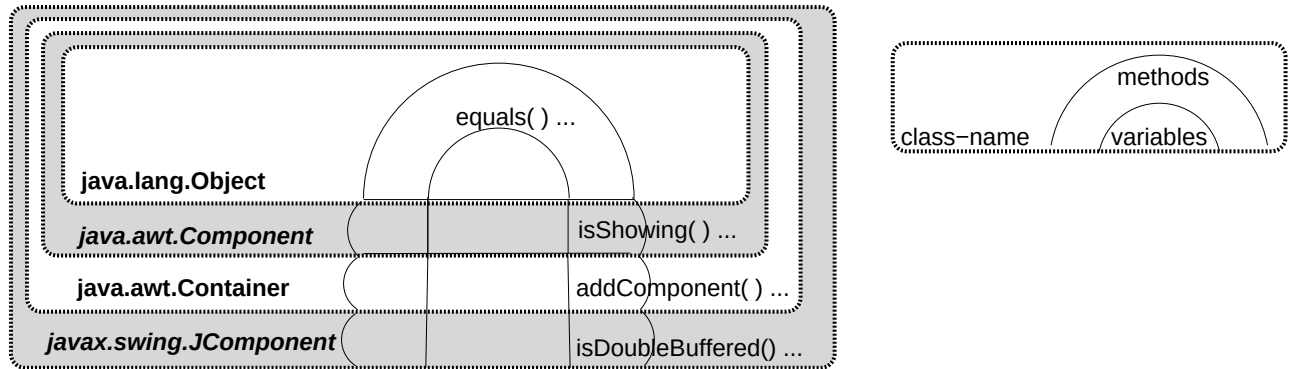
The central features of all Java screen-components are bundled in three basic classes: `java.awt.Component`, `java.awt.Container` and `javax.swing.JComponent`.

- The class `java.awt.Component` extends the class `java.lang.Object` and encapsulates most of the services which have been mentioned in the general discussion of screen-components beginning on page 540. The entire delegation event-infrastructure in Java's single-user-interface programming comes from the `Component`-class (for more on the event-handling see page 571.).
- The class `java.awt.Container` is a direct extension of `Component`. A `Container`-object represents a screen-component fillable with sub-components, also called child-components, (labels, buttons, lists and more, also containers) and allows to add or remove those child-components. The positioning of the child-components within the `Container`-screen-component is done by setting a layout-manager.
- The class `javax.swing.JComponent` directly extends `java.awt.Container` with Swing-specific properties, like automatic double buffering or a revised painting-mechanism (makes the programmer paint on a content-pane and makes the programmer override the method `paintComponent()` instead of the `paint()`-method).

²⁴⁸ See page 184 for references of abstract type to native code.

AWT-classes, `java.awt.Component` and `java.awt.Container` taken together with the basic Swing-class `javax.swing.JComponent` are lightweight, that means their functionality is realized in given Java byte-code. (Heavyweight Java-code in a Java Runtime Environment has mapped most of its methods directly to equivalent native methods of associated native screen-components within the current computer-platform.)

The class `java.awt.Component` has been declared abstract, `java.awt.Container` is not (in Java Versions 1.0 to 1.1 `Container` was declared abstract). The abstract class `javax.swing.JComponent` is the third step in the direct extension of `Component` and `Container`. Abstract classes have to be extended to become instantiable. Nevertheless remember that abstract classes may furnish non-abstract methods and declare variables!

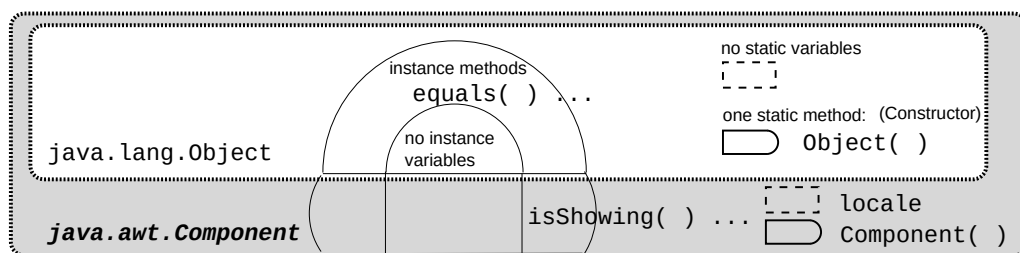


ALL are lightweight

Component and JComponent are abstract

The graphic above uses the class-notation introduced beginning with page 131. Four classes and examples of typical methods have been presented above. A commented listing of all the associated methods would take several pages and still may be subject to changes in future releases of Java. Therefore only an exemplary selection has been presented above and below.

Java: `java.awt.Component`



Java: `java.awt.Component` — All Java screen-components are objects of class-type `Component` and thus can be made visible on-screen by a call of their instance-method `setVisible(true)`. A programmer can override the methods `paint()` or `update()` in those AWT-components to put own drawings onto the screen²⁴⁹. Furthermore, such objects can respond to user-input, for example, by using the method `setEnabled(true/false)`: Then a call of the method `requestFocus()` from the programmer's side (or a mouse-click from the user's side) can put the screen-component into focus; this means the screen-component is ready to receive user-input in form of further mouse-clicks or keyboard-signals.

In Java the `java.awt.Component`-class gives the basic functions of any Java object which is taken to be a screen-component. The class has been declared abstract, which means it cannot be instantiated, there are no pure `Component`-objects, its only constructor is protected. This, as usual, does not exclude the possibility of references pointing to such objects being type-cast to the class-type `Component`. Or parameters within method-definitions may take the class-type `Component`, then the actual arguments in method-calls may reference objects of any sub-class of type `Component`.

²⁴⁹With with the class `javax.swing.JComponent` or extensions, the method `paintComponent()` is the appropriate method to override.

A selection of instance-methods belonging to the class `java.awt.Component` is presented below. Some of those methods may be recognized as already having been used in previously presented Java source-code: (Slashes in a parameter-list indicate alternatives in the parameter-list. This also marks overloaded methods.)

(Display)		
Returns <code>true</code> if component can be seen on the screen (if component <code>isVisible()</code> and in a container that is visible and showing).	<code>public boolean</code>	<code>isShowing()</code>
When <code>true</code> , component will be showing only if its parent-container is showing.	<code>public boolean</code>	<code>isVisible()</code>
Makes the component's screen representation visible; shows top-level components (in Container the helping equivalent is <code>doLayout()</code>).	<code>void</code>	<code>setVisible(boolean)</code>
Method to be overridden by the programmer. Lets the programmer draw something within the graphics context onto the screen-area of the component. That method is called by the Java Runtime Environment after the Component-object has been changed, for example by resizing.	<code>public void</code>	<code>paint(java.awt.Graphics)</code> AWT
Clears the screen-area of the component, - sets background- and foreground colors to their default values, and calls the <code>paint()</code> -method. (This method is not useful for doing animations because the successive clearings make the screen flicker.)	<code>public void</code>	<code>update(java.awt.Graphics)</code> AWT
Asks the system to refresh the display as soon as possible by calling the <code>update()</code> -method. A time-parameter delays the <code>update()</code> -call maximally by the number of milliseconds indicated by the given value of primitive-type <code>long</code> .	<code>public void</code>	<code>repaint()</code> <code>repaint(long millisec)</code> <code>repaint(long millisec, int x,y,width,height)</code>
(Position/Size)		
Gets the location of this component relative to the parent's coordinate-space. (see below for <code>java.awt.Point</code>)	<code>public Point</code>	<code>getLocation()</code>
Positions the component with respect to parent-component's coordinate system.	<code>public void</code>	<code>setLocation(int, int / Point)</code>
Gets the location of this component in screen coordinates.	<code>public Point</code>	<code>getLocationOnScreen()</code>
Changes height and width of the component's screen representation.	<code>public void</code>	<code>setSize(int,int / Dimension)</code>
Gets current height and width of the component in points.	<code>public Dimension</code>	<code>getSize()</code>
Changes height and width AND coordinates relative to the parent-component.	<code>public void</code>	<code>setBounds(Rectangle/x,y,width,height)</code>
Gets the component's width, height, and location relative to its parent.	<code>public Rectangle</code>	<code>getBounds()</code>
Does the component contain the screen point?	<code>public boolean</code>	<code>contains(Point / int, int)</code>
Preferred sizes of components are default sizes. Either they are given by the computer-platform when invoking the method <code>getPreferredSize()</code> . Or the values of preferred sizes can be changed by overriding the method <code>getPreferredSize()</code> . There is no method <code>setPreferredSize()</code> .	<code>public Dimension</code>	<code>getPreferredSize()</code>
(determined by screen-dimension)	<code>public Dimension</code>	<code>getMaximumSize()</code>
(determined by layout)	<code>public Dimension</code>	<code>getMinimumSize()</code>
(Appearance)		
(Endpoint of a container-call chain) Empty method in Component-class.	<code>public void</code>	<code>doLayout()</code>

(Container relevant, usually used only by the system) Ensures that this component has a valid layout. (This method is primarily intended to operate on instances of type Container.)	public void	validate()
(Container relevant, this component and all parents above it are marked as needing to be laid out. This method can be called often, so it needs to execute quickly.)	public void	invalidate()
Components are invalidated when they are first shown on the screen.	public boolean	isValid()
Default color is the one of the parent-container.	void	setBackground(Color)
Gets the current background color.	Color	getBackground()
Sets the default color used for drawing.	void	setForeground(Color)
Gets the current foreground color.	Color	getForeground()
Gets the current color-model (see below).	public ColorModel	getColorModel()
Factory-like-method, creates a java.awt.Graphics context for this component, thus enabling direct drawing on the screen-component. Or with Swing-components, enables drawing on their content-pane.	public Graphics	getGraphics()
(Appearance- time-delayed)		
Creates an off-screen drawable image from the specified image-producer.	public Image	createImage(ImageProducer)
Creates an off-screen drawable image to be used for double-buffering. (see also page 529)	public Image	createImage(int width, int height)
Returns the status of the construction of a screen-representation of the specified image. This method does not trigger the loading of an image. An application has to call the prepareImage()-method to force the loading of an image. Then the method checkImage() returns a mask (the int-value). This mask, bitwise inclusive ORed with ImageObserver flags, indicates what information about the image is currently available.	public int	checkImage(Image, ImageObserver)
Prepares an image for rendering on this component. The image-data is downloaded asynchronously (=in another thread) and the appropriate screen-representation of the image is generated. Returns true if the image has already been fully prepared; false otherwise.	public boolean	prepareImage(Image, ImageObserver)
	public boolean	prepareImage(Image, int width, int height, ImageObserver)
(I/O-state/Focus/Listening)		
The Java Abstract Window Toolkit (AWT) furnishes two possibilities for handling events that come from keyboard and mouse:		
First, register listener-objects and delegate event-handling to these objects. (see page 573)		
Argument false means: Any event-handling on this component has been blocked. The component cannot come into focus and take input.	public void	setEnabled(boolean)
Returns true by default.	public boolean	isEnabled()
Programmer tells the component to be ready to receive keyboard- or mouse-input.	public void	requestFocus()
Component can receive focus with a TAB- or SHIFT-TAB-keypressing issued by the user.	public boolean	isFocusTraversable()
Programmer-induced transfer of focus to the next component.	public void	transferFocus()

Identifies, registers receivers of events. *=Component, Focus, Key, Mouse, Mouse-Motion	public synchronized void	add*Listener(*Listener)
Removes, de-registers, registered receivers of events.	public synchronized void	remove*Listener(*Listener)
Dispatches references of the given event-object from this component to registered receivers.	public final void	dispatchEvent(java.awt.Event)

- The second method to handle events is to process the event-object within the extended component itself (see on **non-delegation event-handling**, page 591.): Enabled components can react to user input and generate event-objects without having registered listeners.

Configures the component to listen for events without having any listeners registered.	protected final void	enableEvents(long eventmask) (Masks are listed below this table under (x))
Reverses the previous method-call.	protected final void	disableEvents(long eventmask)

If overridden, a method-call `super.processEvent()` should be issued to keep the default function: References of event-objects are passed to registered listeners and event-specific processing methods like the ones that follow: (see also page 591.)

*=Focus, Key, or Mouse	protected void	processEvent(java.awt.AWTEvent)
	protected void	process*Event(*Event)
	protected void	processMouseEvent(MouseEvent)

(Reflection)

Give the component a name.	public void	setName(String)
Returns the component's name or rather: Returns a String-reference of the component's name.	public String	getName()
Get a reference to the child-component placed at the given point.	public Component	getComponentAt(int, int / Point)
Gets a reference to the container that contains this component, the parent.	public Container	getParent()
The default implementation is a call of the <code>paint()</code> -method.	public void	print(java.awt.Graphics)
Contains the component's name, its position on-screen, its width and height, validity, visibility and whether it has been enabled or not.	protected String	paramString()
Returns a String-reference to the paramString.	public String	toString()
A listing of this component's data to the standard system-output-stream like <code>System.out</code> .	public void	list()
		list(PrintWriter)
Factory-like-method, returns a reference of Toolkit-type to some native code, associated to the current component.	public Toolkit	getToolkit()

(x) Integer-values of primitive-type "long" from the class `java.awt.AWTEvent`, which represent the masks are

```
public final static long *_EVENT_MASK,
// where *=ACTION, ADJUSTMENT, COMPONENT, CONTAINER, FOCUS,
        ITEM, KEY, MOUSE, MOUSE_MOTION, TEXT and WINDOW
```

Multiple kinds of events can be chosen by combining masks using the bitwise logical or-operation (`|`), thus "ORing" masks, for example by taking `MOUSE_EVENT_MASK | MOUSE_MOTION_EVENT_MASK`.

The following classes store data: Two or four values of primitive-type `int`, in the case of the `Color`-class three `float`-values. Each of these classes has its specific self-explaining methods.

`java.awt.Point` Two `int`-values, an x- and an y-coordinate.

`java.awt.Dimension` Two `int`-values, a width- and an height-coordinate.

`java.awt.Rectangle` Four `int`-values, representing the coordinate-points of the rectangle and its dimension.

`java.awt.Color` Three float-values between 0.0 and 1.0 representing a color in the RGB color-model²⁵⁰. Beginning with Java Version 1.2, the `Color`-class adds transparency to its color specification and uses a fourth float-value to code color-data using the ARGB-model (page 526). The float-value for coding the transparency is given the range 0 **for transparent** to 1 for opaque. The range 0.0 to 1.0 of the float-values translates to 0 to 255 in the case of transforming the values into int-values, for example when used as arguments in a constructor-call.

0 =transparent
and 1 =opaque

`java.awt.image.ColorModel` This default model codes the RGB-data in an integer-value in the format given by a hexadecimally noted number as `0xAARRGGBB` where `AA`, 1byte, the alpha-value, describes the partial transparency (the value `00`, zero means fully transparent) and the remaining three bytes give the red, green and blue contributions as a pair of literals, like `"00"`, `"FF"` or an intermediate value `"06"`. The class `ColorModel` is to be used for offering the services of a specific color-model, whereas the class `Color`, ideally is meant to identify a color independently of any color-model.

`java.awt.Graphics` An abstract class for furnishing platform-specifically implemented methods to draw onto the screen-component's screen-area. This class contains a lot of `"draw...()"` methods.

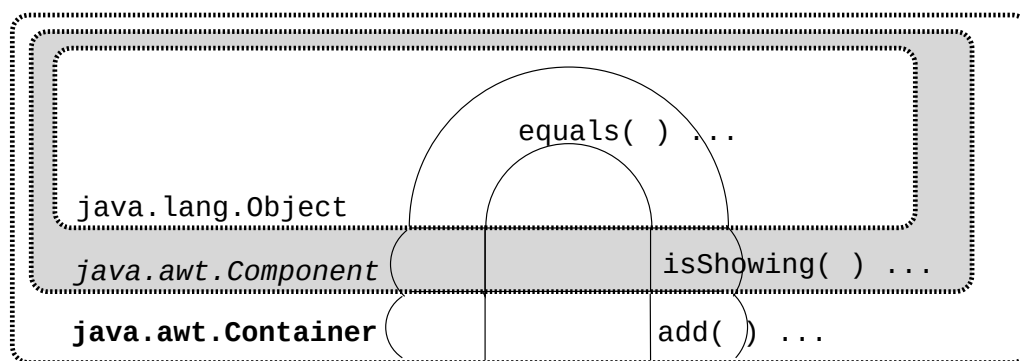
`java.awt.Image` An abstract class with a special function: An image given in a native format on the platform of the current Java Runtime Environment (JRE) can be given a reference of type `Image` and used within the JRE without any additional (time-consuming) conversion into a Java object. Therefore this class provides only few methods for retrieving information about the image's size, source and format. But this class also can be used as a super-class for any images used in the Java context.

`java.awt.Toolkit` an abstract class with platform-specifically implemented utility-methods, furnishes also additional data about the native platform, much like `System.getProperties()`. (See page 258)

There are many other methods in the `java.awt.Component`-class; for example the `getAlignment()`-methods or cursor- and font-settings. They exceed the scope of an introduction followed here.

Java: `java.awt.Container`

Java: `java.awt.Container` — Containers extend Components insofar as containers may contain one or several screen-components. The components of a container are numbered, these numbers are used to identify the components when referencing them. Organizing components within containers can be done with the bulk of methods in the non-abstract²⁵¹ Java class `java.awt.Container`. Several concrete ways to position and shape the container's components within the container's screen-area are given by objects that implement the `java.awt.LayoutManager`-interface. Therefore layout-managers have been taken as a notion separated from the `Container`-class. The layout-managers usually are given, ready-to-use, by software-libraries.



A selection of the methods, specifically introduced to serve the need of a `Container`-object, is presented in the list below:

(Display)

²⁵⁰Information about the color of a pixel can be given in many different formats:

RGB-model: Red, Green and Blue values going from black (0,0,0) to pure red (1,0,0), from black to pure green (0,1,0) from black to pure blue (0,0,1). Advantage: Straight lines in the color-cube make a continuous color-transition, therefore this is used in internal color processing of computers.

HSV-model: Hue, Saturation, Value coding color, intensity and brightness which makes it easier for a human to select a specific coloration.

The CYM K-model is useful for printing.

²⁵¹The class `java.awt.Container` has been designated non-abstract beginning with Java Version 1.2.

Overrides <code>Component.paint()</code> . Invoked by AWT's paint-manager, if painting of the container is needed. Paints the container and delegates further painting to lightweight descendants (child-components) of the container. Avoid overriding without calling <code>super.paint()</code> ! (Overriding has been introduced on pages 135 and 168)	<code>public void</code>	<code>paint(java.awt.Graphics g)</code> AWT
(Grouping/Layout)		
Add the specified component to the end of the list of child-components of this container. (This line and the following ones are arch-examples for the overloading of methods as introduced on page 124)	<code>public Component</code>	<code>add(Component)</code>
Add the specified component to this container at the given position in the component-list.	<code>public Component</code>	<code>add(Component, int)</code>
Adds the specified component to the end of list of child-components of this container. With layout-specific constraints given as an object (depending on <code>LayoutManager</code> used).	<code>public Component</code>	<code>add(Component, Object constraints)</code>
Adds the specified component to this container with the specified constraints at the specified index.	<code>public Component</code>	<code>add(Component, Object constraints, int)</code>
Removes the specified component from this container.	<code>public void</code>	<code>remove(Component)</code>
Removes the component, specified by index, from this container.	<code>public void</code>	<code>remove(int)</code>
Removes all the child-components from this container.	<code>public void</code>	<code>removeAll()</code>
Gets the number of components in this container.	<code>public int</code>	<code>getComponentCount()</code>
Gets a reference to the nth component in this container.	<code>public Component</code>	<code>getComponent(int n)</code>
Gets an array of references to all the components in this container.	<code>public Component[]</code>	<code>getComponents()</code>
Gets a reference of the component which contains the x,y-position or the specified <code>Point</code> -object, relative to the container's origin.	<code>public Component</code>	<code>getComponentAt(int, int/Point)</code>
Sets the layout-manager for this container.	<code>public void</code>	<code>setLayout(LayoutManager)</code>
Gets a reference to the layout-manager for this container.	<code>public LayoutManager</code>	<code>getLayout()</code>
Lets the layout-manager do the layout of this container. (This is usually called when the container is validated.)	<code>public void</code>	<code>doLayout()</code>
Determines the insets of this container, which determine the size of the container's border in which the layout-manager becomes active.	<code>public Insets</code>	<code>getInsets()</code>
Sets this container and its children valid without activating the layout-manager.	<code>public void</code>	<code>validate()</code>
This component and all parents above it are marked as needing to be laid out.	<code>public void</code>	<code>invalidate()</code>

Returns a reference to the preferred size of this container by calling <code>preferredLayoutSize()</code> of the current <code>LayoutManager</code> . Overriding changes the values.	public Dimension	<code>getPreferredSize()</code>
Returns the maximum size of this container, determined by <code>LayoutManager</code> .	public Dimension	<code>getMaximumSize()</code>
Returns the minimum size of this container, determined by <code>LayoutManager</code> .	public Dimension	<code>getMinimumSize()</code>
Paints each of the components in this container.	public void	<code>paintComponents(Graphics)</code>
Forwards calls to the <code>paint()</code> -methods of all the lightweight components inside the container.	public void	<code>paint(Graphics) AWT</code>
(Containment-Hierarchy)		
Paints this component and all sub-components, this means all children and all grandchildren and so on.	public void	<code>paintAll(java.awt.Graphics)</code>
Returns <code>true</code> if the container is a parent or contains containers which are parent(s) of the component (*).	public boolean	<code>isAncestorOf(Component)</code>
(I/O-state/Focus/Listening (continued))		
Adds the container-listener to receive container-events from this container.	public synchronized void	<code>addContainerListener(ContainerListener)</code>
Removes the specified container-listener, so it no longer receives container-events from this container.	public void	<code>removeContainerListener(ContainerListener)</code>
(Reflection (continued))		
Lists all information about the component and the sub-components recursively onto the output channel. The integer-value determines the indentation in units of spaces.	public void	<code>list(PrintWriter, int)</code>

(*) Note the different approaches of the method `java.awt.Container.isAncestorOf(java.awt.Component)` and the closely related method of the super-class `java.awt.Component.getParent()` towards decisions on the containment-hierarchy. And observe the difference of meaning to `java.lang.Class.getSuperclass()`, which concerns a totally different kind of hierarchy, namely the class-hierarchy! As has been mentioned on page 543, the hierarchy of containment of `java.awt`-objects represents something entirely different than the class-hierarchy (page 135).

There are some helper classes like

java.awt.Inset Four `int`-values determine a border of the container in which the layout-manager becomes active. Drawing remains possible on the entire area of the container.

java.awt.LayoutManager Interface furnishing the standard methods used to automatically position the set of child-components within the container's screen-area. Activities of layout-managers already have been visualized by the graphic on page 543. The following gives a list of standard classes implementing the `LayoutManager`-interface:

java.awt.FlowLayout causes the child-components of a container to be arranged from left to right in a line. If there are too many components for a single line, then a second line is used.

java.awt.BorderLayout arranges maximally five components: Two of them as a "North"- and "South"-bar and the remaining three of them as a "West", "Center" and "East" line in between. If there are more than five components, then those with index six and higher are omitted.

java.awt.GridLayout has to be given the number of rows and columns, then a table with cells of equal size is filled with the child-components. Those components with an index that exceeds the number of cells are omitted. (For an example see page 569)

java.awt.GridBagLayout is an extension of **GridLayout** insofar as cells can be merged to contain a single child-component.

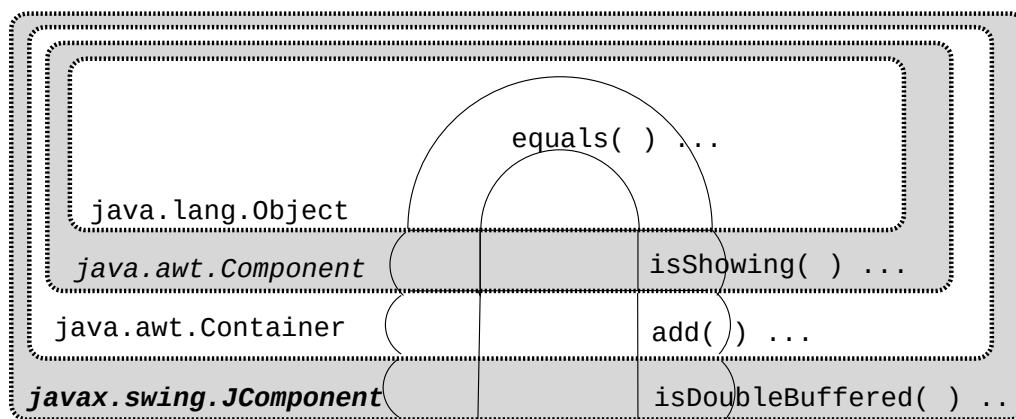
java.awt.CardLayout layers the child-components like a pack of cards and thus shows only one component at a time! The programmer has to furnish the means for the user to make a specific card appear on the screen.

Different layouts can appear in one container by introducing child-containers each having its own layout. Keep in mind that the class `java.awt.Component` has been declared abstract, that means the class `Component` has to be extended to instantiate an object; the class `java.awt.Container` has been declared non-abstract and is a lightweight class. The class `java.awt.Panel` makes it possible to implement a heavyweight `Container`-object, the default layout of the `Panel`-class is `FlowLayout`, the constructor `Panel(java.awt.Layout)` of the `Panel`-class allows to modify that layout, otherwise the layout could be changed with the `java.awt.Container.setLayout(Layout)`-method.

`java.awt.Panel` constitutes an instantiable heavyweight sub-class of the class `java.awt.Container` (now non-abstract). The class `Panel` often is used for instantiating containers with a specific layout-manager. A container, like a top-level screen-component, may contain several panels each one using a different layout-manager. one container with "multiple" layouts

`java.awt.ScrollPane` represents a heavyweight container with a horizontal and vertical `Scrollbar`

Java: javax.swing.JComponent



Java: `javax.swing.JComponent` — The pivotal class which introduces implementations of many of the Java Swing functionalities is the abstract class `javax.swing.JComponent`. The Swing-specific services include automatic double buffering, an extended painting-infrastructure with a specially over-ridden `paint()`-method that calls three auxiliary methods (`paintComponent()`, `paintBorder()` and `paintChildren()`) and a content-pane with different transparent layers. A screen-component of class-type `javax.swing.JComponent` includes a configurable tooltip, which is a name for a pop-up-label that can take string-information for display when the (Swing-)screen-component gets focused (Example: Graphic on page 600). The class `JComponent` features a Java-Beans support and the use of so-called pluggable-look-and-feel themes (pre-written overall graphical styles of single-user-interfaces, behind the horizon of this text's focus).

(Display)

Returns the state of the <code>JComponent</code> 's property "DoubleBuffered".	public boolean	<code>isDoubleBuffered()</code>
Rendering of component-graphics is done into an off-screen buffer, then the bitmap is copied into the screen area, giving a smoother appearance.	public void	<code>setDoubleBuffered(boolean TrueOrFalse)</code>

If the programmer knows that the child-components are tiled (they don't overlap), then this method should be overridden to return <code>true</code> . The repaint-manager of Swing will use this information to speed the rendering of the on-screen appearance.	<code>public boolean</code>	<code>isOptimizedDrawingEnabled()</code>
Overrides <code>Container.paint()</code> . Invoked by Swing's <code>RepaintManager</code> if painting the component is needed. Delegates painting to the three following paint methods in the given order: Calls a delegate's <code>paint()</code> -method; where the <code>UIDelegate</code> is given by the current <code>Look-and-Feel</code> to be rendered. If there is no delegate, then this method does nothing. This is also the method to override in customizing the basic appearance of this Swing-component. Paints the border of the Swing component. Calls successively each child-component's <code>paint()</code> -method.	<code>public void</code> <code>protected void</code> <code>protected void</code> <code>protected void</code>	<code>paint(java.awt.Graphics g)</code> <code>paintComponent(java.awt.Graphics g)</code> SWING <code>paintBorder(java.awt.Graphics g)</code> SWING <code>paintChildren(java.awt.Graphics g)</code> SWING
Called by programmer to update the Swing-component within the indicated rectangle.	<code>public void</code>	<code>repaint(java.awt.Rectangle r)</code>
Called by programmer to update the Swing-component.	<code>public void</code>	<code>repaint()</code>
Called by programmer to update the Swing-component within the indicated rectangle (the time-parameter has been deactivated).	<code>public void</code>	<code>repaint(long time, int x, int y, int width, int height)</code>
Paints the Swing component. Unlike in <code>java.awt.Component</code> , in Swing equivalent to <code>paint()</code> .	<code>public void</code>	<code>update(java.awt.Graphics g)</code>
Makes the installed Look-and-Feel visible.	<code>public void</code>	<code>updateUI()</code>
(Position/Size)		
(Appearance)		
Get the drawing-area of the component.	<code>public JRootPane</code>	<code>getRootPane()</code>
Make the background of this Swing-component non-transparent.	<code>public void</code>	<code>setOpaque(boolean isOpaque)</code>
Factory-method for the non-abstract class <code>javax.swing.JToolTip</code> (The little rectangle, containing helpful text, that appears on-screen when the mouse-pointer touches this Swing-component).	<code>public JToolTip</code>	<code>createToolTip()</code>
Stores a text for the tooltip of this Swing-component.	<code>public void</code>	<code>setToolTipText(String text)</code>
Returns a reference to the text of the tooltip.	<code>public String</code>	<code>getToolTipText()</code>
Intended to return the screen-location of the tooltip.	<code>public java.awt.Point</code>	<code>getToolTipLocation(java.awt.event.MouseEvent evt)</code>
Returns a reference to the <code>String</code> -value that identifies the <code>Look-and-Feel</code> UI-delegate of this Swing component (<code>JButton</code> → <code>"MetalButtonUI"</code>).	<code>public String</code>	<code>getUIClassID()</code>
Installs a new Look-and-Feel without making it visible (→ <code>updateUI()</code>).	<code>protected void</code>	<code>setUI(javax.swing.plaf.ComponentUI newUI)</code>

(I/O-state/Focus/Listening)

Registers listeners according to the bean-specification of this Swing-component. (page 311) - BEANS	public synchronized void	addPropertyChangeListener(...)

Registers listeners according to the bean-specification of this Swing-component. - BEANS	public synchronized void	addVetoableChangeListener(...)

Dispatches the indicated reference to an event-object to all registered listeners according to the bean-specification of this Swing-component. - BEANS	public void	firePropertyChange(String name, Object old, Object new)

Dispatches the indicated reference to an event-object to all registered listeners according to the bean-specification of this Swing-component. - BEANS	protected void	fireVetoableChange(String name, Object old, Object new)

Handler-method inside the Swing-component, called automatically when eventsEnabled() is set to true and the Swing-component generates an appropriate event-object. *=Focus / ComponentKey, Key, or MouseMotion	protected void	process*Event()

(Reflection)

Returns references to all the listeners registered to this bean component.	public EventListener[]	getListeners(Class listenerType)

There are some helper classes like

`javax.swing.plaf.ComponentUI` The part of the software for the screen-component which determines the graphical appearance of the screen-component. A pluggable-look-and-feel (plaf) can universally determine the graphical style of the entire user-interface (UI)

`javax.swing.JRootPane` offers a container that takes only a content-pane and a menubar. And which can be covered with a glass-pane (each child-component on that glass pane overlaps any child-component on the root-pane). (More on page 566)

As can be easily seen by the many new aspects the previous list introduces, screen-components taken from the Swing package combine additional complexity with additional flexibility.

Java: Frame and JFrame

Java: `java.awt.Frame` and `javax.swing.JFrame` — So far only basic specifications of implemented services have been presented. Questions that arise now may sound like “What types of screen-components, which extend the specifications above, can appear on-screen as top-level windows?” (`JFrame`, `JWindow`, `JDialog`)²⁵² The standard window that can appear on the screen is called a frame.

Remember the extensive use of the `JFrame`-components in the introduction of a selection of basic Swing-screen-components (beginning on page 544); here is given a kind of simpler introductory demonstration program:

```
public class GUIHelloWorldSwing
{ public static void main( String unused [] )
  { // instantiates the Component-object
    javax.swing.JFrame aFrame = new javax.swing.JFrame("Hello World" );
    aFrame.setSize( 300,100 ); // sets the frame large enough
    aFrame.setVisible( true ); // makes the component visible on-screen
  }
}
```

²⁵² Objects of type `javax.swing.JApplet` and objects of type `java.applet.Applet` are heavyweight. But screen-components of that type cannot appear as top-level windows, because applets are always made visible in some browser-like environment (an Internet-browser or an appletviewer). See also the rudimentary appletviewer on page 619, there too the applet appears inside a top-level-screen-component of type `java.awt.Frame`.

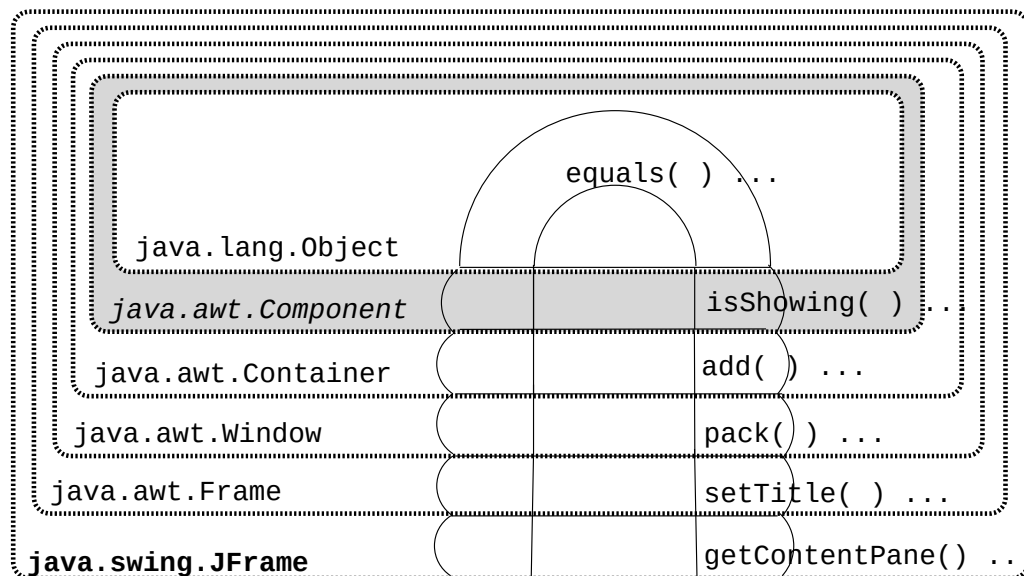
```

} }
/*
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% javac GUIHelloWorldSwing.java
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway% java GUIHelloWorldSwing
(result appears on screen-desktop)
^C (Ctrl-C)
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] marycosway%
*/

```

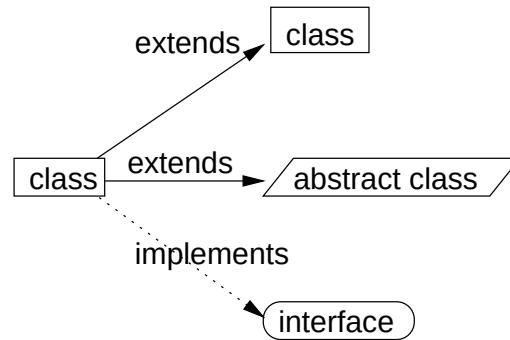


Such simple programs often conceal the richness of the underlying software-libraries. Here the class `JFrame`, for example, is a sub-class of the heavyweight screen-component `java.awt.Window`. The class `java.awt.Window` produces just an empty rectangle on the screen. (The AWT-heavyweight frame of class-type `java.awt.Frame` also extends the class-type `Window`.) The extended Swing counterpart with the name `javax.swing.JFrame`, being heavyweight as an extension of a heavyweight component, is one of the few heavyweight components in Swing (`JFrame`, `JWindow`, `JDialog` and `JApplet`). And one of three Swing-based screen-components (`JFrame`, `JWindow`, `JDialog`) which can appear directly on the desktop of the native operating-system.



In cases of a non-trivial minimal size of the participating components, the method `Window.pack()` may be preferred over the method `Component.setSize()`, because `pack()` activates the component's layout-manager to arrange the child-components automatically according to their preferred size. Otherwise, this preference may turn into a disadvantage if, in elementary cases, the default preferred size is set to zero, thereby making the top-level window really small.

The graphic above shows that the class/object-notation from page 131 can turn out to be cumbersome with larger extension-hierarchies. Another type of notation may be used to visualize non-elementary class-hierarchies: In this context remember also the graphic on page 186; there has been introduced a terser notation of class-extension and interface-implementation. If confronted with such an arrow-notation, then remember the sentence "follow the arrows to the super-classes (and (super-)interfaces)".



The graphic below shows the classes extended by the class `javax.swing.JFrame` and the implemented interfaces. Remember that the interface `java.io.Serializable` has just the function of a tag, it does not furnish any method-signature neither any constants. Whereas the `java.awt.image.ImageObserver`-interface requires the implementing class to define an `imageUpdate()`-method, which remains triggerable during the loading-process by an `ImageConsumer`-object, for example by referencing the `ImageObserver`-object inside a `draw...()`-method. The `java.awt.Component`-class implements three interfaces:

`java.awt.ImageObserver` declares the signature of one single method
`imageUpdate(Image i, int infoBitmap, int xUpperLeftCorner, int yUpperLeftCorner, int width, int height)`
 and some flag/bitmap-values. The Java Runtime Environment calls this method²⁵³ to inform the program about the progress of an image-loading-process (how to start an image-loading-process has been described on page 529). The Java Runtime Environment transfers the information by using the method's arguments. Progress-information is given encoded in an `int`-value that serves as a bit-mask, for more see page 606.

`java.io.Serializable` declares no variables neither method-signatures, this interface functions just as a tag to mark the class as being serializable (if wished, revise the section beginning with page 306).

`java.awt.MenuContainer` acts as some basic interface for AWT-menus. Why implemented twice? When implementing it the second time, at least one method may be overridden.

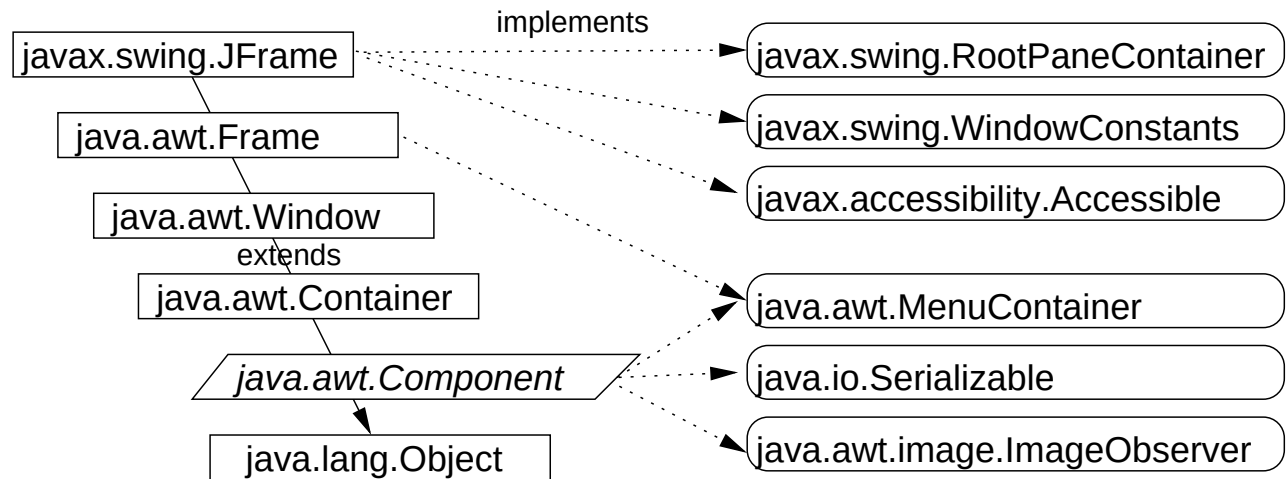
`javax.swing.RootPaneContainer`, is unnecessary in AWT, but in Swing this interface becomes the basis of multilayered filling of a top-level window with child-components. For more see page 566.

The interface `javax.swing.WindowConstants` furnishes just three constants of primitive-type `int`: `DISPOSE_ON_CLOSE=2`, `DO_NOTHING_ON_CLOSE=0` and `HIDE_ON_CLOSE=1`. The `int`-values may change in future versions of the Java code-libraries, therefore the use of the constant-names is encouraged. These are the constants used with the method `javax.swing.JFrame.setDefaultCloseOperation(int)`.

Finally, the interface `javax.accessibility.Accessible` just requires the implementing class to furnish a method `getAccessibleContext()` which returns a reference to an information-bearing object of class-type `javax.accessibility.AccessibleContext`. This object gives the basis for extended hard- and software-support of Swing screen-components (ideally this furnishes software-access to braille-readers and writers, extra-large on-screen representation and more).

These interfaces are described up-to-date in the Java Platform API Specification [JPAPIS].

²⁵³The method `imageUpdate()` is not intended to be called by a regular Java application-program, rather the Java Runtime Environment signals a change of loading status by calling that method itself; similar to the methods `main()` or `paint()` which usually are called by the Java Runtime Environment too. So, the method `imageUpdate()` becomes another example of an event-triggered method (page 127).



The class `java.lang.Object`, as the base-class of all the Java classes, has been introduced in previous sections and graphical representations (for example on page 164). A closer look on the classes `java.awt.Component` (abstract) and `java.awt.Container` has been cast in the previous sections.

The remaining classes `Window`, `Frame` and `javax.swing.JFrame` instantiate screen-components positionable directly on the desktop (top-level-screen-components) of the underlying operating-system. (There are few other classes of that kind: `JWindow`, `java.awt.Dialog`, `JDialog`, `java.awt.PopupMenu`)

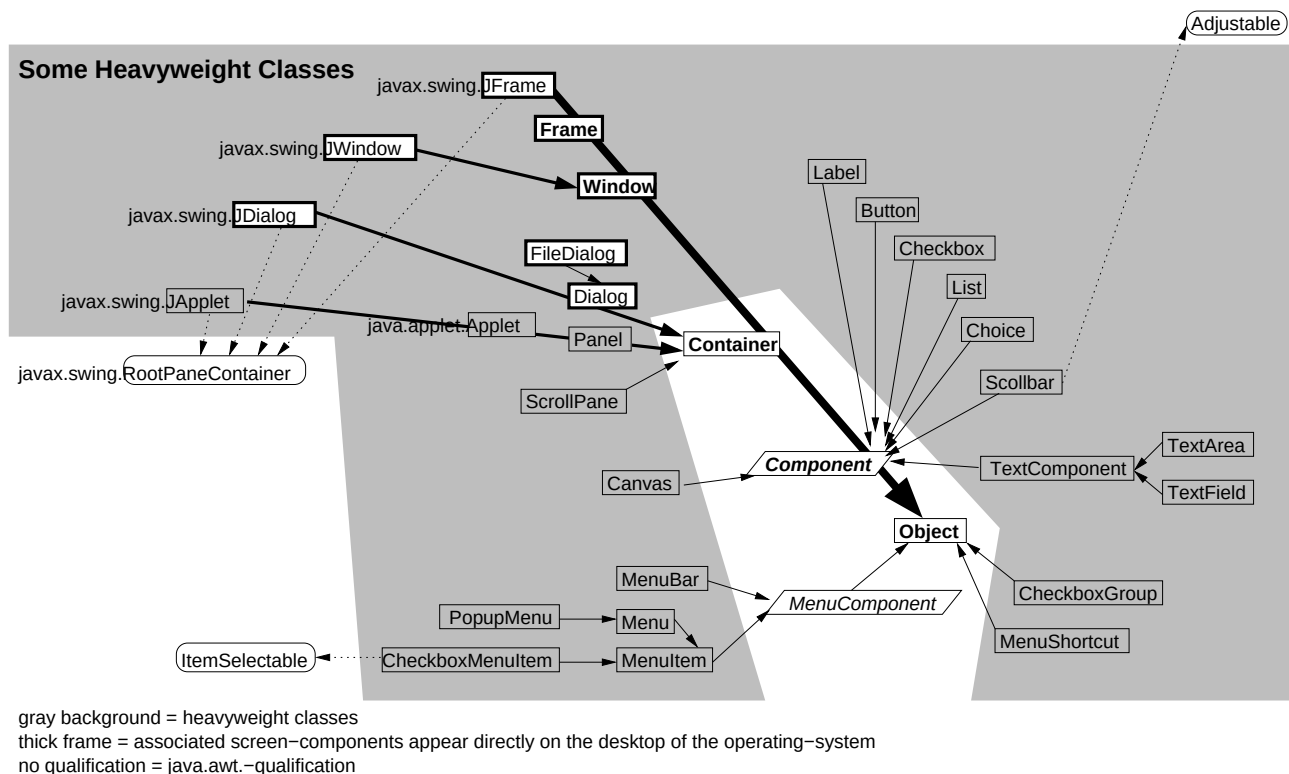
`Window` or `JWindow` screen-components appear on-screen, on the desktop, as blank rectangles; whereas `Frame` and `JFrame` screen-components appear with a titlebar and a border that lets an end-user easily drag the screen-component over the desktop or allows resizing the component. Screen-components of the `Dialog/JDialog`-type relate to another pre-existent screen-component and can be locked into being activated until disposed. So-called modal dialogs remain active until disposed. This effectively blocks all other windows of the application or of the desktop.

`javax.swing.JApplet` or `java.applet.Applet` produce Java screen-components in need of a special applet runtime-environment: This is given by a so-called applet-runner or a Java-enabled Internet-browser. Applet-screen-components can be seen as child-components inside HTML-texts displayed by Internet-browsers or appletviewers. For more consult the applet-section beginning on page 613.

A thorough presentation of the class `javax.swing.JFrame` is left to specialized handbooks or the Java Platform API Specification [JPAPIS].

AWT and Swing Class-Hierarchies

AWT and Swing Class-Hierarchies — Java AWT- and Swing-classes, for instantiating and supporting screen-components, occur by the dozens as indicated by the two class-hierarchies below. The hierarchy of classes which predominantly instantiate heavyweight screen-components has been condensed to the graphic below: (Every class or interface without a qualification is found in the `java.awt` package.)



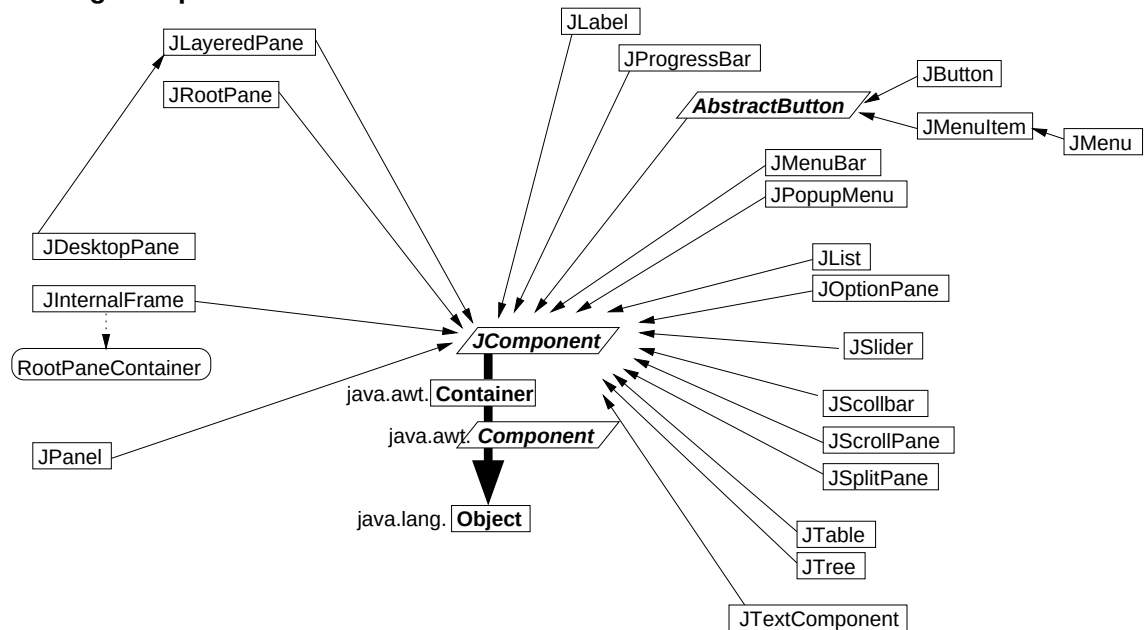
Those classes whose labels are surrounded by the shaded area produce heavyweight objects. That means, many of their Java-methods do directly trigger the corresponding functionality of the equivalent screen-component within the native operating-system. Heavyweight classes presented as labels with bordered and unshaded background can become top-level windows, these are screen-components that appear directly on the desktop of the native operating-system. All other non-abstract screen-components have to appear as child-components inside such top-level screen-components.

Observe that the classes `Component` and `Container`, as far as they can instantiate an object, may produce lightweight screen-components. Extending the `Container`-class constitutes the standard method of producing a lightweight component within Java's Abstract Window Toolkit (AWT).

Consider the class `javax.swing.JFrame` and the other classes from which the Swing heavyweight screen-components are derived, `javax.swing.JWindow`, `javax.swing.JDialog`, and finally the class `javax.swing.JApplet`: These Swing-classes are immediate extensions of the associable counterparts `java.awt.Frame`, `java.awt.Window`, `java.awt.Dialog` and `java.applet.Applet` which produce heavyweight screen-components. The other Swing-classes which produce lightweight screen-components extend the Swing-class `javax.swing.JComponent`:

top-level
window
custom
lightweight
component

Lightweight Swing-Components



Observe the kinship existing between several AWT- and Swing screen-components, already demonstrated by the choice of their names (`java.awt.Label` - `javax.swing.JLabel`, `java.awt.Button` - `javax.swing.JButton`, `java.awt.List` - `javax.swing.JList`, ... `java.awt.MenuBar` - `javax.swing.JMenuBar`, `java.awt.Menu` - `javax.swing.JMenu`). This similarity does extend to their *basic functionality* presented to an end-user. But, usually the pure concepts (AWT and Swing) and ensuing method-lists encountered by the Java programmer do differ considerably!

The lightweight Swing-components can use all the resources of the classes `Object`, `Component`, `Container` and `JComponent`. Some of these features have been introduced in preceding sections. The individual characteristics of classes like `JLabel`, `JProgressBar`, `JButton`, `JList`, ..., which instantiate fully-functional (lightweight) screen-components to be positioned inside some top-level-component, are not discussed here. (Hints have been given in the section beginning on page 544.)

This omission does not imply that mastering the use of classes like `JTable`, `JTree` or `JTextComponent` could be considered easy or straightforward, this is clearly not the case! But developing the systematics of their use can be considered to be far out of the reach of this text. The listing of these classes means to hint at the volume of know-how still to be acquired to use Swing-components proficiently, if found necessary.

The class `javax.swing.JPanel` is a lightweight container much like `java.awt.Container`. Extending the features of the class `Container` with its layout, `JPanel` can additionally control Swing features inherited from `JComponent`, like being double buffered, opaque or transparent. (The class `java.awt.Panel` represents a heavyweight extension of the lightweight class `java.awt.Container`.) Usually panels are used as child-components of a given container, each child-component implementing a different layout-manager; thus the container displays different layout-managers within its screen-area.

The two classes `javax.swing.RootPane` and `javax.swing.LayeredPane` together with the interface `javax.swing.RootPaneContainer` — implemented by `JFrame`, `JWindow`, `JDialog`, `JApplet` — give more control over child-components than their AWT-counterparts. The means of control include layering of child-components and, if needed, division of the root-pane into an area for the menu (`javax.swing.JMenu`) and a content-pane (`java.awt.Container`).

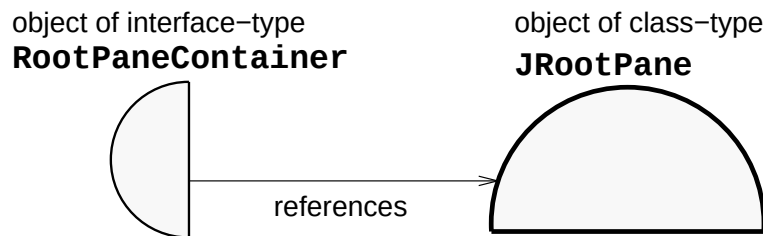
The two classes `javax.swing.DesktopPane` and `javax.swing.JInternalFrame` are the starting-point for programming an application that furnishes a desktop of its own, INSIDE A TOP-LEVEL WINDOW (for a minimalistic introduction see page 610).

Java: ContentPane and JLayeredPane

Java: `ContentPane` and `JLayeredPane` — The five Swing-classes `JFrame`, `JWindow`, `JDialog`, `JApplet` and `JInternalFrame` implement the interface `javax.swing.RootPaneContainer`. The first four classes instantiate heavyweight top-level screen-components. ALL THESE SCREEN-COMPONENTS HAVE EXACTLY ONE CHILD-COMPONENT: An instance of class-type `JRootPane`.

one container
with "multiple"
layouts

RootPaneContainer
contain a JRootPane



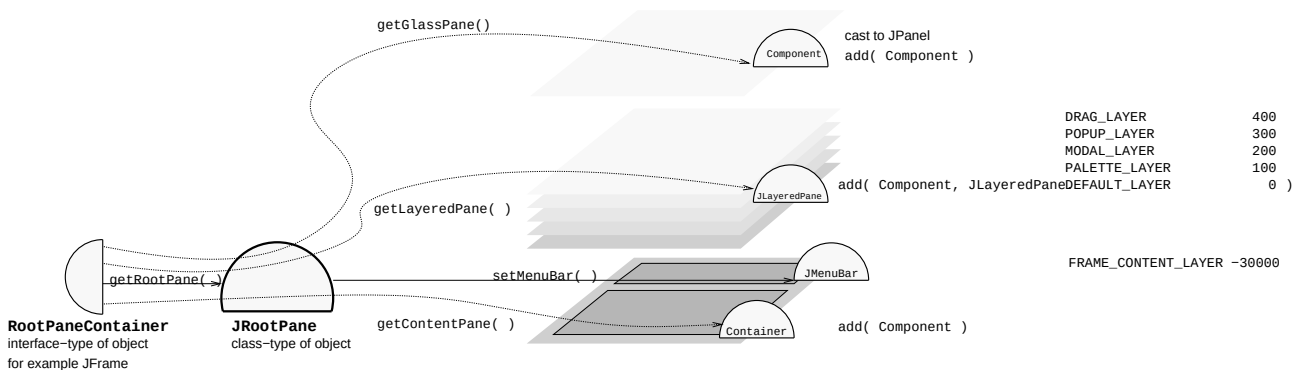
Thus an object of interface-type `RootPaneContainer` references the `JRootPane`-object which also furnishes the standard container for positioning child-components inside the associated screen-area of the object `RootPaneContainer`. So far, the overall service is comparable with that of an AWT-component, but the Swing-context offers additional features: Child-components may be placed within an entire stack of layers, child-components in the upper layers visually overlapping the child-components within the lower layers. The layers can be characterized by dividing them into three types of panes:

glass-pane given by an object of type `Component`, CASTABLE TO `JPanel`²⁵⁴. Drawings on this component or child-components, added to this panel on the screen, always do visually cover all used layers of the underlying screen-component of type `RootPaneContainer`. Thus the glass-pane covers the entire fillable pane of top-level screen-components in Swing and of objects of type `JInternalFrame`.

layer-panes are given by a single object of type `JLayeredPane`. This object of super-type `Container` has `add(Component, int)`-methods which use integers to specify the accessed layer. Some predefined integer-constants (`POPUP_LAYER = 300`) denote standard layers. The gist of this layer-concept is that the child-component in upper layers visually overlap those in lower layers.

base-pane is represented by the `JRootPane`-object and divided into a content-pane and a menu-area which are managed by two objects, respectively a `Container`-object and an object of type `JMenuBar`. Menubar and content-pane are located on a `FRAME_CONTENT_LAYER` when referenced within the domain of a `JLayeredPane`-object.

All these features are controlled by the object of class-type `javax.swing.JRootPane`: The `JRootPane` can get references to the pane-related objects by using the several methods (`create-/get-/setGlassPane()-LayeredPane()-ContentPane()`, `get-/setJMenuBar()`). These methods used by the `JRootPane`-object to get or set the references of the aforementioned objects (except the menubar) are redeclared in any `RootPaneContainer`, thus a `RootPaneContainer` can directly access the various objects for layer-management, as can be seen in the graphic below²⁵⁵.



What about layout? How are the added child-objects positioned in the various layers? Individual `Container`-objects have individual layout-managers, like the `JPanel`-object which represents the glass-pane

²⁵⁴ If this type-cast is possible, then the object of type `Component` has been instantiated from a sub-class of type `JPanel`. (see especially the graphics past page 169)

²⁵⁵ The method `getRootPane()` of a `RootPaneContainer`-object furnishes a reference to the `JRootPane`-object. To avoid a constantly explicitly re-occurring double method-call, `RootPaneContainer`-objects redeclare the method `getContentPane()` (`getLayeredPane()` and `getGlassPane()`) in the following form:

```
public java.awt.Container getContentPane()
{ return getRootPane().getContentPane();
}
```

This and the analogous forwarding implementations permit any `RootPaneContainer`-object to directly access the `JRootPane` methods without explicitly referencing the `JRootPane` itself!

or the content-pane of type `Container`. Contrary to the `JPanel`-object, the `JLayeredPane`-object has no default layout-manager: But each used layer can be furnished with a layout-manager of its own, by adding first a `Container`-object equipped with the desired layout-manager and then adding the child-component to this container.

The following program demonstrates the use of that layering-infrastructure: A screen-component of type `JFrame` together with some functions of the associated `JRootPane` and `JLayeredPane`-objects are considered. Note especially the use of the `RootPaneContainer`-interface when directly producing references to the content-pane and the layered-pane.

The `java.awt.Component.setBounds()`-method (see also the table on page 553) does absolute positioning and sizing of the screen-component, if not overridden by various layout-managers.

```
import javax.swing.JFrame; // or JWindow, JApplet, JDialog, JInternalFrame
import javax.swing.JRootPane;
import java.awt.Component; // for glassPane
import java.awt.Container; // for ContentPane
import java.awt.Color;
import javax.swing.JLabel; // stands for any component
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JLayeredPane;
import javax.swing.JPanel;

class SwingRootPane // ComponentContainers in Swing
{
    public static void main( String unused[] )
    { JFrame aFrame = new JFrame("RootPane");

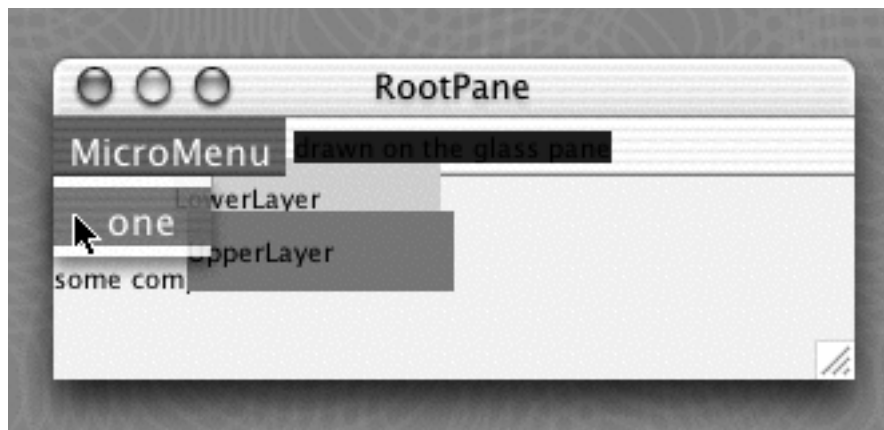
        // ----- glass-pane: top level ---
        Component glassPane = aFrame.getGlassPane();
        /* ready to take components like popup-menus or toolbars, even a LayoutManager,
           drawings (paint()) or event-objects (add...Listener)
        */
        JPanel cGlassPane = (JPanel)glassPane;
        cGlassPane.add( new TestComponent("drawn on the glass pane", Color.blue ) );
        cGlassPane.setVisible(true); // = new Color(0,0,255) RGB

        // ===== layered-pane: access to intermediate levels ===
        JLayeredPane paneStack = aFrame.getLayeredPane();
        JLabel upperComponent = new TestComponent("UpperLayer", new Color(100,100,255) );
        upperComponent.setBounds(50,35,100,30);
        // in pixels: location-horizontal, location-vertical, length, height
        JLabel lowerComponent = new TestComponent("LowerLayer", new Color(200,200,255) );
        lowerComponent.setBounds(45,15,100,30);
        paneStack.add( lowerComponent, JLayeredPane.DEFAULT_LAYER );
        paneStack.add( upperComponent, JLayeredPane.DRAG_LAYER );
        // paneStack.setVisible(true); not needed
        // ----- base-pane: frame content layer ----
        JRootPane rootPane = aFrame.getRootPane();
        JMenuItem item = new JMenuItem("one");
        JMenu menu = new JMenu("MicroMenu");
        JMenuBar aBar = new JMenuBar();
        menu.add( item );
        aBar.add( menu );
        rootPane.setJMenuBar( aBar );
        Container aContentPane = aFrame.getContentPane();
        aContentPane.add( new TestComponent("some component", new Color(240,240,255) ) );

        aFrame.setSize(300,120);
        aFrame.setVisible(true);
    } }

class TestComponent extends JLabel
{ TestComponent( String text )
  { this( text, null ); }
  TestComponent( String text, java.awt.Color color )
  { super( text );
    if( color != null )
    { setOpaque( true );
      setBackground( color ); // contrary to foreground, the letters of the text
    } } }
/*
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] andreadipietro% javac SwingRootPane.java
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] andreadipietro% java SwingRootPane
^C (Ctrl-C in terminal-window)
```

```
[localhost:~/JavaCodeFiles/SingleUserInterface/ScreenOutput] andreadipietro%
*/
```



As has been mentioned earlier, when doing Java programming with screen-components, the need for experimentation cannot be underestimated.

Java: Fonts

Java: Fonts — Remember the section beginning with page 266. How to put text, written in different fonts, actually on-screen? The following piece of source-code demonstrates this by using the abstract class²⁵⁶ `java.awt.GraphicsEnvironment` (introduced first in Java Version 1.2) with a factory-like-method that establishes an interface to native code:

```
import java.awt.GraphicsEnvironment;
import java.awt.Container;
import java.awt.GridLayout;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import java.awt.Font;
class PlatformFontViewer extends JFrame
{
    public static void main( String unused[] )
    { new PlatformFontViewer();
    }
    PlatformFontViewer()
    { super("Some Installed Fonts");
      GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
      String [] fontList = ge.getAvailableFontFamilyNames();
      int selection;
      if( fontList.length > 20 ) selection = 20;
      else selection = fontList.length;
      Container showPane = this.getContentPane();
      showPane.setLayout( new GridLayout( selection , 2 ));
      JComponent textComponent [] = new JComponent [ selection ];
      for( int i = 0; i < selection; i++ )
      { System.out.println( fontList[i] );
        showPane.add( new JLabel( fontList[i] ) );
        textComponent[i] = new JLabel( fontList[i] );
        // textComponent[i].setFont( Font.getFont( fontList[i] ) ); // does not work
        textComponent[i].setFont( new Font( fontList[i], Font.PLAIN, 16 ) );
        showPane.add( textComponent[i] );
      }
      //pack()
      setSize(200,200);
      setVisible(true);
    } }
}
/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > javac PlatformFontViewer.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput > java PlatformFontViewer
Bitstream Charter
BookmanL
CenturySchoolbookL
ChanceryL
```

²⁵⁶Such abstract classes with factory-like-methods accessing native code have been introduced on page 184.

```

Courier
Courier 10 Pitch
Cursor
Default
Dialog
DialogInput
Dingbats
GothicL
Lucida Bright
Lucida Sans
Lucida Sans Typewriter
LuciduxMono
LuciduxSans
LuciduxSerif
Monospaced
NimbusMonoL
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/ScreenOutput >
*/

```



This concludes the introduction of how to put graphics, screen-components and fonts on-screen.

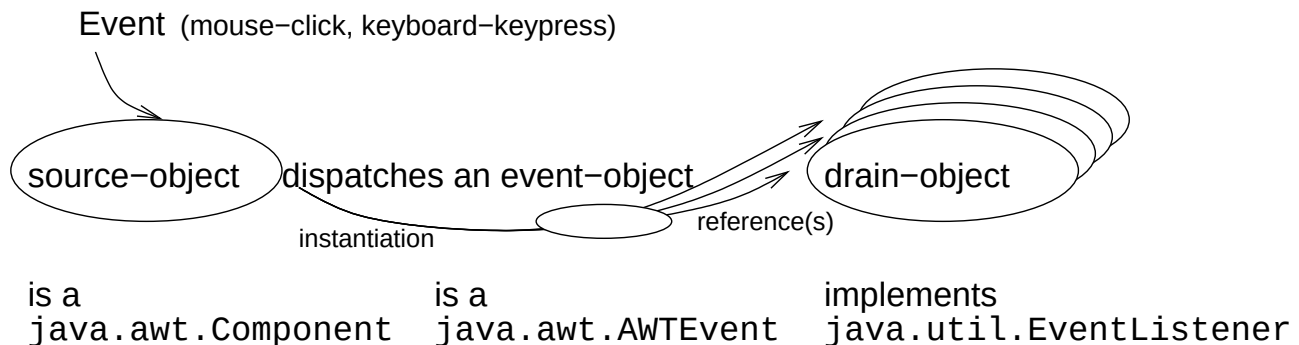
Event-Communication - Mouse and Keyboard

Event-Communication - Mouse and Keyboard — This chapter introduces the AWT (Abstract Window Toolkit)-implementation of the delegation event-model/subscriber-event-model: Event-sources register listeners. Then, the event-sources notify the listeners by sending references of event-objects. This fully-implemented infrastructure for event-propagation is described by considering trigger-, source-, event- and drain- or listener-functionalities. The AWT-implementation of the delegation event-model is also (re-)used by Swing-classes and -objects.

Event Delegation within the Abstract Window Toolkit (AWT)

Event Delegation within the Abstract Window Toolkit (AWT) — Recall the virtually unimplemented delegation event-framework, as introduced on page 286. This framework anticipates much of the fully-functional delegation event-infrastructure of the Abstract Window Toolkit within a Java Runtime Environment (JRE): After a drain-/listener-object has registered with a source-object – typically a screen-component –, the proper switching of event-objects is guaranteed. Then a mouse-click, a keyboard-keypress or, for example, internal timer-signals, also called external- and internal events, trigger the source-object into dispatching an event-object. The delegation event-infrastructure sends each of the registered listener-objects a reference of the single event-object. — Usually, next to registering listener-objects to source-objects, a Java programmer implements so-called handler-methods within the listener-objects, these handler-methods tell how to react to the initial event. On receiving the notification about an existing event-object, a listener-object automatically calls the appropriate handler-method.

AWT Event-Delegation

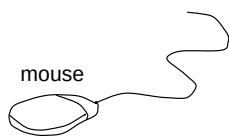


Another model, also called chain-model (see page 276), will only be mentioned shortly due to its scarce use. The chain-model channels events along the containment-hierarchy of screen-components, just like the exception-infrastructure channels exception-events along a method-call hierarchy. — This chapter uses the AWT delegation event-infrastructure to describe the processing of events coming from mouse, keyboard and Swing-/AWT-screen-components (AWT = Abstract Window Toolkit). A final section considers those parts of the automated switchboard-mechanism (event-dispatch-thread and `java.awt.EventQueue`-object) which can be influenced by the Java programmer.

Concept: Mouse Interaction

Concept: Mouse Interaction — Next to the keyboard, a mouse gives an arch-example of an event-generating piece of hardware: Essentially the mouse-function reduces to “mouse-key down”-signals, “mouse-key up”-signals and to signals coding the position of the mouse-pointer on the screen: After the mouse-pointer on

the screen has been positioned on some screen-component, a mouse may emit single or multiple clicks (“mouse up”-, “mouse down”-signals given in a short interval) or, when drawing, the mouse-pointer may be moved outside the screen-component without sending a “mouse-key up” signal.

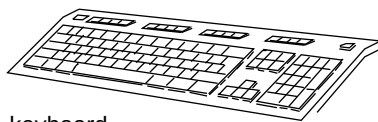


The “mouse-key up”- and “mouse-key down”-signals together with the position of the mouse-pointer, are registered by the operating-system and may be forwarded to a runtime-environment, there these events may be transformed to event-objects. Objects containing the unprocessed event-information are called **context free-** or **generic event-objects**. Generic means here original or context-independent, unprocessed.

An operating-system or/and runtime-environment may also pre-process generic mouse-event-objects: Depending on the kind of screen-component the pointer has marked, when a mouse-key has been pressed, the generic event-objects are used to produce context-dependent mouse event-objects: A “mouse-key down”-event, followed by a “mouse-key up”-event on a button-component, is interpreted as a button-click. Drawing the mouse on a list-component or on a text-field means selecting, giving an item-event.

What about mice with more than one key, sometimes mice can have up to three keys? Usually pressing one of the auxiliary keys of the mouse actually is equivalent to the pressing of the primary mouse-key, while the Ctrl-key of the keyboard is pressed simultaneously. And pressing another auxiliary mouse-key is seen as equivalent to pressing the primary mouse-key together with the ALT-key of the keyboard pressed simultaneously. Considerately-written software usually adheres to that rule. The concrete implementation depends on operating-system and the configuration of the mouse as left- or right-handed mouse.

Concept: Keyboard Interaction



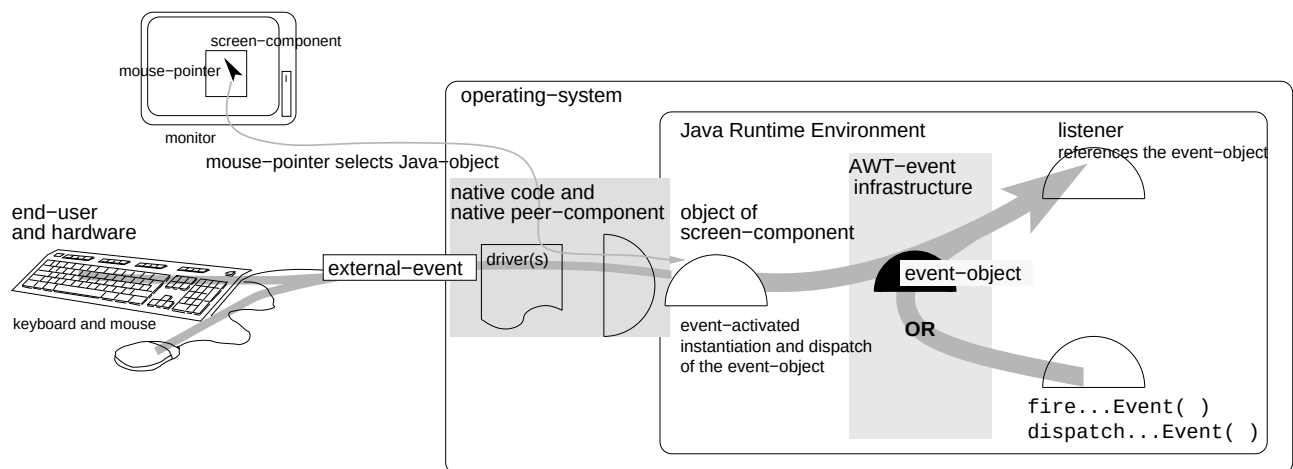
keyboard

Concept: Keyboard Interaction — Similar to the “mouse-key down”- and “mouse-key up”-signals, each single key on the keyboard sends two types of signals: “I have been pressed” and “I have been released”. This includes keys like SHIFT and CTRL. The pronoun “I” in the preceding sentences stands for a keyboard-specific code which is a part of the signal and identifies the key. — With that generic information the rest

of the hardware and the operating-system construct the wealth of key-combinations, like an upper-case letter (A) from succeeding keypress-signals (SHIFT pressed - A pressed - SHIFT released, some of the key-release signals may be ignored, if desired). Also here, the operating- or runtime-system transforms the context-free, generic keyboard-events to mostly context-dependent keyboard-event-objects.

Java: The AWT-Event-Communication

Java: The AWT-Event-Communication — The following part of the text requires basic knowledge of the source-event-drain concept (beginning with page 271) and the extension to the subscriber-model of switch-boarding and scheduling the events (page 276).



Consider the graphic above, in the first case a Java-object may fire or dispatch an event-object, and then let the AWT-event-infrastructure channel a reference of the event-object to a registered listener-object. This constitutes the case of messaging inside the Java Runtime Environment (JRE).

Alternatively, an end-user, using mouse and keyboard, produces electrical signals which the computer-platform receives. The native platform processes that information (hardware-controller, driver-software,

operating-system, custom-written handler-code) until it appears as an event-object in the Java Runtime Environment (JRE).

So mouse- and keyboard-interactions send messages or “external events” to some receiving entities. Such notification-processes also may be described with the source-switchboard-drain-model. These **messages** or **events** are interpreted by the operating-system (subsequently by a runtime-system) and are TRANSFORMED INTO **event(-object)s** within some software-environment.

events
transformed
into
event-objects

The position of the mouse-pointer either can be associated with the desktop or with a screen-component, like a button contained by a window. Especially an ensuing mouse-click should be associable to the receiving screen-component for further internal processing. Thus, a service of the operating-system or the runtime-environment is to ASSOCIATE EVENTS WITH THE SCREEN-COMPONENTS from which they originate. This makes it possible for the application-programmer to see SCREEN-COMPONENTS AS SOURCES OF EVENT-OBJECTS.

Furthermore, instead of receiving **generic, context-free event-objects** (mouse-up, mouse-down, mouse-moved to position, key-down, key-up, key-code), these event-objects may reflect the meaning of the input with respect to that screen-component, encoded in values of variables: For example, clicking on a button²⁵⁷ means something different than clicking into an area of the scrollbar to trigger a page-change. There can be imagined a wealth of so-called **context-related event-objects** (MouseEvent, MouseMotionEvent, ComponentEvent, WindowEvent, ... , ActionEvent). Remember also that screen-components, which are set by their program to accept input, are called **enabled** screen-components.

generic- and
context-
related
event-objects

If the wealth of imaginable contextual-events would be explicitly produced, each time the platform receives a hardware signal, then this entire concept could stall most computers, due to the sometimes vast number of event-objects to be generated from one generic event. To avoid a flow of unnecessarily generated event-objects a REGISTERING OF LISTENERS OR SINKS causes only those event-objects to be generated that are actually needed!

As already mentioned, on pure software level, the NOTIFYING-MECHANISM ALSO CAN BE USED FOR INTERNAL COMMUNICATION. For example a change of the window’s size by the end-user is communicated as a WindowEvent-object which triggers the firing of a java.awt.event.PaintEvent-object to automatically repaint the view of the screen-component, here the source and sink of the communication-process being the same screen-component.

Within AWT and Swing, sources (screen-components of class-type java.awt.Component), events (objects of class-type²⁵⁸ java.awt.AWTEvent), drains (objects of interface-type java.util.EventListener) AND the switchboard functionality are fully implemented by the Java Runtime Environment, together with a registering-mechanism! (switchboard-related notions are: event-dispatch-thread and an event-queue)

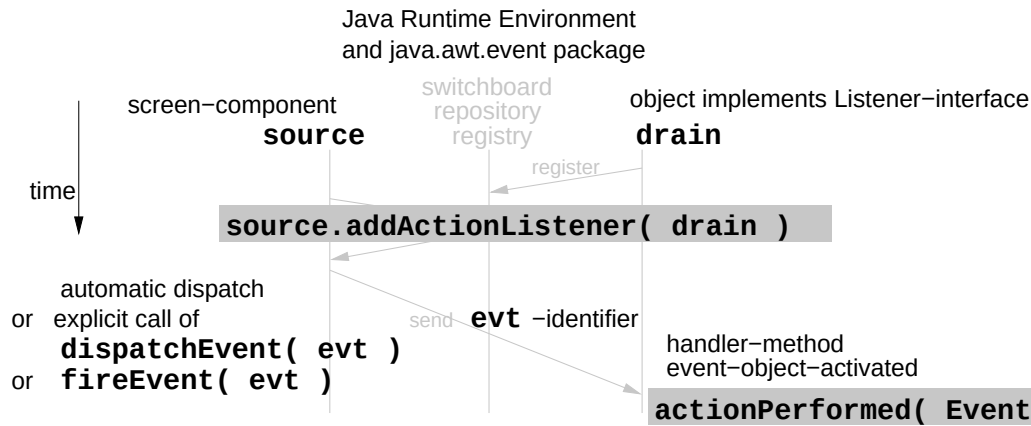
The dispatch of event-objects may be triggered by potentially internal changes (resizing a window) or firing an event-object may be caused by external events (pressing the mouse-key). Keyboard-usage or mouse-movements trigger the affected or activated screen-components to emit an associated event-object automatically. Even the Java programmer may dispatch an event-object by calling an instance-method (fire...Event(new AWTEvent())) of a screen-component.

The **delegation event-model** in Java Abstract Window Toolkit (AWT) has been available with the release of Java Version 1.1. The graphic below shows how a screen-component may produce only those event-objects for which a “listener” has been registered explicitly. Then, the production of unwanted event-types can be avoided right away and does not unnecessarily block resources of the platform. A piece of source-code, implementing the scheme below, already has been anticipated on page 544.

delegation
event-model

²⁵⁷Clicking: Rapid succession of “mouse-key down”- and “mouse-key up”-signals, having the mouse-pointer positioned on the screen-area of the screen-component.

²⁵⁸Since the class java.awt.AWTEvent has been declared abstract, all objects of that class-type are instantiated from a sub-class of the class java.awt.AWTEvent.



Compare this also to the more conceptual graphic on page 276, from which the graphic above has been derived. In the vicinity of the conceptual graphic referred to, can be found another description of the so-called chain-model of event-propagation. Java versions below Number 1.1 used a **component-chain-design of event-propagation** up the containment-hierarchy of the screen-components, much like the throwing of exceptions up the calling-hierarchy of the called methods (graphic on page 282). If needed, a **chain-design of event-propagation** may be emulated using the `processEvent()`-method in a special way as described beginning with page 593. The chain-design usually adds conceptual complexity to source-code.

chain-design
of event-
propagation

Java: Delegation Event-Handling

Java: Delegation Event-Handling — Triggers, source-objects, event-objects, switchboard, sink-objects and finally the handler-methods within the sink-objects are the main notions to be introduced in the Java context.

Java: Triggering

Java: Triggering — Standard triggering is done by the source, thus implementing a PUSH-model of event-triggering (graphic on page 274). In those cases, in which the triggering becomes a foreseeable feature of the screen-component, for example in a button-screen-component, **external triggering has been readily implemented**, so that the application-programmer just has to instantiate the screen-component and make it visible on the screen (For an example see the Java source-code for letting an end-user's pressing of a screen-button be noticed by the Java program; page 544).

But a Java application-programmer also can do the triggering explicitly by calling the method:

protected void fireEventType(...)-methods to be found preferably in Swing-classes (Java Bean classes) from which are instantiated mostly screen-components. For example: Within the class `javax.swing.AbstractButton` and its extensions that do not override the method differently, the method `fireActionPerformed( java.awt.event.ActionEvent )` can be called, to explicitly dispatch an `ActionEvent`-object. If appropriate listeners have been registered, then each listener automatically receives a reference to this event-object, otherwise nothing happens. The same occurs with the method `fireStateChanged()`, which fires a `ChangeEvent`-object within `AbstractButton` or `JSlider`.

The triggering-methods above can be found implemented in many pre-written screen-components. How can a Java programmer implement these methods? In the context of the Java-Beans infrastructure, calls of fire-event-methods have been documented within the source-codes on page 315 and on page 316. More on instantiating event-objects and subjecting them to Java's delegation event-processing can be found beginning with page 591.

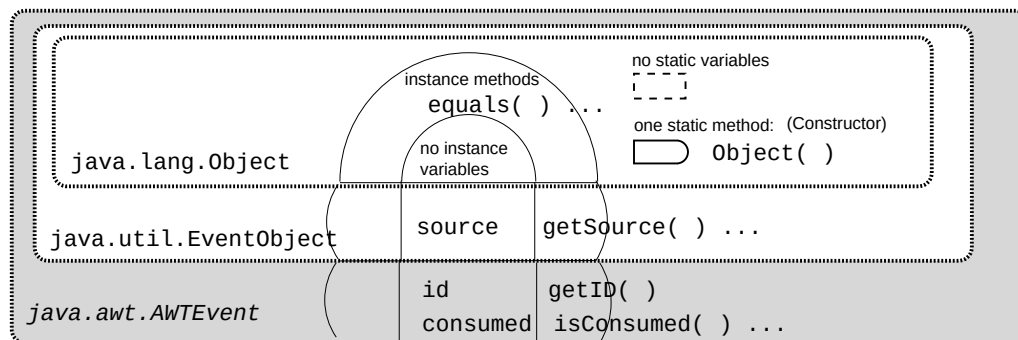
Java: Switchboarding/Event-Scheduling

Java: Switchboarding/Event-Scheduling — Switchboarding/Event-Scheduling is fully automated in AWT and Swing with the help of the `java.awt.event`-package. Within a delegation event-model, a registering-mechanism of listeners (sinks) tells the sources which event-objects are in demand. Thus only those event-objects are instantiated that are actually needed. Within the computer, the switchboard-mechanism

(incorporated in the operating-system and/or the runtime-system) processes or forwards the (external) events²⁵⁹. If needed, external events are converted into event-objects and then the Java Runtime Environment fires their references from a source or observable (the screen-component) to a drain or observer (an event-handler). Means of influencing this switchboard-mechanism, given to the Java programmer, are presented in a section (beginning with page 586) at the end of this chapter.

Java: Event Types

Java: Event Types — Since the operating-system or the Java Runtime System furnish an appropriate infrastructure of event notification, the event-objects themselves gain greater importance:



All AWT- and Swing-event-objects extend the abstract base-class `java.awt.AWTEvent`. — One of the main features of this class is given by an instance-variable "id" of primitive-type `int` which encodes a specification of the given event-object²⁶⁰. These id-values are given as constants (final variables of primitive-type `int`) within the respective extensions of the class `java.awt.AWTEvent`. (see below the table on page 577, for mouse-event-related constants) — Those `AWTEvent`s whose boolean instance-variable `consumed` is set `true`, are ignored by any listener or handler-routine. — The `AWTEvent`-class constitutes the base-class for all the following event-types:

ComponentEvent-objects inform about a component being hidden, moved, resized, made visible. The Java Runtime Environment uses them for internal signaling, application-programmers may listen to them only for informational purposes.

WindowEvent-objects tell about the window being iconified, deiconified, opened, closed, activated or deactivated.

FocusEvent-objects may tell about a component having gained focus or having lost focus. (Also temporarily, if the window, of which the component is a child, has been deactivated.)

AdjustmentEvent-objects may specify signals coming from scrollbar/slider movements or components with a similar input-behavior involving values or sub-intervals in a *quasi-continuous interval*.

ItemEvent-objects may cover the cases where the input-behavior involves a subset or a single choice out of a set of *tens of possible choices*.

TextEvent-objects may take the role of the messenger telling that something in a text-component has been changed.

Event-objects of type `ItemEvent` need to come from screen-components like a list or a menu. Such event-objects or event-types are sent or used exclusively by specific components, they represent events which can come only from special contexts (**contextual events**). And there are event-objects which can be dispatched by any screen-component (the event-class of type `ComponentEvent` can come from any Java screen-component, all the others above have to be considered as contextual). Event-types with instantiated event-objects, that could be fired from any screen-component, are called **context-independent-events**:

²⁵⁹In this context remember the event-framework given by the `java.util`-package. This package and its almost non-implemented event-infrastructure has been presented in a rudimentary form in the chapter about communication, beginning with page 286.

²⁶⁰The older `java.awt.Event` class, with its constants, used within the event-chain-model of the Java Version 1.0, has nothing to do with the delegation event-model beginning with Java Version 1.1, which is treated here.

KeyEvent-objects (`java.awt.event`) come from keyboard usage by the end-user. Such event-objects contain key-codes, the key-character and a descriptive text of the key. Constants of primitive-type `int`, like `KEY_PRESSED`, `KEY_RELEASED` and `KEY_TYPED` specify what kind of key-event is transmitted.

MouseEvent-objects (`java.awt.event`) codify the event of the mouse-pointer entering or exiting the on-screen area of the screen-component and the pressing, releasing clicks of the mouse-key(s). This is done by coding a member-variable of primitive-type `int` with constant-masks like `MOUSE_ENTERED`, `MOUSE_EXITED`, `MOUSE_PRESSED`, `MOUSE_RELEASED`, `MOUSE_CLICKED`. A **MouseEvent**-object ALSO CONTAINS the position of the mouse-pointer at the time the event occurred. A mouse-event may be also part of a stream of mouse-events indicating a mouse-movement, with identifiers like `MOUSE_MOVED` or `MOUSE_DRAGGED`.

ChangeEvent-objects (`javax.swing`) are used as a simple notification of unspecified changes. For example, a kind of such event-object may be used by the Java Runtime Environment to send timer-signals in regular intervals (see `java.swing.Timer` on page 603).

Other **general-purpose event-types** may nevertheless be context dependent:

ActionEvent-objects (`java.awt.event`) contain a string identifying the specific action which has occurred.

PropertyChangeEvent-objects (`java.beans`) are used for transferring information similarly like the **ActionEvent**. But here the information is given by its new and old values and a property-name.

Event-objects may also be used for notifying of a change that doesn't need to have an (immediate) relation to an external input:

ContainerEvent-objects may inform about the container being added to or removed from a component, the rest like layouting is handled automatically by the Java Runtime Environment (JRE). In that sense **ContainerEvent**-objects can be used by an application-programmer for purely informational purposes.

PaintEvent-objects may be used by the Java Runtime Environment to internally communicate the repainting of parts of a window which has been overlapped by some other window. Application-programmers usually don't work with **PaintEvent**-objects. Application-programmers just override the `paintComponent()`-method of a Swing-component (or the `paint()`-method of an AWT-component), the rest is done automatically by the Java Runtime Environment. (Since they have no listener-classes, **PaintEvent**-objects are not used with the delegation event-model!)

A simple classification-scheme for event types give the following sentences: Event-types which are un-specific or independent of the type of screen-component are called **context-independent event-classes**. Other events, only to be sent by a source of a certain type, are called **contextual event-types** — like an **ActionEvent** sent by a **JBUTTON**-object, to be handled by an `actionPerformed()`-method.

As already indicated on page 573, such a number of input-interpretations and internal-state-changes would easily exhaust the operating-system or any runtime-system in event-managing because of the wealth of potential drains and potentially constructible types of event-objects. Under such circumstances, the switching-infrastructure couldn't target the relevant references to really interested listeners. A proper event-generating- and forwarding-infrastructure with explicitly registered drains or handlers makes this diversity surprisingly easy to handle: Only those event-objects needed for the registered listeners have to be generated. For a general view on Java event-subscriber processing, unextended by background-information, see the prototypes in the section beginning with page 276 and the text on Java-Beans following page 311.

Java: Sources, Event-Objects, Sinks (Listeners)

Java: Sources, Event-Objects, Sinks (Listeners) — The graphic following page 573 illustrates the AWT-specific implementation of the delegation event-model. After having registered a drain- or listener-object with the source's `add...Listener()`-method, event-objects can be dispatched from the source to the drain and be processed there by handler-methods. Below, the two tables list a collection of controlling

entities (source, event, drain, handler) for differing types of such communication-processes. The first table contains only cases that work with context-independent events. This can be easily seen by checking the “SOURCE”-column, it shows only sources of type `java.awt.Component` (or extensions).

Component-unspecific CONTEXT-INDEPENDENT			
SOURCE	EVENT	DRAIN	HANDLER-method
	<code>java.awt.event.</code>	<code>java.awt.event.</code>	<code>public void</code>
<code>java.awt.Component</code> and extensions	<code>MouseEvent</code>	<code>MouseListener</code> <code>MouseMotionListener</code>	<code>mouseEntered(...)</code> <code>mouseExited(...)</code> <code>mousePressed(...)</code> <code>mouseReleased(...)</code> <code>mouseClicked(...)</code> <code>mouseMoved(...)</code> <code>mouseDragged(...)</code>
<code>java.awt.Component</code> and extensions	<code>KeyEvent</code>	<code>KeyListener</code>	<code>keyPressed(...)</code> <code>keyReleased(...)</code> <code>keyTyped(...)</code>
<code>java.awt.Component</code> and extensions	<code>ComponentEvent</code>	<code>ComponentListener</code>	<code>componentResized(...)</code> <code>componentMoved(...)</code> <code>componentShown(...)</code> <code>componentHidden(...)</code>
<code>java.awt.Component</code> and extensions	<code>FocusEvent</code>	<code>FocusListener</code>	<code>focusGained(...)</code> <code>focusLost(...)</code>
Components (*) with <code>fireStateChanged()</code>	<code>javax.swing.event.</code> <code>ChangeEvent</code>	<code>javax.swing.event.</code> <code>ChangeListener</code>	<code>stateChanged(...)</code>
<code>java.awt.Component</code> and extensions	<code>PaintEvent</code>	(same component, non-explicit)	<code>paint(Graphics g)</code> see also page 595

(*) For a listing of all classes or interfaces, known to be declaring the method `fireStateChanged()`, see the index of the   Java Platform API Specification [JPAPIS].

Observe that one mouse-event-object can service two different listener types and multiple event-handler-methods. This is possible because each `java.awt.AWTEvent`-object above uses a value (named `id`) of the primitive-type `int` to further specify its state: The `MouseEvent`-class furnishes several integer constants `MOUSE_ENTERED` (=504), `MOUSE_EXITED` (=505), `MOUSE_PRESSED` (=501), `MOUSE_RELEASED` (=502), `MOUSE_CLICKED` (=500), `MOUSE_MOVED` (=503) and `MOUSE_DRAGGED` (=506). A regular `MouseEvent`-object has an `id`-value of the kind above, according to which Java selects the handler-method to call. This routing of event-objects is done automatically by the Java Runtime Environment, THE PROGRAMMER ONLY NEEDS TO IMPLEMENT THE HANDLER-METHODS, FURNISH THE SOURCE-OBJECTS AND REGISTER THE LISTENER(S).

Remember also the relation between the notions “focus of screen-components” and “activation of windows” (page 509): **Activated windows** always are top-level windows on the desktop and can take input. **Screen-components are on focus**, they can take input, if they are child-components of an activated window and have been clicked with the mouse or are pointed at with the mouse to make them take the input. If the window is deactivated, usually by not being top-level any more, then usually the focus has been temporarily lost; after the window has been activated again, the focus is usually re-established automatically and immediately.

programmer's
tasks in a fully
implemented
event-
infrastructure
activation and
focus

Some events like mouse-clicks or keyboard-signals occur very slowly, compared to the processing speed of the computer. But there are cases of high-frequency event-dispatching having to be considered: For example, mouse-movements on sliders may emit events with a relatively high frequency. High-frequency event-firing may cause performance-problems. — Suggested solution: A change-notification with a resource-economizing `javax.swing.event.ChangeEvent`, the program itself may decide on a callback for getting an event-object which encapsulates a complete description of the new state. (Anticipated in the source-code on page 547.)

Most parts of these tables are left uncommented at this point. To a certain extent, the entries in the table should be self-explaining and they should give a preparation for consulting the online-documentation.

Below has been listed a selection of component-specific ways of event-handling, notice that the sources of the event-objects are of a rather specialized class-type, if compared to the class `java.awt.Component`:

Component-SPECIFIC CON-TEXTUAL EVENT HANDLING SOURCE	EVENT		
java.awt.	java.awt.event.	java.awt.event.	HANDLER-method
Components (*) with firePropertyChange()	java.beans.PropertyChangeEvent	java.beans.PropertyChangeListener	propertyChange(...)
Container and extensions	ContainerEvent	ContainerListener	componentAdded(...) componentRemove(...)
Window and extensions	WindowEvent	WindowListener	windowOpened(...) windowClosed(...) windowClosing(...) windowIconified(...) windowDeiconified(...) windowActivated(...) windowDeactivated(...)
ItemSelectable	ItemEvent	ItemListener	itemStateChanged(...)
List	ListSelectionEvent	ListSelectionListener	valueChanged(...)
Adjustable	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged(...)
	ActionEvent	ActionListener	actionPerformed(...)
TextComponent, TextArea	TextEvent	TextListener	textValueChanged(...)

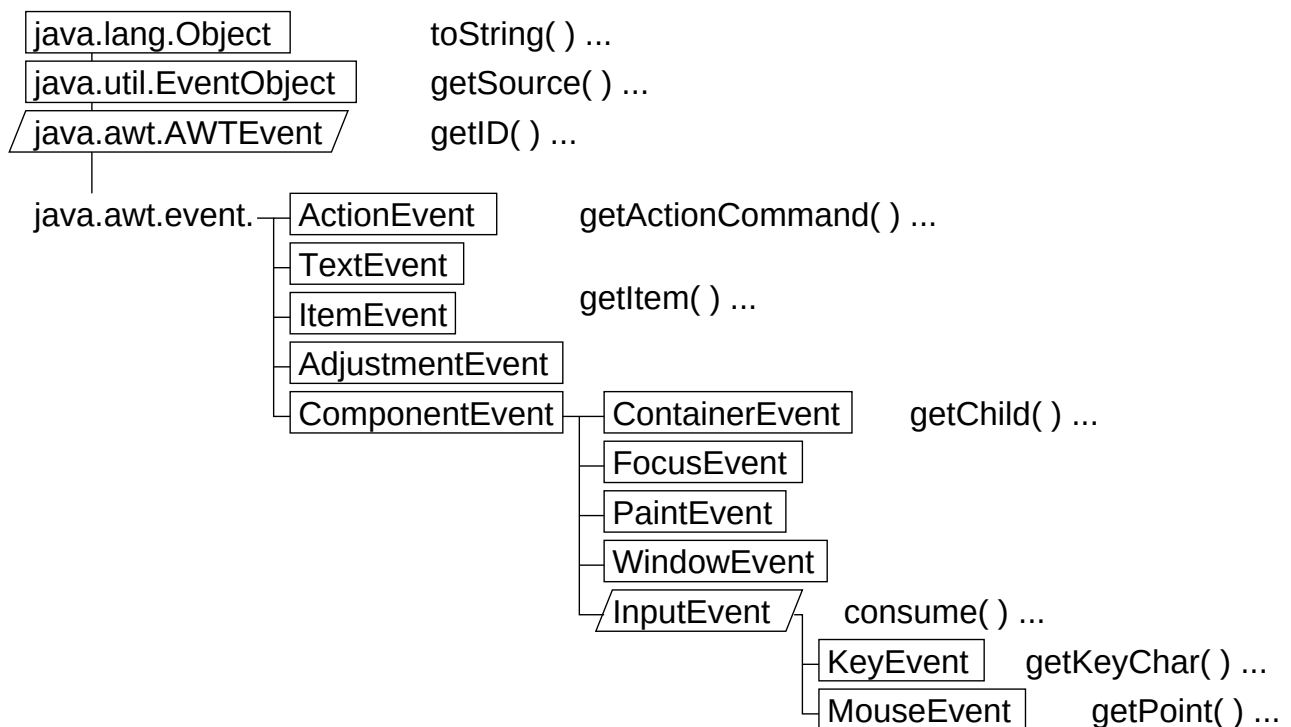
(*) For a listing of known classes or interfaces declaring the method `firePropertyChange()`, see the index of the  Java Platform API Specification [JPAPIS]. `PropertyChangeEvent`-objects are meant to be thrown in a Java-Bean context (around page 315).

contextual event
= semantic event

Contextual events often are also called “semantic events” because, for example, all the original keyboard- and mouse-actions are heavily processed to yield an event-object particularly suited to the needs of the given screen-component (click, double-click, drag, instead of mere “mouse-up”-, “mouse-down”- and position-signaling).

The class-hierarchy of the most commonly used event-types has the following form:

Hierarchy of Event Classes in the Application Window Toolkit



Remember that a `MouseEvent`-object can be further specified by receiving an `id`-value from the event-source. This feature has already been installed with the definition of the abstract class `java.awt.AWTEvent`.

Any extension of the abstract class `java.awt.AWTEvent` is free to furnish a list of more specific integer-constants. An event-object of that type, then can be further scrutinized by considering its `id`-value. So AWT-EVENT-OBJECTS IDENTIFY THEMSELVES BASICALLY IN TWO WAYS: How they extend the class `java.awt.AWTEvent` and, what `int`-value takes their `id`-variable. If properly defined, these `id`-values also may be used as masks for discerning events binarily.

AWTEvent-object
= class-type +
id-value

In the graphic above, the new abstract class `java.awt.event.InputEvent` contains the codifications of pressed secondary mouse-keys and other special keyboard-keys like CTRL and ALT or ALTGR.

This hierarchy of event-classes is complemented by associated listeners, as indicated in the tables above, but there exists one listener which receives all events that are exchanged within the Java Runtime Environment, an implementation of the `java.awt.event.AWTEventListener`. Without registration to any source, its handler-method `eventDispatched()` is called each time anywhere an event is dispatched within the current Java Runtime Environment (JRE). The following class shows how such a listening device can be installed in a Java program:

```
class BigEar implements java.awt.event.AWTEventListener
{ public static final long ALL_EVENTS = java.lang.Long.MAX_VALUE;
  BigEar()
  { java.awt.Toolkit.getDefaultToolkit().addAWTEventListener( this, ALL_EVENTS );
  }
  public void eventDispatched( java.awt.AWTEvent e )
  { System.out.println( e.toString() );          // What to do with the events received.
  }
  public static void main( String [] unused )
  {
    new BigEar();
    new TestFrame( "Do something on me" );
    System.out.println( "main ended, still listening" );
  }
}
class TestFrame extends java.awt.Frame
{ TestFrame( String s )
  { super(s); enableEvents( BigEar.ALL_EVENTS ); setSize(200,200); setVisible(true);
  }
}
/*
jacobneuhaus@linux:~/JavaCodeFiles/SingleUserInterface/Events > javac BigEar.java
jacobneuhaus@linux:~/JavaCodeFiles/SingleUserInterface/Events > java BigEar
java.awt.event.ComponentEvent[COMPONENT_RESIZED (0,0 200x200)] on frame0
java.awt.event.HierarchyEvent[HIERARCHY_CHANGED (DISPLAYABILITY_CHANGED, TestFrame[frame0,0,0,200x200,invalid,hidden,
layout=java.awt.BorderLayout,resizable,title=Do something on me,null]) on frame0
java.awt.event.HierarchyEvent[HIERARCHY_CHANGED (SHOWING_CHANGED,TestFrame[frame0,0,0,200x200,layout=java.awt.BorderLayout,
resizable,title=Do something on me,null]) on frame0
java.awt.event.ComponentEvent[COMPONENT_RESIZED (4,18 202x201)] on frame0
java.awt.event.WindowEvent[WINDOW_ACTIVATED] on frame0
java.awt.event.FocusEvent[FOCUS_GAINED,permanent] on frame0
main ended, still listening
java.awt.event.ComponentEvent[COMPONENT_SHOWN] on frame0
java.awt.event.WindowEvent[WINDOW_OPENED] on frame0
java.awt.event.PaintEvent[PAINT,updateRect=java.awt.Rectangle[x=200,y=0,width=1,height=200]] on frame0
java.awt.event.PaintEvent[UPDATE,updateRect=java.awt.Rectangle[x=0,y=0,width=202,height=201]] on frame0
java.awt.event.MouseEvent[MOUSE_ENTERED,(200,118),mods=0,clickCount=0] on frame0
java.awt.event.MouseEvent[MOUSE_MOVED,(199,118),mods=0,clickCount=0] on frame0
...
java.awt.event.MouseEvent[MOUSE_MOVED,(197,120),mods=0,clickCount=0] on frame0
java.awt.event.MouseEvent[MOUSE_PRESSED,(197,120),mods=16,clickCount=1] on frame0
java.awt.event.MouseEvent[MOUSE_RELEASED,(197,120),mods=16,clickCount=1] on frame0
java.awt.event.MouseEvent[MOUSE_CLICKED,(197,120),mods=16,clickCount=1] on frame0
java.awt.event.MouseEvent[MOUSE_PRESSED,(197,120),mods=16,clickCount=1] on frame0
java.awt.event.MouseEvent[MOUSE_DRAGGED,(197,121),mods=16,clickCount=0] on frame0
java.awt.event.MouseEvent[MOUSE_DRAGGED,(197,122),mods=16,clickCount=0] on frame0
java.awt.event.MouseEvent[MOUSE_RELEASED,(197,122),mods=16,clickCount=0] on frame0
java.awt.event.MouseEvent[MOUSE_PRESSED,(197,122),mods=16,clickCount=1] on frame0
java.awt.event.MouseEvent[MOUSE_RELEASED,(197,122),mods=16,clickCount=1] on frame0
java.awt.event.MouseEvent[MOUSE_CLICKED,(197,122),mods=16,clickCount=1] on frame0
java.awt.event.MouseEvent[MOUSE_MOVED,(198,122),mods=0,clickCount=0] on frame0
...
java.awt.event.MouseEvent[MOUSE_EXITED,(201,125),mods=0,clickCount=0] on frame0
java.awt.event.ComponentEvent[COMPONENT_RESIZED (4,18 209x207)] on frame0
java.awt.event.PaintEvent[PAINT,updateRect=java.awt.Rectangle[x=1,y=0,width=201,height=201]] on frame0
java.awt.event.PaintEvent[PAINT,updateRect=java.awt.Rectangle[x=1,y=0,width=207,height=206]] on frame0
java.awt.event.PaintEvent[UPDATE,updateRect=java.awt.Rectangle[x=0,y=0,width=209,height=207]] on frame0
java.awt.event.WindowEvent[WINDOW_DEACTIVATED] on frame0
java.awt.event.FocusEvent[FOCUS_LOST,temporary] on frame0
jacobneuhaus@linux:~/JavaCodeFiles/SingleUserInterface/Events >
*/
```

In this context reconsider also that Swing-components of type `javax.swing.JComponent` or an extension have a member-variable called `listenerList` of the type `javax.swing.event.EventListenerList`. This member-variable returns the reference to a list of all listener-objects which are currently registered to this screen-component!

Summary: Java Variants of Event-Listening

Summary: Java Variants of Event-Listening — This section collects the event-listening structures as they have been introduced in the programming examples beginning on page 544. On the previously mentioned

page, the source-code which demonstrates the use of JButton-screen-components (ButtonDemo, AButtonActionListener), exemplifies the following scheme: Different objects represent source and listener as described by the pseudo-source-code below.

```
class sourceA // screen-component or else
{ EventTypeListenerInterface l = new Listener();
    // observe the implicit type-cast!
    sourceA.addEventTypeListener( l );
}
class Listener implements EventTypeListenerInterface
{ handlerMethod( EventType e ){ implementation } }
```

The listener- or drain-object may also be instantiated – as an anonymous object – during registration (seen in the source-code which demonstrates the use of list-components, on page 545 (ListDemo, SomeListSelectionListener)):

```
class sourceB // screen-component or else
{ sourceB.addEventTypeListener( new CustomListener() )
}
class CustomListener implements EventTypeListenerInterface
{ handlerMethod( someEventType e ){ customImplementation } }
```

Or, as demonstrated on page 546 (MenuDemo), the source-object handles its own event-objects, by implementing the listener-interface itself and by registering itself as a listener. This may be used within smaller pieces of source-code, where it is not necessary to explicitly separate source and drain of the event.

```
class sourceC implements EventTypeListenerInterface
{ screenComponent.addEventTypeListener( this )
    handlerMethod( someEventType e ){ customImplementation }
}
```

Finally, a possibility that might turn out to be very confusing when trying to re-read it in a larger programming context: Here the listener-object is defined AND instantiated during the registration-process; thus the listener and its type, *both* remain anonymous:

```
screenComponent.addEventTypeListener(
    new EventTypeListenerInterface()
    { handlerMethod( someEventType e ){ customImplementation } }
)
```

Event-handling constructs of the kind above, still may occur in variations, for example by using adapter-classes, as described in the following section. — In those special cases where the source does not support producing event-objects for which a method add**EventType**Listener() can be called (producing Java compiler errors like: Method add**EventType**Listener(**EventType**Listener) not found in class source.), the Java programmer has to furnish a triggering structure of its own, this is shown on page 591.

Java: Adapter-Classes

Java: Adapter-Classes — The notion of an adapter-class has been already introduced on page 190. Adapter-classes extend listener-interfaces by IMPLEMENTING THE DECLARED HANDLER-METHODS WITH TRIVIAL CODE “{ }”. Why may that be useful? A Java programmer can simply extend an adapter-class by solely implementing, thus overriding, the handler-method of interest. The Java programmer does not need to rewrite the empty implementations of all the other handler-methods. The mechanics are shown below in pseudo-source-code:

```
interface exampleListener
{ methodSignature1 // no implementation!
  methodSignature2
  ...
  methodSignatureN }
```



```

}
abstract class exampleAdapter extends exampleListener
{ methodSignature1{} // empty implementation!
  methodSignature2{}
  ...
  methodSignatureN{}
}

```

To avoid a direct instantiation of an adapter-class (without any overridden method) which would produce an equally trivial object, ADAPTER-CLASSES ARE USUALLY DECLARED **abstract**. This encourages the programmer to extend them non-trivially.

The advantage of an adapter-class over the (listener-)interface can be reformulated as follows: Implementation of an interface requires to define all N methods above; whereas the extension and instantiation of the adapter-class makes it possible to override only those methods of interest (the other methods remain defined with an EMPTY-BLOCK). The source-code on page 600 gives an example of using the pre-written adapter-class `java.awt.event.MouseAdapter`.

Adapters have a disadvantage: Adapter-classes have to be the base-class of the actual implementing class. This problem often is overcome by using inner, anonymously instantiated instances of the adapter-class:

```

source.addListener(
    new exampleAdapter(){ methodSignature2{ implementation } } )

```

This is short, but often incomprehensible to the untrained or unaccustomed eye; its use is not to be overly encouraged. But as an example consider a piece of Java source-code which activates the window-closing-box of a screen-component (identifier: `window`). This piece of Java source-code implements the interface `java.awt.event.WindowListener` by instantiating an adapter:

```

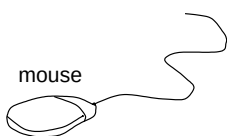
window.addWindowListener
( new java.awt.WindowAdapter()
  { public void windowClosing( java.awt.event.WindowEvent e )
    { System.exit( 0 ); } }
)

```

The package `java.awt.event` furnishes a number of pre-written adapter-classes: There are interface/-adapter pairings like `KeyListener/KeyAdapter`, `MouseListener/MouseAdapter`, `MouseMotionListener/MouseMotionAdapter` and more to be found when, for example referencing the ➡☐ ➡☐ Java Platform API Specification [JPAPIS]²⁶¹.

Java: Examples for Mouse- and Keyboard Event Handling

Java: Examples for Mouse- and Keyboard Event Handling — The processing of mouse- and keyboard-input in Java is based on the Java delegation event-mechanism. First, an end-user has to move the mouse, so that the mouse-pointer hovers over the chosen screen-component. In some operating-systems, this already puts the screen-component in focus. In most operating-systems, a single click of the mouse-key brings the screen-component into focus. Thus the screen-component has been marked to take further input; it is ready to channel mouse movements, mouse-clicks, and keyboard-inputs to a program's handler-method. The following ignores the event-processing on the levels of the operating-system and the Java Runtime Environment (JRE) and just considers the Java programmer's task of handling the events which have been processed into `MouseEvent`-objects and `KeyEvent`-objects.



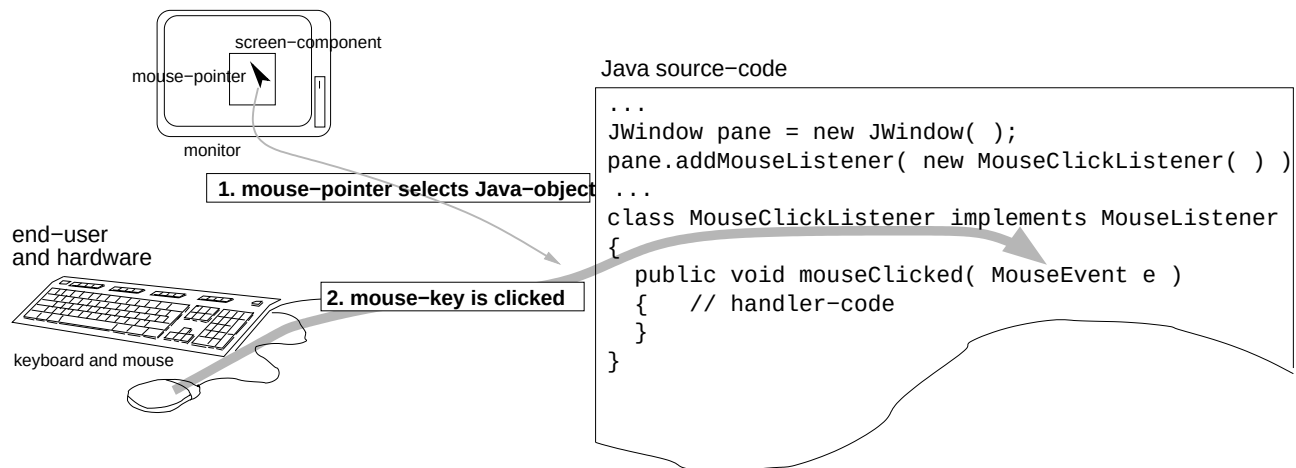
Remember that a mouse needs only one key to effectively communicate with the computer-platform. Often the functionality of the mouse can be extended by pressing the keyboard's CTRL- or ALT-keys simultaneously with the mouse-key. With two- or three-keyed mice these key combinations are coded into the auxiliary mouse-keys.

- The cases of pressing auxiliary mouse-keys are treated in the Java Runtime Environment by changing the state of the `MouseEvent` so that inherited methods of `java.awt.event.InputEvent` give the proper signaling (`isAltDown()`, `isControlDown()`, `isMetaDown()`, `isAltGraphDown()`, `isShiftDown()`). See the source-code below for more.

²⁶¹See also the Java TechTip of May 9th, 2000.

- Or if a direct identification of the mouse-key is needed, test the `java.awt.event.InputEvent` modifier-mask (`getModifiers()` returns an `int`-value used as a binary mask) against the constants `InputEvent.BUTTON1_MASK` (the left mouse-key), `InputEvent.BUTTON2_MASK` (the main- or middle-mouse-key) or `InputEvent.BUTTON3_MASK` (the right-mouse-key). See the source-code below for more.

Java's `MouseEvent`-object even identifies double or triple mouse-click-sequences! (Thus already featuring non-generic processing of mouse-events!)



The class "Mouse" implements the structure indicated in the graphic above for mouse-events and for the Java Runtime-generated `MouseEvent`-objects. Furthermore are demonstrated the different ways to analyze the key-presses of the mouse-key(s):

```

import javax.swing.JWindow;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.InputEvent;
import java.awt.event.MouseEvent;

class Mouse
{
    public static void main( String unused[] )
    {
        JWindow pane = new JWindow();
        pane.addMouseListener( new MouseClickListener() );
        pane.addMouseMotionListener( new MovingMouseListener() );
        pane.setSize( 200, 200 );
        pane.setVisible( true );
    }
}

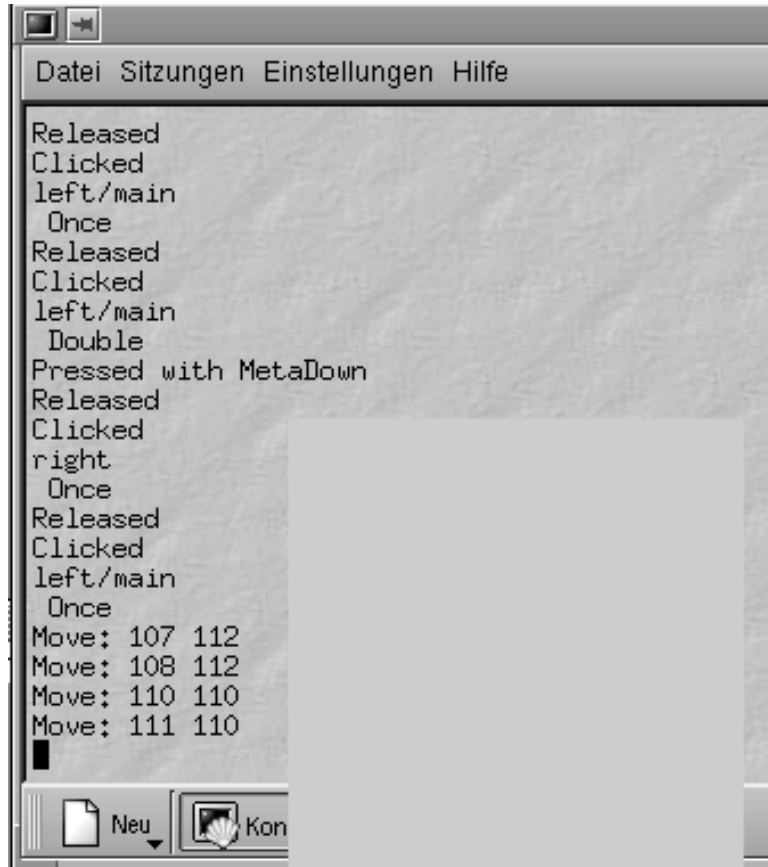
class MouseClickListener implements MouseListener
{
    private void out( String text ){ System.out.println( text ); }
    public void mouseClicked( MouseEvent e )
    {
        out("Clicked");
        if( ( e.getModifiers() & InputEvent.BUTTON1_MASK ) !=0 ) out( "left/main" );
        if( ( e.getModifiers() & InputEvent.BUTTON2_MASK ) !=0 ) out( "middle/main" );
        if( ( e.getModifiers() & InputEvent.BUTTON3_MASK ) !=0 ) out( "right" );
        switch( e.getClickCount() )
        {
            case 1: out(" Once"); break;
            case 2: out(" Double"); break;
            case 3: out(" Triple");
        }
    }
    public void mouseEntered( MouseEvent e ){ out( "Entered" ); }
    public void mouseExited( MouseEvent e ){ out( "Exited" ); }
    public void mousePressed( MouseEvent e )
    {
        if( e.isAltDown() ) out( "Pressed with AltDown" );
        if( e.isControlDown() ) out( "Pressed with ControlDown" );
        if( e.isMetaDown() ) out( "Pressed with MetaDown" );
        if( e.isAltGraphDown() ) out( "Pressed with AltGraphDown" );
        if( e.isShiftDown() ) out( "Pressed with ShiftDown" );
    }
    public void mouseReleased( MouseEvent e ){ out( "Released" ); }
}

```

```

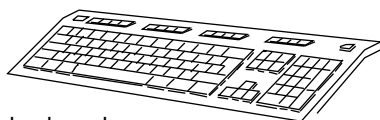
class MovingMouseListener implements MouseMotionListener
{ public void mouseMoved( MouseEvent e )
  { System.out.println( "Move: " + e.getX() + " " + e.getY() );
  }
  public void mouseDragged( MouseEvent e )
  { System.out.println( "Drag: " + e.getX() + " " + e.getY() );
  } }
/* Click + ALT (ALT or ALTGR) = middle mouse-key, if existent.
   Click + META (Apple key) = right mouse-key, if existent.
   Click + SHIFT = left mouse key or main mouse-key
*/
/*
jacobneuhau@linux:~/JavaCodeFiles/SingleUserInterface/Events > javac Mouse.java
jacobneuhau@linux:~/JavaCodeFiles/SingleUserInterface/Events > java Mouse
Entered
Move: 197 149
Move: 196 149
Move: 195 149
Move: 194 149
Move: 193 149
Move: 192 149
Move: 191 149
Drag: 190 149
Drag: 189 149
Drag: 188 149
Drag: 187 149
Drag: 187 150
Released
Pressed with MetaDown
Drag: 187 151
Drag: 187 152
Drag: 187 153
Drag: 186 153
Released
Pressed with AltDown
Drag: 185 153
Drag: 183 155
Drag: 183 156
Drag: 181 158
Drag: 181 159
Drag: 181 160
Drag: 181 161
Drag: 182 161
Drag: 182 162
Released
Move: 183 162
Move: 184 162
Move: 184 161
Move: 185 161
Move: 186 161
Move: 187 161
Move: 189 159
Move: 191 157
Move: 192 157
Move: 193 157
Move: 194 157
Move: 195 157
Move: 196 157
Move: 197 157
Move: 198 157
Move: 199 157
Exited
(Ctrl-C in terminal-window)
jacobneuhau@linux:~/JavaCodeFiles/SingleUserInterface/Events >
*/

```



In the screen-printout above, the gray rectangle represents the JWindow-screen-component which has to be a top-level screen component to receive the mouse-movements and mouse-actions. Some general remarks on the programming style: As can be seen above, standard procedural coding appears only when putting the screen-component on-screen and when writing the handler-methods! Clustering if-statements like the one above may be avoided in favor of nesting if-statements, with the most often occurring case being in the outer if-statement. This may speed processing.

In this context remember also the sections on adapters: The classes `java.awt.event.MouseAdapter` and `java.awt.event.MouseMotionAdapter` make it possible to implement only those event-handler-methods that are actually needed. For example, extending the `MouseListener`-class may override only the handler-method `mouseClicked()` and automatically leave any other handler-method empty.



keyboard

Keyboard input usually requires a screen-component which processes keyboard activities like an editable text-component. On focus, those screen-components may throw `java.awt.event.KeyEvent`-objects. Those objects also present the original keyboard-signals heavily pre-processed, to ease the handling:

- A `KeyEvent`-object has two specific properties called `keyCode` and `keyChar`. The public instance-method `KeyEvent.getKeyChar()` returns a value of the primitive-type `char` which gives the Unicode-representation of the key and can be printed into the terminal-window or onto the screen-component.
- Whereas the method `KeyEvent.getKeyCode()` returns a value of the primitive-type `int` which represents a “virtual keycode”. The coded symbol represented by that value is identifiable with a key-constant like `KeyEvent.VK_A`, `KeyEvent.VK_SPACE` even `KeyEvent.VK_EURO_SIGN`. The texts of these “virtual key-codes” is returned by the method `KeyEvent.getKeyText(int keyCode)`. The probably most updated list of all virtual keycodes can be seen in the online-documentation ➡ [Java Platform API Specification \[JPAPIS\]](#), which is downloadable from the Internet.

The source-code below should illustrate these statements:

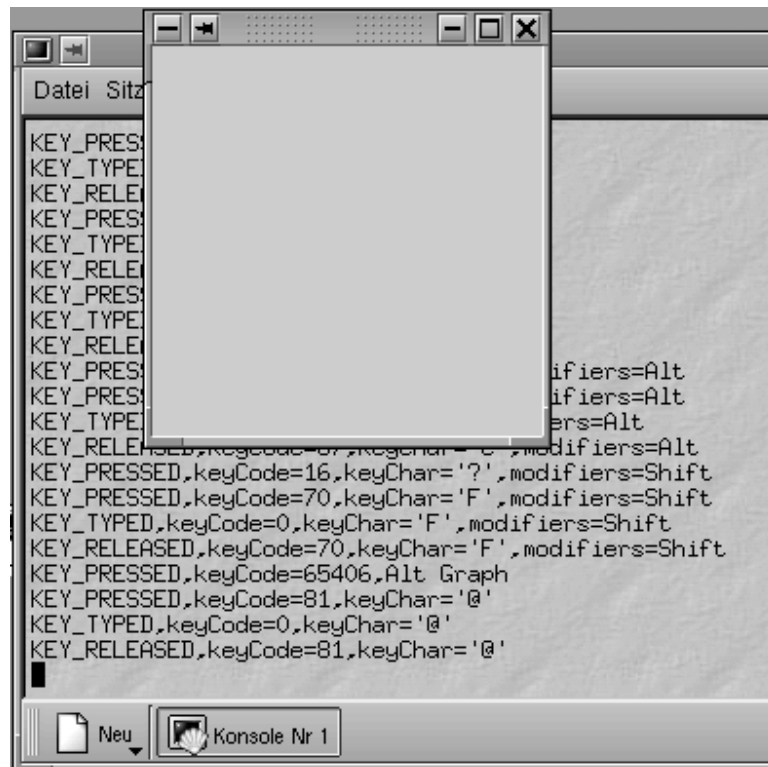
```
import javax.swing.JFrame;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
```

```

class Keyboard
{
    public static void main( String unused[] )
    {
        JFrame window = new JFrame();
        window.addKeyListener( new KeyboardListener() );
        window.setSize( 200, 200 );
        window.setLocation( 0, 0 ); // may be overridden by front-end of OS (KDE of Linux)
        window.setVisible( true );
    } }
class KeyboardListener implements KeyListener
{
    public void keyTyped( KeyEvent e ){ out(e); }
    public void keyPressed( KeyEvent e ){ out(e); }
    public void keyReleased( KeyEvent e ){ out(e); }
    private void out( KeyEvent e ){ System.out.println( e.paramString() ); }
} }
// make the Java window the top-level window by clicking it and press keyboard:
/*
jacobneuhau@linux:~/JavaCodeFiles/SingleUserInterface/Events > javac Keyboard.java
jacobneuhau@linux:~/JavaCodeFiles/SingleUserInterface/Events > java Keyboard
KEY_PRESSED, keyCode=17, keyChar='?', modifiers=Ctrl
KEY_RELEASED, keyCode=17, keyChar='?', modifiers=Ctrl
KEY_PRESSED, keyCode=16, keyChar='?', modifiers=Shift
KEY_PRESSED, keyCode=75, keyChar='K', modifiers=Shift
KEY_TYPED, keyCode=0, keyChar='K', modifiers=Shift
KEY_RELEASED, keyCode=75, keyChar='K', modifiers=Shift
KEY_PRESSED, keyCode=18, keyChar='?', modifiers=Alt
KEY_RELEASED, keyCode=18, keyChar='?', modifiers=Alt
KEY_PRESSED, keyCode=65406, Alt Graph
KEY_PRESSED, keyCode=81, keyChar='@'
KEY_TYPED, keyCode=0, keyChar='@'
KEY_RELEASED, keyCode=81, keyChar='@'
KEY_PRESSED, keyCode=17, keyChar='?', modifiers=Ctrl
KEY_PRESSED, keyCode=67, keyChar=" ", modifiers=Ctrl
KEY_TYPED, keyCode=0, keyChar=" ", modifiers=Ctrl
KEY_RELEASED, keyCode=67, keyChar=" ", modifiers=Ctrl
(Make the terminal-window the top-level window, by clicking it and enter Ctrl-C to terminate)

jacobneuhau@linux:~/JavaCodeFiles/SingleUserInterface/Events >
*/

```



Observe that one key-activation produces three `KeyEvent`-objects with identifiers: `KEY_PRESSED`, `KEY_TYPED`, `KEY_RELEASED`. Observe also the apparent bug in Java that lets the `keyCode` of the `KEY_TYPED` event be zero.

Positioning and printing a character on the screen and the choice of font is the task of software for managing the output or the `VIEW`; the font-related source-code on page 569 can be seen as a simple example of such a `VIEW`-part of a program. Here have been presented solely input-related phenomena.

AWT-Switchboarding and Java

AWT-Switchboarding and Java — Remember that the delegation event-model is implemented only partially within the `java.util`-package (page 286). There, the programmer is responsible for organizing the channeling the information of the `java.util.EventObject`-objects (switchboarding). — This section gives a superficial impression of the switchboard-mechanism in general and the ways of interaction within the AWT-packages.

The Java Runtime System, beginning with Java Version 1.1, implements the so-called delegation event-model (page 276) in the AWT²⁶²-context, namely the `java.awt` and `java.awt.event`-packages: Source and drain are intended to be found in different classes AND event-objects are only dispatched if the drains (now called listeners) are registered by the programmer with the potential event-emitting sources. Only after registration follows a notification with a reference of an event-object (also called a callback-concept or callback-model). Thus, the switchboard-infrastructure of the Java Runtime Environment and the `java.awt/java.awt.event`-packages generates event-objects only when the associated listeners have been registered; this economizes greatly on resources for object-creation. Remember the graphic on page 573, for an illustration of the registering of listeners.

So, the `java.awt/java.awt.event`-packages, fully implement the delegation event-model. Event-objects, instantiated from pre-written extensions of the abstract type `java.awt.AWTEvent`, find an automated switchboarding-mechanism and fully-functional methods for registering listener-objects. Only the event-handler-methods of the listener-objects have to be overridden to act according to the programmer's wish, as could be seen in some of the examples above.

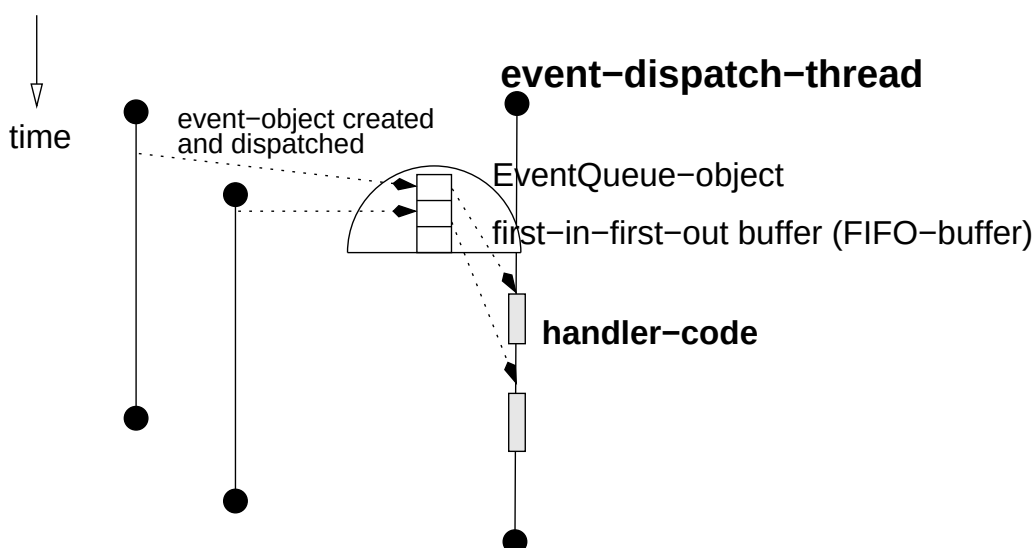
The channeling of the event-information is supposed to happen independently of other proceedings within the Java Runtime Environment (JRE). Therefore, switching the references of the event-objects to reach their listeners and the handling of the events happens in a thread of its own: The AWT-implementation of the delegation event-model includes an own switchboard-thread or a so-called **event-dispatch-thread** which GATHERS THE INCOMING REFERENCES TO EVENT-OBJECTS AND SORTS THEM OUT FOR HANDLING BY THE ASSOCIATED LISTENERS. This event-dispatch-thread usually uses an array of changeable length (`java.util.Vector`) as a buffer or queue for the event-objects — **event-queue** — to dispatch the reference of each event-object orderly to the registered associated listener(s).

(Event-Dispatch-Thread and Event-Queue) Each Java Runtime Environment implements exactly one event-dispatch-thread and one `EventQueue`-object for routing and handling references to `AWTEvent`-objects. When the event-queue processes a reference to an `AWTEvent`-object, then the `EVENT-HANDLER-METHODS`, BY DEFAULT, RUN IN THE `EVENT-DISPATCH-THREAD`! The event-dispatch-thread is a system-thread that starts with the system-start and stops when the system stops, a so-called daemon-thread as introduced on page 223.

event-dispatch-thread

event-handler-methods, by default, run in the event-dispatch-thread!

AWT-Switchboarding



A given piece of source-code can test whether it is running in the event-dispatch-thread by calling

```
boolean thisCodeIsRunningInTheEventDispatchThread =
```

²⁶²Abstract Window Toolkit

```
javax.swing.SwingUtilities.isEventDispatchThread();
```

Within a standard handler-method (`paintComponent()` or `mouseClicked()`), calling this static method should produce the return-value `true`!

(For an example see the source-code on page 538.) Java programmers can control some aspects of the event-queue object (class-type `java.awt.EventQueue`): A reference to that already existing object can be got by calling the factory-like method

```
java.awt.Toolkit.getSystemEventQueue()
```

The instance-methods which allow to get references of the event-objects inside the queue (`peekEvent()`) or to directly put an event onto the queue (`postEvent()`), as well as a method to remove an event-object from the queue (`getNextEvent()`) are documented in the   Java Platform API Specification [JPAPIS].

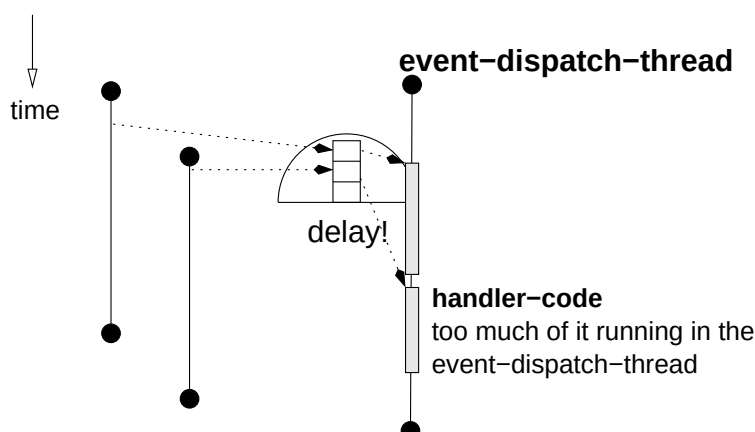
There exists another way to access the relative core of the processing of AWTEvent-objects: An instance of the `java.awt.event.AWTEventListener`-class, specially registered with a static Toolkit-method (`java.awt.Toolkit.addAWTEventListener()`) is notified about all events that are processed by the event-queue in the event-dispatch-thread (as demonstrated by the Java source-code on page 579).

Using the Event-Dispatch-Thread

Using the Event-Dispatch-Thread — Recall that, `java.awt.AWTEvent`-objects²⁶³ find a completely implemented routing-infrastructure in their Java Runtime Environment (JRE). Such a Java Runtime Environment automatically uses a single thread, the so-called event-dispatch-thread to route references of AWTEvent-objects and run the associated handler-code (`mouseMoved()`, `mouseDragged()`, `keyTyped()`). Specifically, the incoming references of AWTEvent-objects are buffered by the `java.awt.EventQueue`-object and are used to successively call and RUN THE ASSOCIATED HANDLER-METHODS WITHIN THE EVENT-DISPATCH-THREAD.

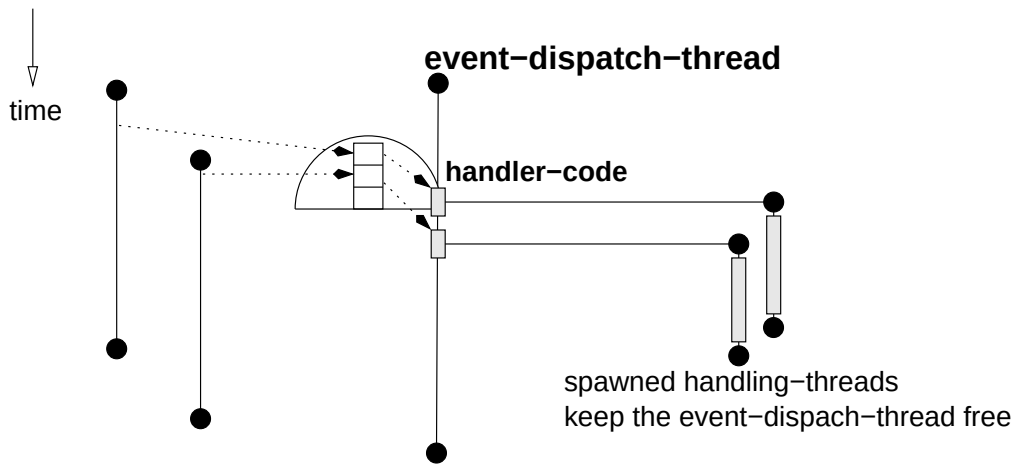
WHEN SHOULD SOURCE-CODE RUN IN THE EVENT-DISPATCH-THREAD? To minimize event-routing delays, generally handler-code should be kept short and little resource-consuming. Code pieces that are run sequentially in the event-dispatch-thread, cannot parallelly interfere with each other (to produce a race-condition) when manipulating the Graphical User Interface on-screen. — In the graphic below is shown how an oversized handler-method blocks the event-dispatch-thread:

handler-code:
short and
resource
sparing



WHEN SHOULD SOURCE-CODE RUN OUTSIDE THE EVENT-DISPATCH-THREAD AND HOW CAN IT BE PUT THERE? For working on larger tasks like extensive loading-processes from a network or rendering-processes of pictures, separate threads may be initiated in the handler-routines. This could be described as implementing the handling-code inside non-blocking methods (page 231):

²⁶³ Since the class `java.awt.AWTEvent` has been declared abstract, the expression “`java.awt.AWTEvent`-object” is used as a short-cut for: “object instantiated from a class which extends the class `java.awt.AWTEvent`”.



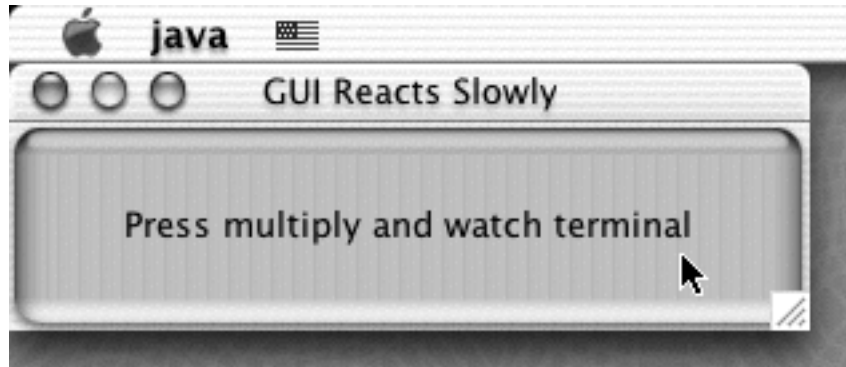
Of course, if possible, the (quasi-)parallel run of handling-routines should avoid to provoke race-conditions. The following pair of programs exemplifies how the event processing can be significantly improved: The first program processes a delayed (`Thread.sleep()`) loop in the event-dispatch-thread, the second program lets these loops run in separate threads. The second program reacts much faster to the user-input and since these loops do mainly pure waiting, the processing itself happens much faster!

Latency: The degree of reactivity of a system to external input is also called **latency**. Higher latency means faster reaction times, lower latency means slower reaction, blocking means no reaction.

```
// Button presses are memorized, but not acknowledged,
// and then are processed sequentially.
// (The button does not block)
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener; // interface
import java.awt.event.ActionEvent;
import javax.swing.SwingUtilities;

public class GUILowLatency
{
    public static void main( String unused[] ) throws Exception
    {
        JFrame aFrame = new JFrame( "GUI Reacts Slowly" );
        JButton aButton = new JButton( "Press multiply and watch terminal" );
        aFrame.getContentPane().add( aButton );
        aFrame.setSize( 300, 100 );
        aFrame.setVisible( true );
        ActionListener al = new AButtonActionListener();
        aButton.addActionListener( al );
        System.out.println( " main-thread, event-dispatch-thread? "
                           + SwingUtilities.isEventDispatchThread() );
    }
}

class AButtonActionListener implements ActionListener
{
    static char eventID= 64; // 65 = 'A'
    public void actionPerformed( ActionEvent e ) /////////////// the blocking method
    {
        eventID++;
        int i = 0;
        while( ++i < 10 )
        {
            System.out.print( " " + eventID );
            try{ Thread.sleep(1000); }
            catch( java.lang.InterruptedException evt ){ evt.printStackTrace(); }
        }
        System.out.println
        ( " button-press has been processed, in event-dispatch-thread? "
          + SwingUtilities.isEventDispatchThread() );
    }
}
/*
[localhost:~/JavaCodeFiles/SingleUserInterface/Events] jacobneuhhaus% javac GUILowLatency.java
[localhost:~/JavaCodeFiles/SingleUserInterface/Events] jacobneuhhaus% java GUILowLatency
main-thread, event-dispatch-thread? false
A A A A A A A A button-press has been processed, in event-dispatch-thread? true
B B B B B B B B button-press has been processed, in event-dispatch-thread? true
C C C C C C C C button-press has been processed, in event-dispatch-thread? true
D D D D D D D D button-press has been processed, in event-dispatch-thread? true
^C (Ctrl-C)
[localhost:~/JavaCodeFiles/SingleUserInterface/Events] jacobneuhhaus%
*/
```

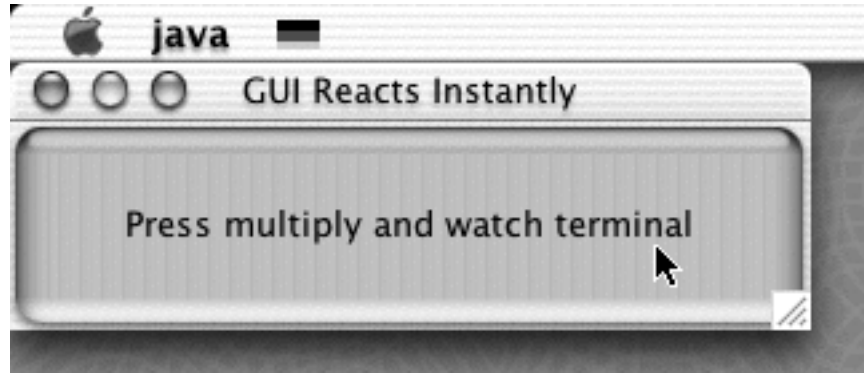
Pressing the button above in a fast succession should produce a *noticeably delayed* reaction in the terminal-window. The following source-code is a copy of the above, except for modifying the handler-method in such a way that the original handler-code runs in a thread of its own. This way the handler uses little resources of the event-dispatch-thread and with respect to the original version of the handler-code becomes a non-blocking method.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import java.awt.event.ActionListener; // interface
import java.awt.event.ActionEvent;
import javax.swing.SwingUtilities;

public class GUIHighLatency
{
    public static void main( String unused[] ) throws Exception
    { JFrame aFrame = new JFrame( "GUI Reacts Instantly" );
      JButton aButton = new JButton( "Press multiply and watch terminal" );
      aFrame.getContentPane().add( aButton );
      aFrame.setSize( 300, 100 );
      aFrame.setVisible( true );
      ActionListener al = new AButtonActionListener();
      aButton.addActionListener( al );
      System.out.println( " main, event-dispatch-thread? "
                          + SwingUtilities.isEventDispatchThread() );
    }
}

class AButtonActionListener implements ActionListener
{ static char eventID= 65; // 65 = 'A'
  public void actionPerformed( ActionEvent e ) //////////////// non-blocking method
  {
      new Thread()
      { public void run()
        { char threadEventID = eventID++;
          int i = 0;
          while( ++i < 10 )
          { System.out.print( " " + threadEventID );
            try{ Thread.sleep( 1000 ); }
            catch( java.lang.InterruptedException evt ){ evt.printStackTrace(); }
          }
          System.out.println
            ( " - " + threadEventID +
              " button-press has been processed, event-dispatch-thread? " +
              SwingUtilities.isEventDispatchThread()
            );
        }
      }.start();
  }
}

/*
[localhost:~/JavaCodeFiles/SingleUserInterface/Events] marycosway% javac GUIHighLatency.java
[localhost:~/JavaCodeFiles/SingleUserInterface/Events] marycosway% java GUIHighLatency
main, event-dispatch-thread? false
A A A A A B A C D B A C D B A C D B A C D B - A button-press has been processed, event-dispatch-
thread? false
C D B C D B C D B C D B C D B C D B - B button-press has been processed, event-dispatch-thread? false
- C button-press has been processed, event-dispatch-thread? false
- D button-press has been processed, event-dispatch-thread? false
^C (-> Ctrl-C pressed on keyboard)
[localhost:~/JavaCodeFiles/SingleUserInterface/Events] marycosway%
*/
```



Here, pressing the button above in a fast succession produces a “*non*”-delayed reaction in the terminal-window. — Up to now has been tried to relieve the event-dispatch-thread from unnecessary code, but how can a Java programmer, by a simple method-call, make the event-queue schedule to run code on the event-dispatch-thread? HOW CAN A JAVA PROGRAMMER SCHEDULE CODE IN THE EVENT-QUEUE TO BE RUN IN THE EVENT-DISPATCH-THREAD?

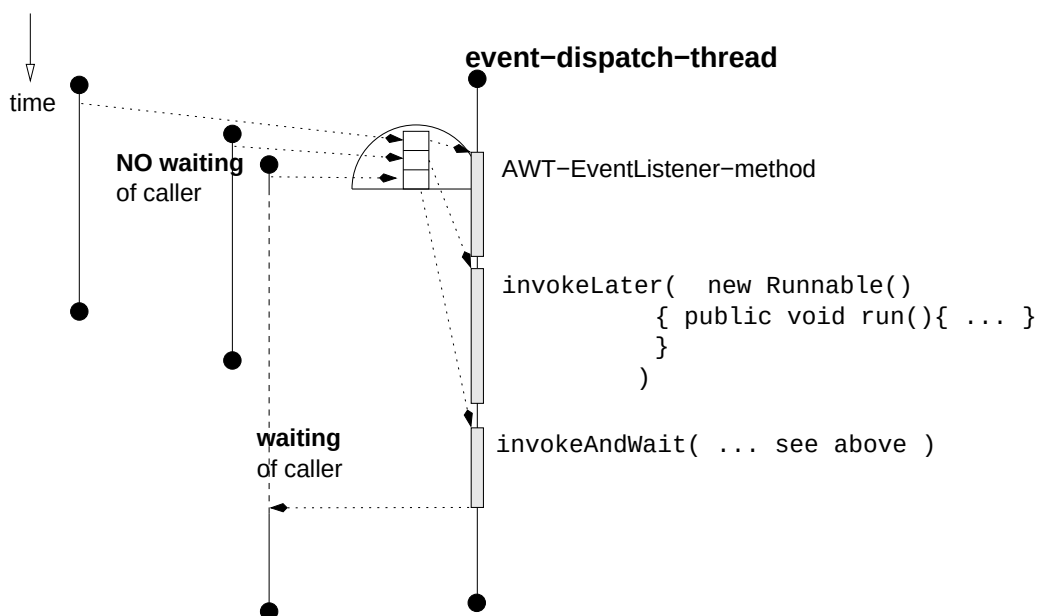
- First by writing that code into some standard handler-method (`mousePressed()`, `mouseReleased()`, `paintComponent()`). This requires explicit event-throwing on the side of the Java programmer.
- Secondly, by circumventing any event-throwing, it is possible to register a piece of source-code in the event queue and let have its turn of running in the event-dispatch-thread:

```
javax.swing.SwingUtilities.invokeLater
( new Runnable(){ public void run(){ block_of_source-code goes_here } } )
```
- A variant of this method **BLOCKS THE CALLER-THREAD** until the source-code has been run:

```
javax.swing.SwingUtilities.invokeLater
( new Runnable(){ public void run(){ block_of_source-code goes_here } } )
```

(The thread associated with the Runnable-interface is the event-dispatch-thread, each of the methods implement this association for the programmer!)

Means of Running Code in the Event-Dispatch-Thread



Swing also furnishes a class `javax.swing.Worker` which is basically a thread with additional Swing-specific services. The `SwingWorker`-threads are not treated in this text.

Java Programmer: Sources of AWTEvent-Objects

Java Programmer: Sources of AWTEvent-Objects — Up to now the instantiation and actual firing of AWTEvent-objects has been done automatically. Remember (beginning with page 574) that, mostly Swing screen-components of type `javax.swing.JComponent` (or extensions) furnish methods with a signature **`fireEventTypeEvent()`**, which enable the Java programmer to send an event directly from Java source-code. — Another problem has been still untouched, which has been hinted at as the event-listening-schemes were presented on page 579: How can the Java programmer establish a source of event-objects fully on her or his own?

(instantiation) First by instantiating an event-object of a class-type which extends the abstract class `java.awt.Event`:

```
java.awt.AWTEvent e = new AWTEvent()
```

Then there are different ways to bring an instantiated event-object into the standard AWT-switchboarding-mechanism:

(sending with `postEvent()`) A very direct way to “send” the given event-object is to put the object’s reference directly into the fifo-buffer of the system’s `EventQueue`-object, thereby using the standard AWT-infrastructure for event-propagation:

```
Component.getToolkit().getSystemEventQueue().postEvent( e );
```

(sending over a customized infrastructure) Using an infrastructure of one’s own for creating and organizing the relaying event-objects. (See the event-infrastructure used by the `java.beans`-package on page 314)

(sending with `fire...Event()`) After having registered appropriate listener-objects on a Swing screen-component, a call of a method `fireEventTypeEvent( e )` makes the component the source of an event-object:

```
addEventListener( referenceToEventListener );
// this makes the following work
fireEventTypeEvent( e );
```

(sending with `dispatchEvent()`) The event-object may be made an argument of the `dispatchEvent()`-method declared in classes of (base-)type `java.awt.Component`, `java.awt.MenuComponent` and `java.awt.EventQueue` (beginning with Java Version 1.2)

```
enableEvents(java.awt.event.AWTEvent.EVENT_MASK); // OR
addAWTEventListener( referenceToAWTEventListener );
// makes the following work
dispatchEvent( e );
```

This method may resemble the `fire...Event()`-methods but differs in one respect: A registered event-listener OR a previous call of the component’s `enableEvents()`-method lets the `dispatchEvent()`-method cause the event-object to be put automatically into the fifo-buffer of the system’s `EventQueue`-object. Thus the `dispatchEvent()`-method may also work without a registered listener! (In these cases the event-handling usually is done within the same class by a `processEvent()`-method thus used as an event-handler, as discussed later in this section.)

(sending with `processEvent()`) The low-level method, which actually puts the event-object into the event-queue, is the `processEvent( java.awt.AWTEvent e )`-method(s) of the `java.awt.Component` itself (and of the child-components). By overriding it, the method can be used as an event-handler, by calling the default-implementation of that method it can be used for event-sending purposes:

```
enableEvents( java.awt.AWTEvent.EVENT_MASK );
processEvent( e );
```

This method is called in the automatic event-dispatching-process (therefore it can be called a triggerable method). The Java programmer can influence the course of event-processing significantly by overriding this method. Then the standard behavior of forwarding the event-object to the system's event-queue (an object of type `java.awt.EventQueue`) can be conserved by placing the call `super.processEvent()` at a suitable place in the new definition of that method. In some cases, for example those of highest priority, the overriding `processEvent()`-method may call directly the handler-method of the listener-object, thus bypassing the event-queue of the event-dispatch-thread. More on the variety of uses for the method `processEvent()` can be seen in the next section.

The switchboarding-infrastructure given by the AWT-package, can be used to explicitly formulating the listener registration-process, and when overriding the `processEvent()`-method to explicitly see the event delivery reduced to a simple method-call! This is exemplified with the following implementation of a frequency-modulated timer. The timer signals are event-objects with their references sent to registered listeners in time-intervals which vary in length according to a cosine function. The timer runs in its own thread. The explicitly implemented `processEvent()`-method calls the handler-method (`actionPerformed()`) of the listener directly:

```
import java.awt.Component;
import java.awt.AWTEvent;
import java.awt.event.ActionEvent;
import java.awt.AWTEventMulticaster;
import java.awt.event.ActionListener;
import java.awt.EventQueue;
// main()-method found in class "RollerCoaster"

class OscillatingTimer extends java.awt.Component implements Runnable
{ EventQueue q;
  OscillatingTimer()
  { super();
    enableEvents( java.awt.AWTEvent.ACTION_EVENT_MASK );
  }
  /* // the conventional approach, here out-commented.
  this.addActionListener
  ( new ActionListener()
    { public void actionPerformed( ActionEvent e )
      { System.out.println( e.getActionCommand() + " action event happened" );
    } }
  );
  */
  // setEnabled(true); // for screen input (default)
  q = getToolkit().getSystemEventQueue();
  new Thread( this ).start();
} // end of constructor
public void run()
{ float x = 0; int d = 0;
  while(true)
  { x = x + (float)0.1;
    d = java.lang.Math.round( 500*(float)(java.lang.Math.cos(x)+1) );
    System.out.println(" Delay in milliseconds: " + d );
    try { Thread.sleep( d ); } catch( Exception e ){ e.printStackTrace(); };
    ActionEvent e =
      new ActionEvent( this, ActionEvent.ACTION_PERFORMED, "TimerSignal" );
    //dispatchEvent( e ); // alternative, out-commented
    processEvent( e ); >>>> sending an event-object by calling processEvent() >>>>
    /* // out-commented alternatives for experimentation
      q.postEvent( e );
      AWTEvent xx = q.peekEvent();
      System.out.println("Eventqueue: " + xx.toString() );
      //q.dispatchEvent( (AWTEvent)e );
    */
  } }
  ActionListener actionListener = null;
  public synchronized void addActionListener( ActionListener l )
  { actionListener = AWTEventMulticaster.add( actionListener, l );
  }
  public synchronized void removeActionListener( ActionListener l )
  { actionListener = AWTEventMulticaster.remove( actionListener, l );
  }
  public void processEvent( AWTEvent e ) >>>> handling an event directly >>>>
  { if (actionListener != null)
    { actionListener.actionPerformed( (ActionEvent)e ); // no event-queue
    }
    //super.processEvent( e );
    >>>> forwarding the event-object to be processed regularly >>>>
  } }
}
```

```

public class RollerCoaster implements ActionListener
{ public static void main( String unused[] )
  { RollerCoaster t = new RollerCoaster();
    OscillatingTimer o = new OscillatingTimer();
    o.addActionListener( t );
  }
  public void actionPerformed((ActionEvent e) )
  { System.out.println( e.getActionCommand() + " action-event received " );
    // isEventDispatchThread ??? Yes.
  } }
/*
C:\WINDOWS\JavaCodeFiles\SingleUserInterface\Events>javac RollerCoaster.java

C:\WINDOWS\JavaCodeFiles\SingleUserInterface\Events>java RollerCoaster
Delay in milliseconds: 998
TimerSignal action-event received
Delay in milliseconds: 990
TimerSignal action-event received
Delay in milliseconds: 978
TimerSignal action-event received
Delay in milliseconds: 961
TimerSignal action-event received
Delay in milliseconds: 939
TimerSignal action-event received
Delay in milliseconds: 913
...
(Ctrl-C)
C:\WINDOWS\JavaCodeFiles\SingleUserInterface\Events>
*/

```

(An introduction to the use of timers also is given in the Java TechTip of May 30th 2000.)

Java: processEvent () -Method as Drain and Source of AWTEvent-Objects

Java: processEvent ()-Method as Drain and Source of AWTEvent-Objects — The AWT-infrastructure of sending, switchboarding and handling event-objects of the `java.awt.AWTEvent` class can be influenced by the Java programmer with the `java.awt.Component.processEvent ( )`-method. As could be seen above, programming the dispatch of event-objects by calling `processEvent ( new java.awt.AWTEvent ( ) )` displays one aspect of the `enableEvent ( )/processEvent ( )`-service. Another aspect is given by the offer to intercept the delivery of the event-objects by properly overriding the `processEvent ( )`-method: In the AWT event-infrastructure, each object of class-type `java.awt.Component` has an empty triggerable method called `processEvent ( java.awt.AWTEvent e )`.

When extending the class of the screen-component, a Java programmer may override this method as follows:

```

protected void processEvent( java.awt.AWTEvent e ) // triggerable
{ preEventHandlingOrProcessingOfEventObjects();
  super.processEvent( e ); // an explicit call, not directly triggerable
}

```

Then the code found under `preEventHandlingOrProcessingOfEventObjects ( )` is run before the reference of the event-object is distributed across the listeners. If the forwarding-call `super . processEvent ( e )` is omitted, then the event-object's references are not channeled to the listeners which may have been registered using the regular AWT event-infrastructure. In that case, the method `processEvent ( )` changes from an event- handler/event-forwarder to a pure event-handler.

Thus a screen-component has two options for handling events by itself:

Listen to oneself Let the class-extension implement a listener-interface, that means especially to write a handler-method, and register it as a listener to itself as demonstrated on page 546, where an object of class `JFrame` handles events coming from its child-containers.

Implement processEvent () without forwarding Let the class-extension implement the `processEvent ( java.awt.AWTEvent e )`-method as a handler-method, by omitting the forwarding-call `super . processEvent ( e )`. If no listeners have been registered, then the instantiation and triggering of the event-objects has to be initiated by previously calling (usually within the constructor) the `enableEvents ( long BIT_MASK )`-method:

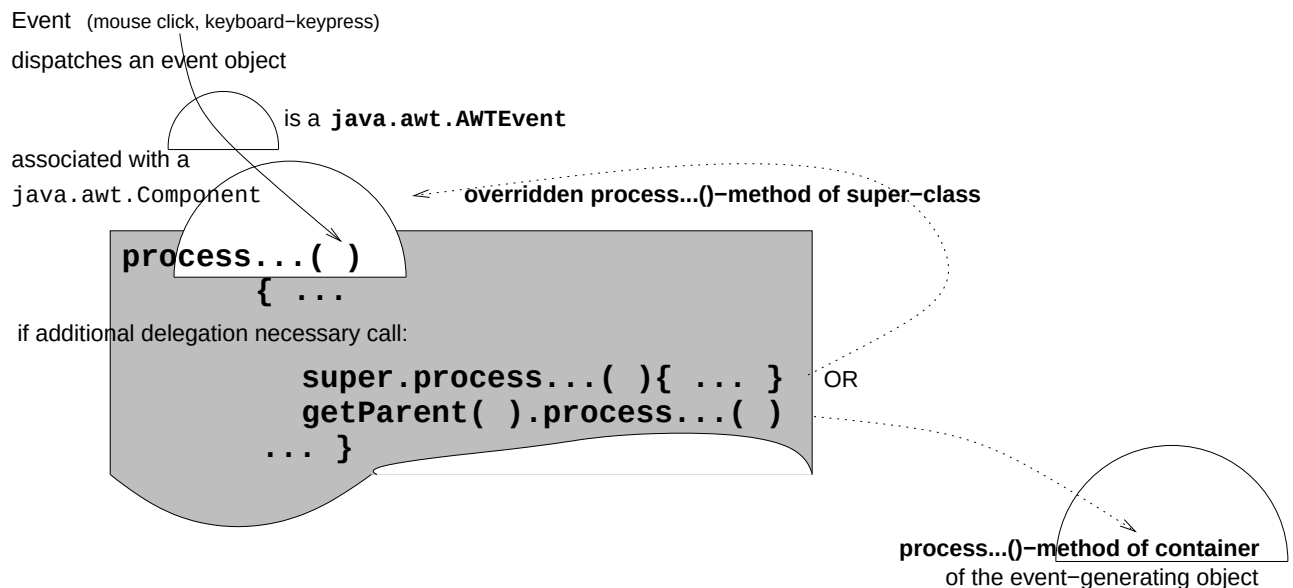
```

...
enableEvents( long BIT_MASK );
           // BIT_MASK values to be found as constants of java.awt.AWTEvent
...
protected void processEvent( java.awt.AWTEvent e ) // triggerable
{ preEventHandlingOrProcessingOfEventObjects();
  // super.processEvent( e );
  // An explicit call, useful if registered listeners are to be served.
}

```

The `java.awt.Component.processEvent()`-method is supplemented by many similar methods for context-dependent event-types, like the `processItemEvent()`-method.

NON-Delegational Event-Handling in AWT



Observe two options of delegation illustrated by the graphic above: Either calling the `process...()`-method of the super-class (along the class-extension hierarchy) or calling the `process...()`-method of the container-object (along the containment hierarchy of the screen-components).

As a summary recall the three basic uses of the group of `processEvent()`-methods:

Source A call of screen-component's object `processEvent( new java.awt.AWTEvent() )`-method with an event-object in its argument makes that object a source of an event-object (of `java.awt.AWTEvent` class-type).

Switchboard The use of an overridden `processEvent()`-method, with a call `super.processEvent( java.awt.AWTEvent )` declared inside, makes that method at least a forwarding part of the AWT-switchboard infrastructure.

Drain The use of an overridden `processEvent()`-method containing **NO** call of the un-overriding kind `super.processEvent( java.awt.AWTEvent )`, makes that method a pure event-handler.

Before the Java Version 1.1, the Java Runtime System used an event-communication-model that basically coded the event-types in a single variable of primitive-type `int` and each component had to identify and handle its own events. This model proved difficult in every-day use and isn't discussed here, although some of the conceptual aspects of that model effectively have been discussed above.

Interaction: Screen, Keyboard and Mouse

Interaction: Screen, Keyboard and Mouse — In this section are collected some ideas and examples of how to produce Java code fully interacting with the end-user via a dynamically changing Graphical User Interface. This also can be treated under the title “Event Dependent Painting”, which reflects the main aspect. Another form of event-dependent-painting can be seen when painting many images in fast succession or when repainting of parts of images; this is also called animation.

Java: Changing Components and Images On-Screen

Java: Changing Components and Images On-Screen — Before asking how to change the on-screen image of an existing Java screen-component, it may be useful to remember how to put pixel-defined images, graphics-objects and screen-components onto the screen. With this in mind, the way to change existing screen-objects should be easier to grasp.

Java: Painting-Methods

Java: Painting-Methods — The methods `paint()`, `paintComponent()` and the heavyweight component's `update()`-method have something in common with methods like `main()` and event-handlers like `mouseClicked()`, `mouseMoved()`: A Java programmer virtually never calls them from source-code; instead, these methods are being called by the Java Runtime Environment (triggerable methods, introduced on page 127). — An end-user resizing a frame by dragging the mouse, or a `setVisible()` method-call in the Java source-code are examples for actions that trigger the Java Runtime Environment (JRE) into instantiating objects of type `PaintEvent`. The JRE queues these event-objects into the `EventQueue`-object in the event-dispatch-thread, and lets this mechanism schedule the calls to the `paint()`, `paintComponent()`, `update()`-methods. This allows the Java Runtime Environment to coherently handle the GUI-actions initiated by the user and by the programmer. A Java programmer may take these triggerable methods as offered or a Java programmer may override these methods.

Even base-classes like `java.awt.Component` and `java.awt.Container` implement their `paint()`-method differently:

- The `paint()`-method of the class `java.awt.Component` is empty.

```
paint(Graphics g){}
```

- The `paint()`-method of an object of class-type `Container` calls the `paint()`-methods of all the child-components found inside the container screen-component. Conceptually:

```
paint(Graphics g){ for(int i=0; i<getComponentCount(); i++) addComponent(i); }
```

- Especially the Swing-classes are designed to override the `paint()`-method, making it call three additional methods in the following order: `paintComponent()` (the one that is usually overridden by a programmer), `paintBorder()` and finally `paintChildren()`. Conceptually:

```
paint(Graphics g){ paintComponent(); paintBorder(); paintChildren(); }
```

(This can be seen by comparing the respective entries in the tables on page 553, page 556 and page 559)

- By default, the `update()`-method of heavyweight objects completely redraws the background of the screen-component and calls the `paint()`-method.

```

public void update( java.awt.Graphics g )
{ g.setColor( this.getBackground() );
  g.fillRect( 0, 0, this.getWidth(), this.getHeight() );
  g.setColor( this.getForeground() );
  paint( g );
}
// "this" references the surrounding object of type Component.

```

The methods `update()` and `paint()/paintComponent()` often are overridden when doing animations. With non-heavyweight objects or Swing-objects the method `update()` has no special function.

Java programmers typically override the `paintComponent()`, `paint()` and `update()`-methods to specify what is to be painted. If the code of such an overridden method does not seem to readily cause the change of the screen-component's appearance, then a repaint can be triggered by the Java programmer from source-code by explicitly calling the `repaint()`-method.

Triggering rendering-methods in Java:
succession of calls

```

repaint( )                                explicit call by Java programmer from source-code
↓
update( java.awt.Graphics g ) automatically called
↓
paint( java.awt.Graphics g )  automatically called

```

Why should a Java programmer use explicit calls of the `java.awt.Component.repaint()`-method, instead of issuing an explicit call of the screen-component's `paint()`-method right away? Because an explicit call of the `repaint()`-method constitutes the conceptual equivalent to a `dispatch( PaintEvent e )`-call, sending the information of a `PaintEvent`-object regularly over the event-dispatch-thread to the associated handler-methods `update()`, `paint()` or `paintComponent()`. A direct call of these handler-methods would circumvent the regular event-handling and GUI-management and thus may cause race-conditions, confusing the Java programmer or the end-user.

Pixel-Based Images

Pixel-Based Images — Remember how images, essentially given as structured collections of pixels, can be displayed on-screen:

- A given array of `int`-values may code the full pixel-information according to a format, which is implemented by an object of type `java.awt.image.ColorModel`. Such an array can be transformed into a `MemoryImageSource`-object. From this, the method `java.awt.Toolkit.getDefaultToolkit().createImage(MemoryImageSource)` produces a displayable `java.awt.Image`-object. See also the program beginning on page 530.
- Images of standard formats like GIF or JPEG (introduced on page 521) can be registered for display by factory-like-methods like `getImage( String fileName )` giving a reference of abstract class-type `java.awt.Image` to the original file. — A call of image-related methods like `Graphics.drawImage(Image)` or `Component.prepareImage(Image)` transforms them into *displayable* `java.awt.Image`-references. (see also the table on page 529)

Images referenced by identifiers of type `java.awt.Image` can be scaled and cropped but a lot of refined methods for image-manipulation are only found in the package `java.jai` (Java Advanced Imaging). Images are presented on-screen by drawing them onto screen-components. This screen-component becomes specialized to painting the image by overriding its `paintComponent()`-method as follows:

```

public void paintComponent( java.awt.Graphics g)
{ g.drawImage(image); }           // SWING

```


Ultimately the image can be seen on-screen when the screen-component has been properly sized (calling the method `setSize()`) and when the screen-component is set to be visible either by being a top-level-screen-component (`setVisible(true)`) or by being the descendant of one. In a simple case of changing the displayed images, the Java program may reassign the image-reference and call the screen-component's **repaint()**-method, thus changing the image on-screen.

Graphics-Objects

Graphics-objects are simple geometrical structures (lines, ovals, rectangles, polygons) which can be drawn onto the surface of a screen-component (often a `javax.swing.JPanel`, `java.awt.Panel`). This drawing of graphical-objects relieves the programmer from working with the much more tedious pixel-graphics. Here of course, the transformation of the source-code to the screen-bitmaps always requires the computer to do some additional processing (rendering the code²⁶⁴ into a bitmap).

- The source-code for graphics-objects to be painted on a screen-component is stored in the screen-component's overridden method `paintComponent( java.awt.Graphics g )`; this is the case when extending Swing-classes. Contrary to this Swing-case, extensions of pure AWT-classes override the method `paint( java.awt.Graphics g )`. If AWT-child-components are to be painted, then the Java programmer calls `super.paint( g )` at the end of the `paint()`-code-block. Such an additional call is unnecessary in Swing, because of its triggered `paintChildren()`-method. The following code-fragment defines a line-drawing Swing-component: reason for Swing's paintChildren()-method

```
public void paintComponent( java.awt.Graphics g )
{ g.drawLine(10, 20, 100, 110); }           // SWING
```

For heavyweight screen-components (`JFrame`, `JWindow`, `JDialog`, `JApplet`, the only ones in Swing) it is also possible to override `update( java.awt.Graphics g )`-methods for incremental drawing, because then this method may not be used to redraw the background, avoiding flicker-effects.

Changing the graphics-objects painted on a screen-component can be done by making the `paintComponent()`'s code more flexible, for example by parameterizing the graphical-objects which are used or by letting the `paintComponent()`-method choose from multiple method-calls of which each one uses differing graphical-objects. There exists also the option of producing an array of child-components that are successively drawn into their part-time container.

Ultimately the graphics-objects can be seen on-screen when the screen-component has been properly sized (`setSize()`) and when the screen-component is set to be visible either by being a top-level-screen-component or by being the descendant of one (`setVisible(true)`).

To update the graphics-objects on-screen, the Java program may call the screen-component's **repaint()**-method.

Beginning with Version 1.1 of Java, drawing is possible on any screen-component (Before, drawing was only possible on a `java.awt.Canvas`-object). Thus lightweight components can be created more easily.

Screen-Components

Screen-Components — Remember that screen-components differ from images in implementing a functionality and are often represented by various images (labels show text and can change the text, buttons change their appearance upon mouse-clicks and even may send event-objects).

Furthermore, there are two kinds of screen-components, those (frames, windows, dialog boxes) which can be put directly onto the on-screen desktop of the underlying, native operating-system, the so-called top-level components; and the other kind of screen-components which are encountered as child-components (labels, buttons) within container-components.

To make screen-components visible on-screen, the Java programmer instantiates heavyweight classes like `javax.swing.JFrame` and, together with the added child-components (`add(child)` like `javax.swing.JButton`), makes them visible on the screen by calling `setVisible(true)` on the ancestor (here of type `JFrame`). Often the ancestor is given a fixed size (`setSize()`), otherwise the screen representation might become minimalized and is hardly to be found on the screen.

In that context there are some other control methods to be noted:

²⁶⁴ With pixel-based images *no rendering* is necessary, but if the images given pixel-wise have been coded in a non-native format, nevertheless a *transformation of formats* may become necessary!

`Component.setSize()` often is necessary to expand the component area from its minimal default size. Sends a `ComponentEvent`-object and may be deactivated by a non-default layout-manager. Alternatively, non-trivially overriding the methods `setMinimalSize()`, `setPreferredSize()` or `setMaximalSize()` may control the size of the (top-level-)screen-component.

`Component.setVisible( true )` – this command triggers a **top-level**-screen-component and all the contained child-components and other descendants into appearing on the screen; that means to become showing and thus actually perceivable for the end-user's eye! Observe the slight difference between being visible and showing, as already introduced on page 541 and as reformulated below in the gray rectangle.

`Container.add( childComponent )` callable on- or off-screen, this command registers an additional child-component with the layout-manager of the container. This method generates and sends a `ContainerEvent`-object. Nevertheless, the change becomes visible on-screen after calling `pack()` or `doLayout()`.

`Container.remove( childComponent )` deregisters a child-component. This method generates and sends a `ContainerEvent`-object. The change becomes visible on-screen after calling `pack()` or `doLayout()`.

`Window.pack()` first sets the sizes of the child-components to the values of their preferred size, then does the `Window`-object's layout and finally sets the size of the container accordingly, thus returning the `Window`-object validated. (Is also available with `JComponent` and `JInternalFrame`.) In short: The method `pack()` resizes child-components according to their `PREFERRED SIZE` and lays them out and sizes the container to fit the sizing of the child-components. — The same is done with containers of type `JPopupMenu` but here the child-components are sized according to their `MINIMAL` size!

`Container.validate()` usually is triggerable, only in special cases called directly in Java source-code. The method `validate()` triggers the layout of the current container (by calling `Container.doLayout()` on each sub-container which tells the default layout-managers, to layout the respective sub-containers.) Useful in cases of on-screen adding and removing of the child-components of a container.

`Component.repaint()` renews the on-screen appearance of the screen-component. The conceptual equivalent of a call `dispatch(PaintEvent)`. Callable on any child-component. Emits a `PaintEvent`-object which is added to the event-queue, when processed the `PaintEvent` triggers an `update()`-call that may include a `paint()`-call. If the part of the repainted image is known to the Java programmer, and that area can be circumscribed by a rectangle, then the `repaint()`-method may restrict itself to updating solely the area of that rectangle on the screen.

`Component.setVisible( false )` removes the on-screen appearance of the top-level screen-component.

`Window.dispose()` removes the object which represents the screen-component from the computer's memory. A call of `System.gc()`, which explicitly invokes the garbage-collector (page 444) would have the same effect, if the object wasn't referenced any more.

`Component.isDisplayable()` True when a top-level container or some other heavyweight component has found its native peer-code (by a system-call of `addNotify()`).

`Component.isVisible()` Top-level components like `JFrame`, `Frame`, `JWindow`, `Window`, `JDialog`, `Dialog` are initially set to be nonvisible. All other objects are set to be visible. Being visible means in the case of non-top-level components that they are ready to be displayed on-screen; such components show up on-screen when their ancestor containers and their top-level container are set to be visible. In the case of top-level screen-components this indicates that they are in the process of appearing on-screen or already have appeared on-screen.

`Component.isShowing()` signals that an end-user can find the component on the screen. If the end-user cannot see that component on the screen, then the end-user can make it appear by closing overlaying windows or by dragging them away.

visible, showing

Screen-components can have three basic states of display:

Visible: Ready for display on-screen or in the process of being displayed on-screen.

Showing: An end-user can find and perceive "showing" components on the screen with hers or his own eyes.

Nonvisible: The component cannot be seen by the end-user and is not scheduled for any display-process.

Each time the text above mentioned the term “Java programmer”, programmable control options have been shown which permit to control the Graphical User Interface right from the Java source-code. These are the possibilities of the so-called “application-triggered-programming”: A button-click may enlarge a window to show a list, or the movement of the mouse-pointer on the screen-component may paint a trace on the component’s surface. A text-field may take input from a user. An animation may be presented, or a time-delayed process (loading, rendering) may be reported. These are clearly **application-oriented drawing**-processes.

But then there are tasks, for example in windowing control, which are taken as being non-application-specific: A user may resize a frame on-screen by dragging the frame resizing area. A user may make a hidden window the top-level window by activating it with a mouse-click.

All these redrawing procedures are fully-automated by the Java Runtime Environment and by the native operating-system’s Graphical User Interface. A Java programmer needs not bother with those **application-unspecific drawing** tasks. Therefore these kinds of redrawing-mechanisms are also called system-triggered painting.

Consider this hint on background of system-triggered painting: Upon the GUI-actions of the end-user, the front-end of the underlying operating-system causes the Java Runtime Environment, or rather its `javax.swing.RepaintManager`, to throw event-objects when responding has become necessary. These `PaintEvent`-objects automatically are handled in the event-dispatch-thread by the `paint()`-method of the affected container and child-components. In the event-dispatch-thread, the `EventQueue`-object schedules the GUI-related handling of associated event-objects. This successive scheduling and handling avoids race-conditions which may arise if the GUI was changed concurrently from calls originating in differing threads.

Summary: Drawing On-Screen

Summary: Drawing On-Screen — Standard painting in Java is done on screen-components not directly on-screen. Standard painting in Java relies on a computer-platform that uses a front-end in the form of a desktop: Java’s top-level-components can be put directly onto the desktop. On these top-level-components the painting takes place within three conceptual levels:

- Drawing on **bitmap level** means to use the method `paintComponent( Graphics g ){ g.drawImage(ImageData); }` and a method `drawImage()` out of the graphics-toolbox to put graphics onto a screen-component.
- Drawing on **graphics-object level** also uses the method `paintComponent(Graphics g ){ g.drawRectangle(); }` but together with toolbox-methods like `drawRectangle()` for putting graphics-objects onto a screen-component.
- Drawing on **screen-component level** uses statements like `topLevelContainer.add( screen-Component );` to organize components, containers. Properties like size, layout (`setSize()` or `pack()`) and display (`setVisible(true);`) may be controlled by calling the respective methods.

Especially in the development of Graphical User Interfaces the amount of knowledgeable experimentation should not be underestimated when producing software! Having the basic ideas of a programming-language and the intended application in mind, software-development usually presents itself as an highly incremental process, generally meaning to endure multiple cycles of trying and retrying.

Java: User Changes User Interface

Java: User Changes User Interface — The following two examples let the end-user see an immediate reaction of the Graphical User Interface to the end-user’s input. As a first example is presented a label which can be used like a button: The program counts the number of mouse-clicks on the label and updates the text of the label accordingly. — The class which contains the `main()`-method also acts as a mouse-listener, but not by implementing the `MouseListener`-interface, but by extending the `MouseAdapter`-class. Remember that the adapter-kind of class implements all the methods of the `Listener`-interface as methods with an empty code-block (page 580). Therefore, only the desired handler-method `mouseClicked()` needs to be non-trivially overridden, all other handler-methods of the `Listener`-interface can be ignored. — The updateable label, referenced by “`lblUpdated`”, is put into a `JPanel`-container which itself is added to the content-pane of a top-level screen-component of type `JWindow`. — Various options for changing the screen appearance of the updateable label are demonstrated, from establishing a tooltip, to changing color and font. Calling `setOpaque(true)` makes a background-color visible! Some lines have been outcommented to demonstrate possibilities of experimenting, for example calling the method `pack()` minimizes

unfilled areas and repositions the window; instead of this, the command `setSize()` has been used. This guarantees the experimenting Java programmer to see the label light green, the panel darker blue, the content-pane black and the window appears red, moments before the content-pane is drawn.

```
import javax.swing.JWindow;
import javax.swing.JLabel;
import javax.swing.JPanel;
import java.awt.event.MouseEvent;
import java.awt.event.MouseAdapter;
import javax.swing.SwingUtilities;

public class LabelUpdate extends MouseAdapter
{
    int no = 1;
    JLabel lblUpdated;
    JWindow w;
    JPanel p;
    public static void main( String unused[] ){ new LabelUpdate(); }
    LabelUpdate()
    {
        lblUpdated = new JLabel( "Label unclicked" );
        lblUpdated.setForeground( new java.awt.Color(0,255,0) ); // green
        lblUpdated.setBackground( new java.awt.Color(200,255,200) ); // light green
        lblUpdated.setFont( new java.awt.Font("NewCenturySchoolbook", java.awt.Font.ITALIC, 32) );
        lblUpdated.setToolTipText("This is the Tooltip for the Label!");
        //lblUpdated.setMaximumSize( new java.awt.Dimension(450,90) );
        //lblUpdated.setPreferredSize( new java.awt.Dimension(400,90) );
        // outcommenting makes sizes variable
        //lblUpdated.setSize( new java.awt.Dimension(20,90) );
        lblUpdated.setBorder(
            new javax.swing.border.EtchedBorder( javax.swing.border.EtchedBorder.RAISED ) );
        lblUpdated.setOpaque(true);
        p = new JPanel();
        p.setBackground( new java.awt.Color(100,100,200) ); // lighter blue
        p.add( lblUpdated, "North" );
        w = new JWindow();
        w.setForeground( new java.awt.Color(255,0,0) ); // red
        w.setBackground( new java.awt.Color(255,100,100) ); // light red
        w.getContentPane().add( p, "East" ); // default layout is border layout
        w.getContentPane().setBackground( new java.awt.Color(0,0,0) ); // black
        //w.pack(); // collapses empty areas and may cut label
        w.setSize( new java.awt.Dimension(500,200) );
        // no preferredSize with top-level heavyweights
        lblUpdated.addMouseListener( this );
        // or here new MouseAdapter with handler method inside
        w.setVisible( true );
    }
    public void mouseClicked( MouseEvent e )
    {
        System.out.println( "event dispatch thread? " +
            SwingUtilities.isEventDispatchThread() );
        lblUpdated.setText( "Label clicked " + (no++) + " times " );
        // w.pack(); // Collapses empty areas and repositions window!
    }
}
/*
set...Size makes rigid forms inside a panel.
No set...Size lets the components dynamically readjust to the necessary areas.
setOpaque(true) makes background color visible!
pack() minimizes unfilled areas and repositions the window.
After some waiting appears the tooltip.
Again: incremental development and some experimentation is advised!
Label's color: light green
Panel's color: darker blue
Content-pane's color: black
Window-background color: red (only shortly visible during repaint-procedures)
*/
/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction > javac LabelUpdate.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction > java LabelUpdate
event dispatch thread? true
event dispatch thread? true
event dispatch thread? true
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction >
*/
```



Tooltip: If a user positions the mouse-pointer on a screen-component, with the screen-component implementing a tooltip-functionality as the label shown above, then, after a short waiting-time of a few seconds, a rectangle will appear containing an informative description of the associated screen-component. For example, a folder icon may have a tooltip that reads "This is a folder, double-clicking opens it, drawing moves it into another folder."

The next example shows how a screen-component's text (label) is updated concurrently to paint a graphics-object with the `paintComponent()`-method. This program produces a window with two child-components: The movements of the mouse-pointer on the panel (first child-component) are tracked by painting a line from the last registered mouse-position to the current mouse-position. The current mouse-positions are shown as the text of a label (second child-component) above the panel. — The class implements a `MouseMotionListener`-interface and therefore has to provide a definition for the two methods `mouseMoved()` and `mouseDragged()`. Though dragging is not intended to be processed the method `mouseDragged()` has defined to be an empty handler-method. (Listener-interfaces versus Adapter-classes) — Here three options of `repaint()`-methods can be experimented with: One repaints the entire window and thus lets it flicker (1), this is not very economic and not very nice to look at; the other version (2) repaints the panel. The last version (3) (not out-commented) repaints only the rectangle given by the line which is to be drawn. — Since the `mouseMoved()`-handler-code runs in the event-dispatch-thread so does any method called from that code, especially in this case it is the "updatePosition()" -code of the "MousePad"-panel. — Why is the `MouseEvent` re-referenced by a final reference inside the `mouseMoved()`-handler-code? The given reference remains with the given event-object, thus the event-object cannot be substituted by a modified version.

```
import javax.swing.JWindow;
import java.awt.Container; // for the Swing content-pane
import javax.swing.JPanel;
import javax.swing.JLabel;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import javax.swing.SwingUtilities;

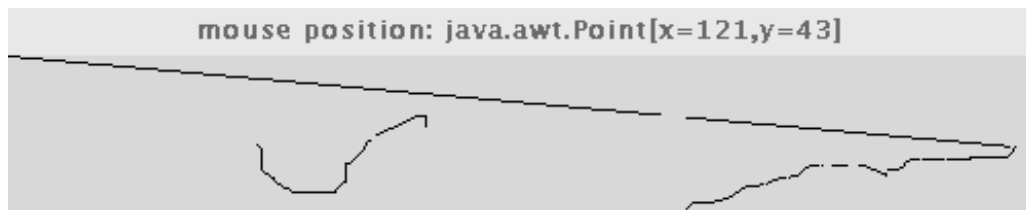
public class SomeMousePad implements MouseMotionListener
{
    JLabel lblFixed, lblUpdated;
    JPanel textDisplay;
    MousePad mousePad;
    JWindow w;
    Container contentPane;
    public static void main( String unused[] ){ new SomeMousePad(); } // --- main()-method
    SomeMousePad() // --- constructor
    {
        lblFixed = new JLabel( "mouse position:" );
        lblUpdated = new JLabel( "no focus yet" );
        textDisplay = new JPanel();
        textDisplay.setOpaque(true);
        textDisplay.setBackground( new java.awt.Color( 230, 230, 255 ) );
        textDisplay.add( lblFixed ); textDisplay.add( lblUpdated );
        mousePad = new MousePad();
        mousePad.setOpaque(true);
        mousePad.setBackground( new java.awt.Color( 230, 200, 255 ) );
        w = new JWindow();
        contentPane = w.getContentPane();
        contentPane.add( mousePad, "Center" );
        contentPane.add( textDisplay, "North" );
        w.setSize(500,100);
    }
}
```

```

        mousePad.addMouseMotionListener( this );
        w.setVisible( true );
    }

    public void mouseMoved( MouseEvent e )                                // --- handler
    { /* System.out.println( "event dispatch thread? " +
        SwingUtilities.isEventDispatchThread() ); */
        final MouseEvent f = e;
        lblUpdated.setText( f.getPoint().toString() );
        textDisplay.repaint();
        mousePad.updatePosition( f );
    }
    public void mouseDragged( MouseEvent e ){                             // --- empty handler
class MousePad extends JPanel // ----- drawing-surface
{ MousePad(){ super();}
  int x1, y1, x2, y2;
  void updatePosition( MouseEvent e )
  { x2 = x1; y2 = y1; x1 = e.getX(); y1 = e.getY();
    // w.repaint() // lets flicker the entire screen-component (1)
    // this.repaint(); // line follows mouse-pointer (2)
    this.repaint( Math.min(x1,x2), Math.min(y1,y2),
        Math.abs(x1-x2)+1, Math.abs(y1-y2)+1 ); // (3)
    // above: only the relevant rectangles are redrawn,
    // thus only erasing lines when lines cross!
  }
  protected void paintComponent( java.awt.Graphics g )
  { super.paintComponent( g ); // paints background
    g.drawLine( x1, y1, x2, y2 );
  } } }
/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction > javac SomeMousePad.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction > java SomeMousePad
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction >
*/

```



This has been a glimpse into event-dependent on-screen updating of screen-components and graphics, where the end-user generated the event-objects. Observe the deficiencies in the painting-routine.

Java: Animation

Java: Animation — As could be seen, the on-screen changing of screen-components can be made possible when furnishing appropriate source-code. This includes Java source-code which presents a sequence of images in rapid succession. If these images are properly drawn, then their rapid appearance in front of the end-user's observing eyes may give the impression of a moving picture. (Depending on the movement to be depicted, "rapid" means 10 frames per second for acceptably displaying smooth movements, but better are 24 frames²⁶⁵ (= images) per second for presenting faster movements without unimpressive erratic jumps). Thus, animation means to present a sequence of images to the viewer, in a succession so fast, to give the viewer the impression of a picture with moving content. (Television and cinema use this effect with a differing technical outfit.)

The following program uses a previously introduced method to produce a sequence of similar images. Then these images are shown in a rapid succession, creating the impression of a ring of pink color continuously surging over a "Hello World"-string. — The timing for showing the succession of images can be done in two ways:

- a while-loop which puts the thread to sleep between `repaint()`-calls or

²⁶⁵“Frame”, in the context of movies or animated pictures, denotes a single image within a sequence of images. “Frame”, in the context of programming a Graphical User Interface (GUI), denotes a top-level screen-component (page 509). “Frame”, in the context of network communication, denotes a packet (page 349), a sequence of bytes containing data and control-information (destination-address, source-address).

- a timer which produces an `ActionEvent`-object in predefined time-intervals and uses a handler-method to call the `repaint()`-method

The second method, though involving a more complex approach may be preferred because the `sleep()`-method locks the code. — The constructor of the “`ImageArrayExample`” fires a `PropertyChangeEvent`-object to notify the “`UpdatableLabel`” of a newly produced image.

```
import java.awt.image.MemoryImageSource;

public class AnimateImageArray
{
    public static void main( String unused[] ) // ----- main
    { javax.swing.JFrame desktopFrame = new javax.swing.JFrame();
      UpdatableLabel label = new UpdatableLabel();
      desktopFrame.getContentPane().add( label, "South" );
      desktopFrame.setSize( new java.awt.Dimension(200,220) );
      desktopFrame.setVisible(true);
      final Projector projector =
          new Projector( new ImageArrayExample( label ).handover() );
      desktopFrame.getContentPane().add( projector, "Center" );
      projector.addPropertyChangeListener( label );
      desktopFrame.validate(); // If omitted, the display window stops.
      /* while(true){ projector.repaint();try{Thread.sleep( 400 );}
        catch(java.lang.Exception e){ e.printStackTrace(); };} */
      new javax.swing.Timer
      ( 100 , // delay in milliseconds
        new java.awt.event.ActionListener() // repaint on timing signal
        { public void actionPerformed( java.awt.event.ActionEvent e )
          { projector.repaint(); }
        }
      ).start();
    } }

class UpdatableLabel
extends javax.swing.JLabel implements java.beans.PropertyChangeListener
{ public void propertyChange( java.beans.PropertyChangeEvent e )
  { setText(e.getPropertyName() + e.getNewValue() );
  } }

class Projector extends javax.swing.JComponent
{ int j, noOfImages;
  java.awt.Image [] image; // array of images
  java.awt.Point imageOrigin;
  Projector( java.awt.Image image[] ) // constructor,
  { noOfImages = image.length; // draws a given image-array onto the screen.
    this.image = image;
    imageOrigin = new java.awt.Point(0,0);
    this.setPreferredSize( new java.awt.Dimension(image[0].getWidth(this),
                                                    image[0].getHeight(this)) );
  }
  public void paint( java.awt.Graphics g ) // or paintComponent()
  { j++;
    g.drawString( "Hello World!", 30, 30 );
    g.drawImage( image[j], imageOrigin.x, imageOrigin.y, this );
    // System.out.println("image number " + j + " painted.");
    firePropertyChange("painted image no.", new Integer(j-1), new Integer(j));
    if( j == noOfImages-1 ) j = -1;
  } }

class ImageArrayExample extends javax.swing.JComponent
{ int noOfImages = 10; // allows to fire property-change-event
  int [] pixArray;
  java.awt.Image [] image = new java.awt.Image[ noOfImages ]; // array of images
  ImageArrayExample( java.beans.PropertyChangeListener listener )
  { int width = 200, // constructor, produces the image-array
    height = 200,
    firstPix = 0, // begin with first entry in array pixArray
    lineLength = width; // number of pixels per horizontal lineLength
    addPropertyChangeListener( listener );
    for(int i = 0; i < noOfImages; i++)
    { pixArray = createPixArray(width, height, (float)i/(float)noOfImages );
      MemoryImageSource mis =
```

```

        new MemoryImageSource(width, height, pixArray, firstPix, lineLength);
        image[i] = java.awt.Toolkit.getDefaultToolkit().createImage( mis );
        firePropertyChange("created image no.", new Integer(i-1), new Integer(i));
        // System.out.println("image number " + i + " generated.");
    } }

    int [] createPixArray(int width, int height, float parameter)
    { int pixArrayLength = width * height; // ----- composing a single image
      int [] pixArray = new int[ pixArrayLength ];
      int blue, green, red, alpha, x, y, value, div = width * width + height * height;
      float phase;
      for(int i = 0; i < pixArrayLength; i++)
      { x = i % width;
        y = i / width;
        phase=(float)(x*x + y*y) / (float)div;
        value = (int)( 255 * java.lang.Math.pow(
                      java.lang.Math.sin( java.lang.Math.PI * (phase - parameter) ),
                      2) );
        red = alpha = value; green = 100; blue = 200;
        pixArray[i] = Transform.intFromBytes(blue, green, red, alpha);
      }
      return pixArray;
    }
    java.awt.Image [] handover(){ return image; }
}

class Transform
// uses int-variable to store an array of four unsigned byte-values
// (used by ARGB color model)
{ static int intFromBytes(int one, int two, int three, int four) // int range 0..255
  { int [] handover = {one,two,three,four};
    return intFromBytes(handover);
  }
  static int intFromBytes(int [] b) // that's b[0],b[1],... least significant first
  // the work is done in the return clause the rest are machinations to avoid errors
  { int [] intArray = new int[4];
    boolean smaller, larger, right;
    for(int i = 0; i < b.length; i++)
    { smaller = b[i] < 0;
      larger = b[i] > 255;
      right = !smaller & !larger;
      if(smaller) intArray[i] = 0;
      if(larger) intArray[i] = 255;
      if(right) intArray[i] = b[i];
    }
    return intArray[3] << 24 | intArray[2] << 16 | intArray[1] << 8 | intArray[0];
  }
  static int [] bytesFromInt(int n)
  { final int BIT_MASK = 255; // binary = 00000000 00000000 00000000 11111111
    int [] b = {0,0,0,0}; // cannot be written in one line return {(byte) n & BIT_MASK, ...};
    b[0] = n & BIT_MASK; // extracts
    b[1] = (n >> 8) & BIT_MASK;
    b[2] = (n >> 16) & BIT_MASK;
    b[3] = (n >> 24) & BIT_MASK;
    return b;
  }
}
/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction > javac AnimateImageArray.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction > java AnimateImageArray
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction >
*/

```

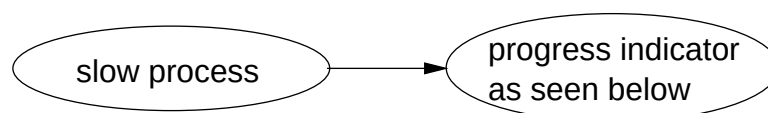



So much for an elementary introduction to animation. The previous example needed some time to generate the images and notified the end-user about the progress. Situations in which the end-user has to be informed about a delay in reactivity (also cases of low latency) of the computer or associated devices like network, printer, or scanner, occur relatively often.

Concept: Time-Delayed Processing

Concept: Time-Delayed Processing — An end-user may receive requested data on-screen (or some other device, like printer, hard-disk or network) so slowly that the end-user notices the delay. In those cases, the operating-system or the application should inform the user about the delay by changing the cursor into a clock-icon or by presenting a so-called progress-bar. Without any such reaction the user might become confused.

The time-delay occurs when resources (processor(s), network- or bus-connections, Random Access Memory or any other input/output software) are used up to the limits of their capacity. If a programmer can foresee delays, then such a user-notifying mechanism constitutes an essential part of the overall Quality of Service.



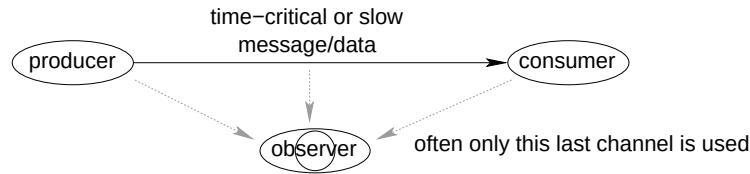
If the process of working through a method takes more than a few seconds, minutes or hours or even longer, then the progress should be made visible to the user. Standard means are a

1.  **rotating cursor** a standby blinking of a cursor, or an
2. **82% done 0:2:32 hours left** updated text possibly showing the percentage of the accomplished task or the estimated time remaining to wait; or
3.  **Working... / Loading... / Rendering...** a graphical representation as a progress-bar, filling continuously during the time the process takes to finish.

The information about the progress has to be generated somewhere close to the actual time-consuming process. A time-consuming process often takes place in a thread of its own or is triggered by a call of a non-blocking method. Remember the distinction between blocking and non-blocking methods, as introduced in the chapter about multithreading on page 231.

From where and how to get the information about the proceedings? A programmer can only inform the user about delays which can be anticipated: If data received from a network gives no information about

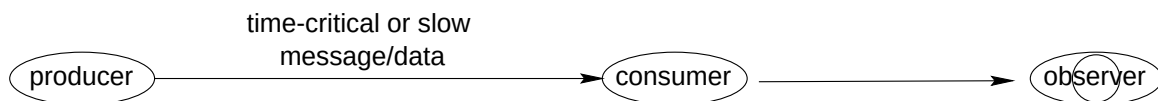
how much data will follow, notification of the end-user about the exact progress becomes difficult. If the volume of the data to expect is known, when sending a large chunk of data, then the amount of the remaining data may be calculated and the user can be informed about how fast the sending can be estimated to progress. Such considerations may take place in a parallel thread which listens for timing signals or other information:



Remember the message-model, which is given by three parts: The source of the message, the message itself and the drain of the message. And remember that messages could be small chunks of data transmitting a single event-information, or a message may contain an entire video-file. (Technically, for easier bus- or network-management, larger messages often are broken into packets, which later are reconstructed to form the original message.)

For example, the transfer of a file may be delayed noticeably for the end-user, or some part of a program needs information about the status of the transfer. Since the progress of the data-transfer gives information about the interaction between the producer (source) and the consumer (drain) of the data(-message(s)), introducing a separate observer-object as a receiver of the status-information may seem sensible²⁶⁶.

This observer can get its information from the producer, the consumer or even from the messages; but if only the success is relevant, the most reliable source for the status-information is given by the consumer. Thus, many times the message-model for time-delayed messaging looks as follows:



In short: Monitoring the transfer of time-critical or "slow" data-movement from a producer to a consumer is USUALLY DONE BY THE CONSUMER INFORMING AN OBSERVER.

This kind of monitoring takes place in downloading or streaming larger images, audio or video files over the Internet. In general, an observer is useful when receiving data-inputs slowly in comparison with the reaction-time of the targeted computer-system. The reason for the "slowness" may be the little available capacity of the data-communication-lines ("low bandwidth") or the time-delayed production of video- or audio-signals or even the time a user needs to think while typing keyboard inputs.

Since producing-, loading- and converting-procedures usually take time, these procedures generally take place in individually associated threads. This threading is either done explicitly by the programmer or is furnished to some extent by the programming-environment. In that context are found methods which start new threads to accomplish a task and return immediately:

Methods initiating a new thread and returning immediately are also called non-blocking methods or asynchronous methods. (Also see page 231)

Since the routing of the data is of least importance in this model, the notion of the message itself usually can be ignored by the application-programmer.

Java: Time-Delayed Data

Java: Time-Delayed Data — The Java AWT (Abstract Window Toolkit) furnishes a (partial) implementation, coupled to some underlying infrastructure, — a framework — of the concept as described above.

Java Overview: Interfaces and Classes for Time-Delay

Java Overview: Interfaces and Classes for Time-Delay — There are three interfaces in the `java.awt.image`-package:

²⁶⁶In a less conceptual, more pragmatic way: If the programmer can anticipate the consumer to be waiting for "slow" data, then the consumer acts within its own thread; and the observer may check the status of the consumer from some other thread.

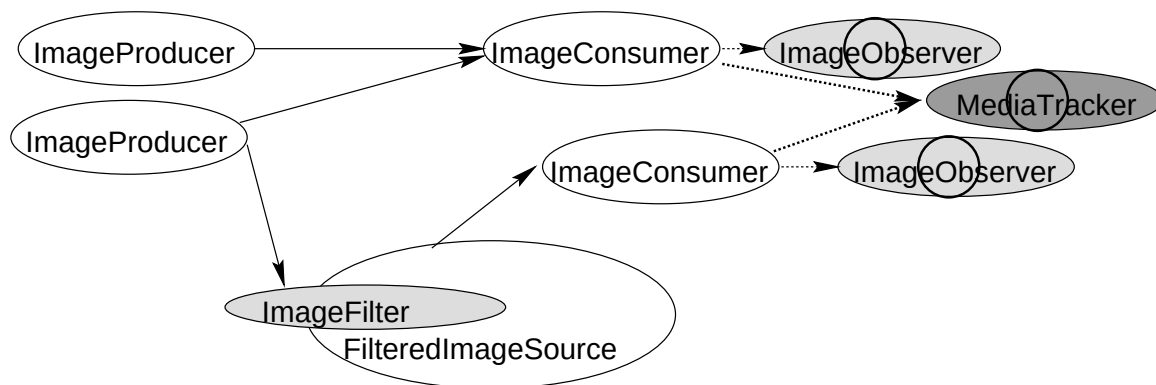
ImageProducer implemented by classes which emit image-data, like `java.awt.image.MemoryImageSource`. This interface requires a small set of methods to be implemented for registering consumers and starting data-production. The actual implementation of the data-transfer is left completely to the choice of the programmer.

ImageConsumer implemented by classes, whose instantiated objects have to register with **ImageProducers** for receiving data. The classes `PixelGrabber` and `ImageFilter` implement the interface **ImageConsumer**. Many consumers per producer are possible.

ImageObserver, implemented by the class `java.awt.Component`; the associated handler-method `imageUpdate()` is available with each object of class-type `Component`.

These three Java reference-types have been mentioned first at the end of the Java multithreading chapter, as an example of implicitly implemented multithreading (page 232). A pragmatic introduction has been given on page 529 around the table that shows the different methods of how to get a Java reference of type `Image` to native image-data.

Another more flexible approach has been attempted by introducing the class `java.awt.MediaTracker`. This class is a direct sub-class of the class `java.lang.Object`. Unlike an **ImageObserver**-object, which is updated only by ONE **ImageConsumer**, a **MediaTracker**-object can register as observer to MANY **ImageConsumer**-objects:



So, only one **ImageObserver** per consumer is possible. One observer for multiple consumers can be realized with a `java.awt.MediaTracker`. The `java.awt.MediaTracker` has to add the to be monitored images explicitly and to give them an ID-number. Other methods are described in the [Java Platform API Specification \[JPAPIS\]](#). Additionally, the `java.awt.image.ImageFilter`- and `ImageFilterSource`-classes introduce the base-classes for standard transformations of image-data; `Filter/FilterSource`-pairs can do some pre-processing for the consumers. The section below shows an explicitly implemented interface of type `java.awt.image.ImageObserver`.

The Workings of an ImageObserver-Object

The Workings of an **ImageObserver**-Object — An **ImageObserver**-object has only one method, the `imageUpdate()`-method. Any other methods taking a reference to that **ImageObserver** in their argument-list are IMPLICIT (MULTIPLE) CALLERS OF THAT METHOD, for example:

```

java.awt.Image.getProperties(),-.getHeight(),-.getWidth()
java.awt.Graphics.drawImage()
java.awt.Toolkit.checkImage()

```

ALL `java.awt.Component`-classes (including their extensions) implement the **ImageObserver**-interface. How is its `imageUpdate()`-method implemented? See the source-code below. How is an **ImageObserver** registered to an **ImageConsumer**? In the `java.awt` package, by making an **ImageObserver**-object an argument in non-blocking methods, like `drawImage(..., ImageObserver)`. One of the simpler uses of this framework means to consider any extension of the `java.awt.Component`-class and to just override the `updateImage()`-method. An example of this gives the following image-viewer:

```

/*
--- Producer: References of class-type Image make it possible to get the name of the
            image-producer, by calling Image.getSource().
--- Consumer: Here hidden within the class JFrame.

```

```

--- Observer: The Display-class extends the javax.swing.JPanel-class,
      which extends the java.awt.Component-class. Therefore the Display-class
      implements the ImageObserver-interface and can override the imageUpdate()-method.
*/
import java.awt.Toolkit;    // direct access to platform-specific resources
import java.awt.Image;      // abstract class gives references to image-data
                          // (native or within JRE)
import java.awt.Graphics;   // a Java toolkit for graphical-objects ...
import javax.swing.JPanel;  // used as drawing-surface
import javax.swing.JFrame;  // screen-area for display
import javax.swing.JComponent;
import javax.swing.SwingUtilities;

class ImageViewerInfo extends JFrame
{
    public static void main( String fileNameOfImage[] )
    { String fileName = fileNameOfImage[0]; // read in image from current directory
      new ImageViewerInfo( fileName );
    }
    int beginWidth = 200, beginHeight = 50;
    Image theImage;
    ImageViewerInfo( String fileName )
    { super( fileName );
      theImage = Toolkit.getDefaultToolkit().getImage( fileName );
      Display imageDisplay = new Display( theImage, this );
      getContentPane().add( imageDisplay, "Center" );
      getContentPane().add( new javax.swing.JLabel("Loading " + fileName), "South" );
      setSize( beginWidth, beginHeight );
      setVisible( true );
    }
}

class Display extends JPanel // implements the interface java.awt.image.ImageObserver
{ Image image; ImageViewerInfo container;
  javax.swing.JLabel lbl;
  Display( Image img, ImageViewerInfo c )
  { image = img; container = c;
    lbl = new javax.swing.JLabel("Loading ... ");
    container.getContentPane().add( lbl, "North" );
  }
  int areaCurrent = 0, areaGoal = 0, imageWidth = 1, imageHeight = 1, count;
      // if all values above are 0, no image is loaded
  double percent = 0;
  //
  //
  /** This method does the actual painting
   */
  public void paintComponent( Graphics drawingTools )
  { drawingTools.drawImage( image, 0, 0, imageWidth, imageHeight, this ); // new thread
  } // container is ImageObserver
  //
  //
  /** This current object is of the interface-type ImageObserver,
      therefore the method imageUpdate() can be overridden
   */
  public boolean
  imageUpdate( java.awt.Image img, int bits, int x, int y, int width, int height )
  { // values of currently loaded rectangles of pixels
    System.out.println( "imageUpdate runs in event-dispatch-thread? " +
      SwingUtilities.isEventDispatchThread() );
    if( (bits & HEIGHT) != 0 )
    { System.out.println("Now available, height of base image = " + height + " pixel");
    }
    if( (bits & WIDTH) != 0 )
    { System.out.println("Now available, width of base image = " + width + " pixel");
    }
    if( (bits & PROPERTIES) != 0 )
    { System.out.println
      ( "Now the java.awt.Image.getProperty( PropertyName, ImageObserver )" +
        " returns non-trivial data to consumer: " +
        img.getProperty( "comment", container ).toString()
      );
    }
    if( (bits & SOMEBITS) != 0 )
    { System.out.println
      ( "some pixels have been delivered, their range: " +
        new java.awt.Rectangle(x,y,width,height) );
      areaCurrent = areaCurrent + width * height;
    }
  }
}

```

```

        percent = Math.round( 10000*(double)areaCurrent / (double)areaGoal )/(double)100;
        System.out.println("Percent loaded: " + percent );
        lbl.setText("Loaded " + percent + " %");
        lbl.repaint(); return true;
    }
    if( (bits & ALLBITS) != 0)
    { System.out.println("Consumer received entire image, size: " + width + " " + height);
      container.setSize( width, height );
      imageWidth = width; imageHeight = height;
      container.validate(); // no use for container.repaint();
      return false;
    }
    if( (bits & FRAMEBITS) != 0)
    { System.out.println
      ("Now available, " + (count++) + ". frame of a multiframe image.");
    }
    if( (bits & ABORT) != 0)
    { System.out.println("Producer-Consumer: automatic repeat ");
    }
    if( (bits & ERROR) != 0)
    { System.out.println("Producer-Consumer: error, no further tries"); return false;
    }
    // what to do further when notified by an consumer:
    // e.g. repaint(), notify user about loading status
    if(areaGoal == 0 && (bits & HEIGHT) != 0 && (bits & WIDTH) != 0)
    { areaGoal = img.getHeight(this) * img.getWidth(this);
    }
    return true; // further imageUpdate()-calls are needed
} }
*/

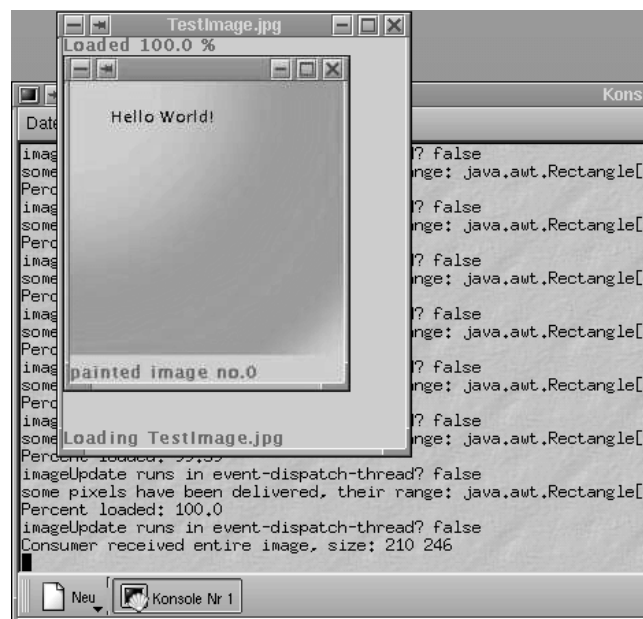
```

```

marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction > java ImageViewerInfo TestImage.png
imageUpdate runs in event-dispatch-thread? false
Now available, height of base image = 246 pixel
Now available, width of base image = 210 pixel
imageUpdate runs in event-dispatch-thread? false
some pixels have been delivered, their range: java.awt.Rectangle[x=0,y=0,width=210,height=1]
Percent loaded: 0.41
imageUpdate runs in event-dispatch-thread? false
some pixels have been delivered, their range: java.awt.Rectangle[x=0,y=1,width=210,height=1]
Percent loaded: 0.81
.....
imageUpdate runs in event-dispatch-thread? false
some pixels have been delivered, their range: java.awt.Rectangle[x=0,y=244,width=210,height=1]
Percent loaded: 99.59
imageUpdate runs in event-dispatch-thread? false
some pixels have been delivered, their range: java.awt.Rectangle[x=0,y=245,width=210,height=1]
Percent loaded: 100.0
imageUpdate runs in event-dispatch-thread? false
Consumer received entire image, size: 210 246

marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction >
*/

```



Here a coarse summary on meaning and usage of some of the interface-definitions:

java.awt.image.ImageProducer

ImageProducers in Java generate pixel-arrays. ImageProducers also generate pixel-arrays from data that comes from an Uniform Resource Locator (URL, denoting a resource on the local file-system or in a computer-network). The interface `java.awt.image.ImageProducer` is implemented by two classes of the `java.awt.image`-package: `java.awt.image.MemoryImageSource` and a class for relaying image-information `java.awt.image.FilteredImageSource`. These two classes implement the method `addConsumers()` and the consumer can order chunks of pixels by invoking the consumer-sided methods `setPixels()` and `newPixels()`.

java.awt.image.ImageConsumer

An ImageConsumer consumes pixel-arrays it receives from one or more ImageProducers. An ImageConsumer is registered with producers using the producer's `addConsumer()`-method. And then the consumer can fetch (parts of) pixel-arrays calling the `setPixels()`-method or, if a stream of images is expected, by calling `setAnimated(true)` and invoking the method `newPixels()`.

In most cases a Java application-programmer uses ImageConsumers as pre-programmed system software hidden from direct access. ImageProducers are coupled to ImageConsumers implicitly via the `createImage()`-method of a Component-object. The `java.awt.image.ImageConsumer` is explicitly implemented by the class `ImageFilter` and its sub-classes `java.awt.image.PixelGrabber`, `ImageFilter`, `CropImageFilter`, `ReplicateImageFilter` extends to `AreaAveragingScaleFilter` and `RGBImageFilter`.

OUTLOOK: Much like in the area of streams, where usually `FilterStream`-classes are extended and instantiated by application-programmers, rather than extending other stream-types; here application-programmers may tend to implement `ImageFilters` and `FilteredImageSources` rather than `ImageProducers` or `ImageConsumers`. For an extensive coverage of these mechanisms consult specialized books on Java graphics²⁶⁷.

Java Swing Desktop

Java Swing Desktop — As a special feature, the Swing package of Java contains the classes `JDesktopPane` and `JInternalFrame`. With the help of these classes a Java programmer can use a `JFrame` or a `JWindow`-object to create a sub-desktop on the original desktop of the underlying operating-system. A maximally sized `JWindow` which takes a `JDesktopPane`-object, even could try to substitute the original front-end of that native operating-system. The following program intends to give only an idea of the basic methodology in that context:

```
import javax.swing.JWindow;
import javax.swing.JDesktopPane;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JPopupMenu;
import javax.swing.JMenuItem;
import javax.swing.JInternalFrame;

class DesktopDemo
{
    public static void main( String unused[] )
    {
        JWindow window = new JWindow(); // JWindow is also a RootPaneContainer (interface)
        java.awt.Toolkit nativeTools = window.getToolkit();
        java.awt.Dimension dim = nativeTools.getScreenSize();
        window.setSize(dim);
        window.setLocation(0,0); // may be overridden by user-interface of OS (Linux)
        window.setVisible(true);

        JDesktopPane desktop = new JDesktopPane();
        System.out.println( desktop.getDesktopManager() );
        desktop.addMouseListener( new MouseClickListener() );
        window.setContentPane( desktop );
    }
}
```

²⁶⁷ Another source for the use of progress-bars is given by the Java TechTip of Sept. 12th of 2000.

```

        // an object of type JDesktopPane is also a JLayeredPane

        // JInternalFrame is a JComponent
        JInternalFrame ifr = new JInternalFrame("InternalFrame", true, true, true, false );
        // title, resizable, closable, maximizable, iconifiable
        ifr.setBounds( 10, 10, 200, 200 );
        desktop.add( ifr, new Integer(300) );
        ifr.show();
        ifr.setBackground( new java.awt.Color(255, 0, 0, 50) );

        javax.swing.JLabel lbl = new javax.swing.JLabel("This is a label");
        lbl.setBackground( java.awt.Color.green );
        lbl.setForeground( java.awt.Color.orange );
        lbl.setBounds(10,10,200,100);
        desktop.add( lbl, new Integer(301) );

        javax.swing.JInternalFrame[] content = desktop.getAllFrames();
        for( int i = 0; i < content.length; i++)
            System.out.println( content[i].getLayer() + "      "+ content[i] );
    } }

class MouseClickListener implements MouseListener
{
    public void mouseClicked( MouseEvent e ){ openPopup(e); }
    public void mouseEntered( MouseEvent e ){ openPopup(e); }
    public void mouseExited( MouseEvent e ){ }
    public void mousePressed( MouseEvent e ){ openPopup(e); }
    public void mouseReleased( MouseEvent e ){ openPopup(e); }
    private void openPopup(MouseEvent e)
    { if(e.isPopupTrigger())
      { ThePopupMenu x = new ThePopupMenu();
        x.show( (java.awt.Component)e.getSource() ,e.getX(),e.getY());
      } } }

class ThePopupMenu extends JPopupMenu implements ActionListener
{ ThePopupMenu()
{ super("Title");
  JMenuItem item;
  this.add( item = new JMenuItem("Exit"));
  item.addActionListener( this);
}
public void actionPerformed( ActionEvent e ){ System.exit( 0 ); }
}
/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction > javac DesktopDemo.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction > java DesktopDemo
(Exit command by pressing the auxiliary mouse-key for the popup-menu)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Interaction >
*/

```



Another approach to using the full screen is given as follows: Beginning with Java Version 1.4, the full-screen exclusive mode API of the AWT-package allows to draw directly on the underlying desktop of the native operating-system.

Java: Applets - Applet Runtime-Environment

Java: Applets - Applet Runtime-Environment — Generally when transferring (non-Java-)code over the Internet, the code has to anticipate the hardware-specifics often carrying entire, little used code-libraries. Being able to expect a full Java Runtime Environment (JRE) with an Abstract Window Toolkit (AWT) or even a Swing-package, makes the Java-code smaller; because the basic information, already distributed with the Java code-libraries, has been loaded from somewhere else earlier. Since the Internet is open to anyone, the code has to be checked for its integrity and authenticity or the code has to be given limited access to platform resources (for example denied access to the file-system or printers; Java-related procedures of that kind have been introduced in the chapters beginning on page 447).

Remember that a **Java application**²⁶⁸ is a Java program, given by a file of byte-code, triggered by the `java`-command to run in a Java Runtime Environment (JRE). The source-code of an **Java application** is given by a class with a static `main()`-method. An **applet**²⁶⁹ is given by Java source-code containing a class which extends the class `java.applet.Applet`. Then, triggered by the programmer, the Java compiler reads the text-file with the source-code to produce a file of Java byte-code. This Java byte-code is run in an applet runtime-environment (applet-sandbox) on top of a Java Runtime Environment (JRE). Implementations of that applet-sandbox are given by Java-enabled Internet-browsers or the software that can be started by an end-user issuing the “`appletviewer`”-command inside a terminal-window of the native operating-system. Programs like Java-enabled Internet-browsers, triggering the running of Java-code also are called **host-applications**. Ideally, all host-applications make an applet-window viewable AND clearly identifiable on the screen. So, some of the basic differences between applets and applications are

Specialized runtime-environment Applets run in a specialized applet runtime-environment (also called applet-sandbox) on top of a given Java Runtime Environment (JRE).

User configurable security The applet-sandbox usually allows end-users to directly (re-)configure the applet’s access to a selection of sensitive system’s resources. Before the advent of the Java security-infrastructure - as presented in one of the previous parts of this text - this feature was unique, in comparison to any other form of code-transfer from computer to computer (compare this to the text on page 475).

(Relatively easy Configurable Default Security) Ordinary class-files (= Java byte-code) or original (Java) source-code and can be freely transferred over networks. But class-files or Java programs or original (Java) source-code experience no special default security restrictions on their new host. (Java-)applets can only run in a software-environment like an applet-runner (or a Java-enabled Internet-browser) which allows the end-user to more or less easily restrict access to the host’s resources. Applet-runners (browser or appletviewer), which can be configured by the user, allow to restrict the applet’s access heavily so that it

- cannot read or write files on the client-platform
- can only open network-connections to the server-platform it has been loaded from
- cannot start other processes on the client-platform

Applets cannot have native methods (methods that are written in another programming-language like C/C++ and are platform-specific) because applets are meant to be portable from platform to platform.

These security aspects are the most striking features that distinguish an applet from, for example, a `java.awt.Frame`-class with a `main()`-method in which that `Frame`-class instantiates itself (see page 546).

²⁶⁸The application-concept has been described in the gray box on page 38 and on page 512. The relation of a Java application and the Java Runtime Environment is indicated on page 51. For triggering an application into running with the `java`-command, see the graphic and text around pages 66 and 431.

²⁶⁹The concept of an applet has been outlined on page 67 and on page 397. How to start an applet within an applet-viewing environment has been shown around page 67, on page 398 and following page 431.

Applets are screen-components The applet-sandbox relies fully on the Abstract Window Toolkit (`java.applet.Applet` extends `java.awt.Panel`) or the Swing-package (`javax.swing.JApplet`) for any form of on-screen representation of the applet.

(User Interaction) An applet is a screen-component, that means: Implementing `MouseListener`-interfaces and registering event-listeners on itself makes the customized event-handling methods receive keyboard- or mouse-events. Besides its four triggerable methods `init()`, `start()`, `stop()` and `destroy()`, this is another reason why `Applet`-objects usually are faster to configure for direct user-interaction than Java classes with a single static `main()`-method. In that sense an applet is a type of program "having an open ear on the screen, mouse and keyboard" (in the same way as every screen-component of the Abstract Window Toolkit has).

Applets have been mentioned already at various places in this text: Remember the section "How is run Java code?" concerning applets on page 67. There an **applet** has been shown to be a file of byte-code (in a class-file) compiled from a file of source-code containing a class that extends the class `java.applet.Applet` (or the extended class `javax.swing.JApplet`). And an applet has been introduced to run in an applet-runtime-environment within the Java Runtime Environment (JRE).

A more extensive introduction to the composition of applet-source-code has been given beginning with page 396. This included the writing of the source-code of an applet, the crucial aspects of running an applet and examples in multithreading (page 401).

Triggering And Configuring an Applet-Run

Triggering And Configuring an Applet-Run — The triggering of an applet-run inside its applet runtime-environment has been illustrated in the graphics on page 67 and on page 398. An end-user may enter a command ("appletviewer") into a terminal-window of the operating-system, but usually applets are activated by entering an URL (Uniform Resource Locator) into the command-line of some Java-enabled Web-browser. Either way of triggering loads HTML-code referencing the applet's byte-code: The HTML-tags `<applet ...>` reference the applet's file of byte-code, so that the screen-representation of the applet can be embedded into the screen-representation of the respective HTML-file. As demonstrated below, the `<applet ...>`-HTML-tags additionally allow the HTML-code to furnish attributed parameters for the applet.

```
This may be the text of an HTML-document <HR>
<applet
alt="no applet runner available in this browser"
<! Text displayed if this browser has no working implementation of an applet-runner.
    If the applet doesn't run and that 'alt'-String is not printed,
    then the browser even doesn't recognize applet-HTML-tags.
    (optional attribute, 'alt' stands for alternate text)
>
codebase="http://www.abc.com/folder1"
<! Relative or absolute URL (Uniform Resource Locator) of the folder
    in which can be found the applet's class-file. This tells the browser from where
    to retrieve the class-file with the applet's byte-code.
    The default value is the URL of the folder from where has been fetched
    the current HTML-file.
>
archive="libraries.jar"
<! URL of JAR- or ZIP-files,
    these may contain code referenced by the applet's code.
    The sandbox's equivalent of an 'external CLASSPATH'-variable (optional attribute).
>
code="HelloWorldApplet.class"
<! Name of the file containing the byte-code of the applet.
    (For global identity use names like "COM.abc.HelloWorldApplet.class".)
    (Either this attribute, or the 'object'-attribute has to occur.)
>
object="serializedApplet"
<! Name of file containing a serialized version of the applet, instead of
"code"-attribute, is loaded by the applet runtime-environment
    only with the appropriate class present
    and runs omitting the init()-method directly by calling the start()-method.
    Also can be used to let the applet run in a non-browser JRE.
>
width=200 <! width in pixels of screen representation of applet >
height=200 <! height in pixels of screen representation of applet >
name="This instance of applet"
<! Another applet in same HTML-document can get the reference to this instance,
```

```

    see page 404.
>
>
<PARAM NAME="externalStringVariable" VALUE="externalStringContent">
<! The 'PARAM'-tag is placed between the two applet HTML-tags.
    Thus the HTML-code can sent information to the applet,
    which takes them with the statement
    'String s = getParameter("externalStringVariable");'.
    Compared to the context of an application, this resembles faintly
    the optional String-variables in the 'java string string'-command,
    which become part of an array-parameter in the 'main( String [] args )'-method.
*/
</applet>

```

(align=left, right, top, middle, bottom, texttop, absmiddle, absbottom, baseline and vspace=10 and hspace=20 are other attributes for positioning the applet in the HTML-document)

Usually the class-file of byte-code has to be found in the same server-subdirectory as the HTML-code. The following (same as on page 400) is the applet's counterpart of the "HelloWorld"-application (page 71):

```

// source-code of the applet from which the class-file code has been generated

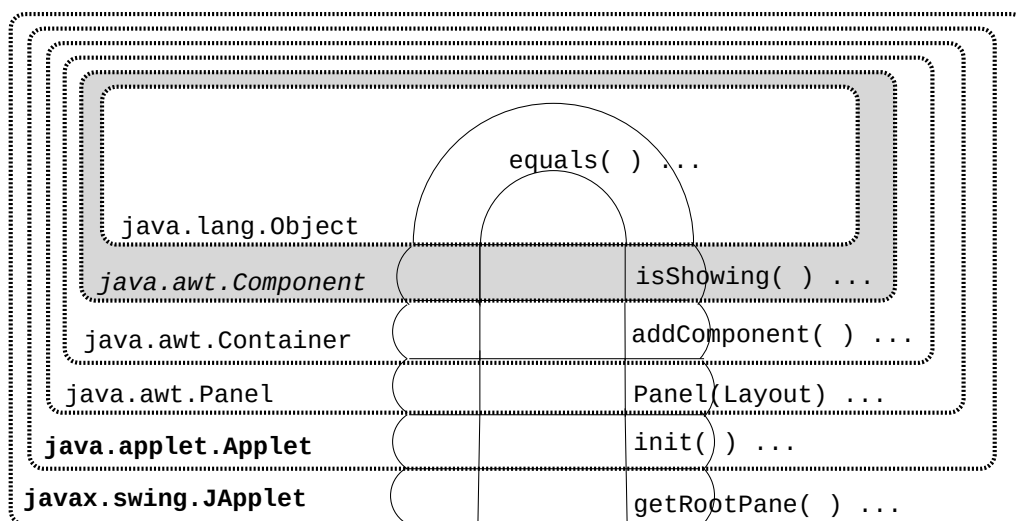
public class HelloWorldApplet extends java.applet.Applet
{
    public void paint( java.awt.Graphics g )
    {
        g.drawString( "Hello World!", 40, 30 );
    }
}
// same paint()-method with javax.swing.JApplet (NO paintComponent(), why?
// Because JApplet extends Applet and not JComponent. See also next section.
/*
C:\WINDOWS\JavaCodeFiles\SingleUserInterface\Applets>javac HelloWorldApplet.java

C:\WINDOWS\JavaCodeFiles\SingleUserInterface\Applets>
*/

```

Class-Hierarchy and Class-Structure of Applets

Class-Hierarchy and Class-Structure of Applets — Due to the applet's design as a notion of the Graphical User Interface (GUI), the description of its class-structuring had to be postponed until now:



(In the scheme above can be seen again: Package-names (java.lang, java.awt, javax.swing), class-hierarchy and file-names (Object, ..., JApplet) may not be related to each other.) The class Applet of the package java.applet is a specialized extension of the Panel → Container → Component → Object-hierarchy of the Abstract Window Toolkit (AWT). This includes the Swing-specific applet-class javax.swing.JApplet, therefore its Swing functionalities include only the implementation of the Swing-^{limited} interfaces javax.accessibility.Accessible and the javax.swing.RootPaneContainer which supports layered drawing! (Automated double buffering and refined features of painting in Swing are NOT supported!) ^{Swing-support of JApplet}

(Display)

Inherited	from	java.awt.Component,	(nothing new)
java.awt.Container,	uses	the	paint()-method.

(Position/Size)

Inherited	from	java.awt.Component,	(nothing new)
java.awt.Container	only if the host-software supports it.		

(Appearance)

Get the drawing area of the AWT-COMPONENT OF TYPE JApplet. All the methods specified by the javax.swing.RootPaneContainer are implemented in the class javax.swing.JApplet.	public JRootPane	getRootPane() JApplet

(I/O and Display continued from java.awt.Container)

Many browsers and appletviewers provide a text-field in which they display what URL the selected link has or how many percents of the file have been loaded. Calling this method puts the contents of the String-argument in that text-field. CAUTION: The browser may overwrite this output with newer information of its own. Therefore the String-message should not contain vital information for the user.	public void	showStatus(String message)

Returns a reference to system-specific code or a Java object that sub-classes java.awt.Image. This method always returns immediately, whether or not the image exists. Only when this applet attempts to draw the image on the screen, the data will be loaded (using prepareImage(), drawImage() or a MediaTracker-object). The graphics-primitives, which may draw the image, paint incrementally on the screen.	public Image	getImage(URL)

Same as above, only the file-name at the end of the URL - so that file exists - is substituted by the content of the String-object. The context of the String-object may represent a relative path in the hierarchy of folders ending with a file-name.	public Image	getImage(URL, String)

From the URL, specified by the URL-argument, this method returns a reference to an object which implements the java.applet.AudioClip-interface. This method always returns immediately, whether or not the audio-clip exists. When this applet attempts to play the audio-clip, the data will be loaded.	public AudioClip	getAudioClip(URL)

Same as with the getImage() and getAudioClip() method.	public AudioClip	getAudioClip(URL, String)

Plays the audio-clip found at the specified absolute URL. Nothing happens if the audio-clip cannot be found.	public void	play(URL)

Plays the audio-clip ...	public void	play(URL, String)

(Reflection continued from java.awt.Container)

Returns true, just before the applet's start()-method is called. Returns false immediately after the applet's stop()-method has been called.	public boolean	isActive()

Gives the URL of the HTML-document in which the applet is embedded. (Rather, gives a reference to the URL-object containing the URL ...)	public URL	getDocumentBase()

Gives the URL of the applet's class-file.	public URL	getCodeBase()

The HTML-tag-pair that causes the loading of the applet may contain a tag of the kind <PARAM NAME="Color" VALUE="green">. For the example above, invoking getParameter("Color") gives the String-value "green". The NAME-argument is not case-sensitive and returns null if the specified parameter-name is not found.	public String	getParameter(String name)

Returns a reference to a two-dimensional array of String-objects, to be defined by the applet's programmer. By convention , the first column is a list of names of all possible parameters the applet expects to receive from the HTML-code. The second column describes their respective types or ranges of values. And the third column can be used to furnish a description of the parameter.	public String[][]	getParameterInfo()
Returns a reference to a String-object that contains information about this applet. As before, the method and the String-content have to be defined by the applet's programmer. By convention the String-object should contain information about the author, version, and copyright of the applet. The default implementation is <code>public String getAppletInfo(){return null;;}</code> .	public String	getAppletInfo()
Returns a reference to an object that implements the interface <code>java.applet.AppletContext</code> . That implemented interface furnishes methods for getting information about other applets in the same document.	public AppletContext	getAppletContext()
Gets the Locale for the applet, if it has been set. If no Locale has been set, then a reference to the default Locale-object is returned.	public Locale	getLocale()
to be overridden with programmers code:		
After the applet's class-file has been loaded into the host application's Java Runtime Environment, this method is called by the applet-runner environment, the browser or an appletviewer. The method <code>init()</code> is called once at the very beginning in the applet-object's lifetime by the Java Runtime Environment. Sub-classing the Applet-class and overriding this method permits programmer-specified initialization . For example, an applet with threads, which are supposed to last the applet's lifetime, would use the <code>init()</code> -method to create the threads and the <code>destroy()</code> -method to terminate them. The default implementation is <code>public void init(){};</code> that means the method does nothing.	public void	init()
Called by the applet runtime-environment directly after the <code>init()</code> -method AND ideally then each time the Applet-object gets visible on the screen. In a browser-environment the method <code>start()</code> is invoked each time BEFORE the applet is reappears on a Web page on-screen. The programmer fills the <code>start()</code>-method with code to be worked on before the applet (re-)appears on the screen. For example, an applet with animation might use the <code>start()</code> -method to position the code for resuming the animation, and the <code>stop()</code> -method to locate the code for suspending the animation. The default implementation is <code>public void start(){};</code> that means the method does nothing except setting the applet's state into being active.	public void	start()

Called by the applet-runner environment ideally **AFTER** the Applet-object gets invisible on the screen AND before the Applet-object is garbage-collected. In a browser or appletviewer-environment the `stop()`-method is *usually* called when the Web-page, that contains this applet, has been replaced by another page or when another window becomes the top-level window. **The programmer fills the `stop()`-method with code that should be worked on AFTER the applet (temporarily) vanishes from the screen.** For example, an applet with animation could use the `start()`-method to resume animation, and the `stop()`-method to suspend the animation. **The default implementation is `public void stop(){};` that means the method does nothing except resetting the applet's state into being nonactive.**

`public void stop()`

Called by the applet runtime-environment - browser or appletviewer - to inform this applet that it is being completely deactivated and that it should release any resources it has currently allocated. The applet-runner environment always calls the `stop()`-method before the `destroy()`-method. A sub-class of Applet should override this method if it has any operation it wants to perform before instantiated objects are erased from the memory by the garbage-collector of the Java Runtime Environment. **The programmer fills the `destroy()`-method with code intended to run before the Applet-object is deallocated.** For example, an applet with threads which should last its lifetime, would use the `init()`-method to create the threads and the `destroy()`-method to terminate them. In that sense the `destroy()`-method somewhat resembles the `finalize()`-method (page 152) of the `java.lang.Object`-class. **The default implementation is `public void destroy(){};` meaning nothing happens before deallocating the applet.**

`public void destroy()`

When extending the class `java.applet.Applet` (that includes `javax.swing.JApplet`), the standard methods which are to be overridden by the Java programmer are: **`init()`, `start()`, `stop()`, `destroy()`**²⁷⁰. After having loaded an applet's class and instantiated an applet-object, the applet runtime-environment routinely calls these (triggerable) methods. The programmer usually only overrides them, implementing desired functionalities. This is the equivalent to the character of the triggerable `main()`-method in Java-applications, that is called automatically by the Java Runtime Environment; where the method `main()` has been filled before with the programmer's commands²⁷¹. "An application has only one `main()`-method, applets have four of the like, then the `paint()`-method for graphical tasks has not been counted."

OBSERVE that the class `javax.swing.JApplet` has NOT been derived from `JComponent` and thus lacks its entire Swing-specific infrastructure! EXCEPT for its extended functionality, given by implementing the interfaces `javax.accessibility.Accessible` and `javax.swing.RootPaneContainer`.

How to draw on such an applet? Override the `paint()`-method suitably (instead of the `paintComponent()`-method of a non-existent super-class like `JComponent`). BUT when painting or adding screen-components to the `JApplet`-object set or use its content-pane! This also enables the use of painting

²⁷⁰ Applets are destroyed, screen-components derived from `java.awt.Window` are disposed; when considered as Java objects all this ends up in a garbage-collection process for the Java objects!

²⁷¹ Source-code written by a Java programmer that does some initialization tasks is found in `init()`-methods of applets and overridden constructors of objects. Also static-initializers of classes (`Static{...}`) may contain that kind of source-code.

Source-code closing network-connections or databases, in general, source-code that terminates access, usually is found in `finalize()`-methods of objects of type `java.lang.Object` or in the `destroy()`-method of an applet.

The core ideas of the software usually are implemented by source-code managed from the static `main()`-method of applications or the `start()`-method of applets. When the host-application suspends the applet's function temporarily, it calls the `stop()`-method, overriding this method gives the programmer the opportunity to let the applet react to its suspension.

All these methods together with the event-handler-methods of the `java.awt.Listener`-interfaces are methods that are called by the Java Runtime Environment either on the beginning or end of some internal process (some constructors, finalizers, `start()`, `stop()`, `main()`) or some external interaction (processing `AWTEvent`-objects). These triggerable methods have to be overridden by the programmer to display the programmer-intended behavior.

or adding screen-components in different layers (as presented on page 566). Within a JApplet-object preferably use only Swing-components to avoid heavyweight/lightweight problems.

A Simple AppletViewer

A Simple AppletViewer — Essentially, an applet runtime-environment can be seen also as a simple Java-application for running applets²⁷². The only difference in principle may be the different means of invoking an applet runtime-environment, for example by issuing an Uniform Resource Locator from a browser-software or by issuing a command like “appletviewer” from a terminal-window of the native operating-system. The simple application below runs an applet, but of course without any security precaution and without activating the `stop()`- or the `destroy()`-methods, even if they have been implemented within the applet.

```
public class MiniAppletRunner extends java.awt.Frame
{
    public MiniAppletRunner( String s )
    { super(s);
      this.setSize(200,100);
    }
    public static void main( String fileNameWithoutClassSuffix[] )
    {
        String className = fileNameWithoutClassSuffix[0];
        MiniAppletRunner aFrame = new MiniAppletRunner( "MiniRunner" );
        aFrame.setVisible( true );
        java.applet.Applet anApplet = null;
        try
        { anApplet = (java.applet.Applet)Class.forName( className ).newInstance();
          aFrame.add("Center", anApplet);
          anApplet.init();
          anApplet.start();
        }
        catch( Exception e ){ System.out.println( e ); }
    }
}
/*
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Applets > javac MiniAppletRunner.java
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Applets > java MiniAppletRunner HelloWorldApplet
(Ctrl-C)
marycosway@linux:~/JavaCodeFiles/SingleUserInterface/Applets >
*/
```



Another specialty are applets given by source-code where the class that extends the `java.applet.Applet` (or `javax.swing.JApplet`)-class ALSO declares a `main()`-method! Yes, an Applet-class is free to declare a `main()`-method! The byte-code of those applets can be run like an application, for example by using the “java”-command from a terminal-window: The triggerable `main()`-method controls the code. But if the same code is called from an applet runtime-environment (entering an URL into a Java enabled HTML-browser which loads a HTML-text-file with applet HTML-tags), then the triggerable methods (`init()`, `start()`, `stop()`, `destroy()`) of the Applet-class control the code! From those triggerable methods of the applet (and the application’s `main()`-method) may be called common code, so that the same functionality of the source-code may be used in applet- or application-contexts! BUT if the code is run as an application, then all automatic security-restraints for applets become immaterial!

²⁷²The idea of applications that can run applications has been introduced past the source-code on page 442 and is presented fully developed on page 630.

Summary: Graphical User Interfaces (GUIs)

Summary: Graphical User Interfaces (GUIs) — Considering a Graphical User Interface means scrutinizing the interaction of an end-user with the computer especially with respect to mouse-, keyboard-input and image-oriented screen-output. The perceptual- and functional level of this interaction has been widely ignored in this text; meaning, the proper design of such an user-interface has not been mentioned. Solely, basic programming-related concepts have been introduced: First, at least three concepts of on-screen graphics can be discerned:

Standard screen-components like the `javax.swing.JFrame`-objects, the `JPanel`-, `JLabel`-, `JButton`-objects. Some of them appear on the native desktop, often as containers; other of these screen-components only appear as child-components within containers. Screen-components are more than on-screen images, screen-components may furnish input-services (accepting mouse- and keyboard-signals) and may change their visual appearance while keeping their identity. Inside a given `JFrame` or `JWindow`-object, instances of the (predefined) type `java.awt.LayoutManager` control the positioning of objects which represent such child-components. Mouse activities can be bound to visible components; additionally, keyboard activities can be processed by top-level frames or windows, especially by components inside them that are “on focus”.

Standard graphics-objects, like circles, rectangles, texts with specified font and size can be put on-screen by overriding a Swing-component's `paintComponent()`-method and, if necessary, by calling `repaint()`. A programmer's call of the `repaint()`-method uses the AWT-event infrastructure to eventually resulting in the system's call of the `paint()`-method putting the graphic on-screen. Programmers, calling `repaint()` instead of `paint()` or `paintComponent()`, integrate their command into the automated processing of GUI-related operations of the Java Runtime Environment; thus avoiding race-conditions which can lead to incoherent on-screen drawing.

Bitmap-graphics are made possible with the `createMemoryImage()`-method.

Keyboard- and mouse-events are relayed over the native operating-system to the Java Runtime Environment (JRE). There, these event-signals are associated with the screen-components at which they originated and transformed into Java event-objects. The Java event-delegation infrastructure distributes references to those event-objects to all registered listener-objects. These listeners implement handler-methods, which process the event-information taken from the event-objects.

Making data-input with keyboard and mouse truly interactive poses another challenge to programming in Java. For example, there are two specific time-critical problems to be addressed:

Speeding Animation: For speeding up animation standard tricks are: Redrawing only relevant clipped regions, and using double buffering by rendering an image-object off-screen and finally displaying the entire image using a fast internal copying-command.

Monitoring Loading-Processes: If some information doesn't reach the user fast enough, like loading a large file over the Internet, then the user should be informed about the progress of the loading.

The package `java.applet` together with some appletviewer-software, which furnishes an extended Java Runtime Environment, integrates the programming of a Graphical User Interface into programming pieces of code for some end-user.

Outlook: Single User Interface

Outlook: Single User Interface — The packages of the Abstract Window Toolkit (AWT) and Swing contain much more features than those presented or introduced here. Among those additional features are tree- and table-support given by the component frameworks of `javax.swing.JTree` and `javax.swing.JTable` which can be implemented with a fully-developed Model View Controller (MVC) concept.

In Java Runtime Systems (JRSs) of Version 1.2 and higher it is possible to change the overall style of Swing components uniformly within the entire Runtime Environment; this is even possible during the run-time of a program! The simple choice between “Java Look and Feel” and others (Mac OS, Motif, Windows) is possible with a relatively small piece of source-code. The development of own components or an own Look-and-Feel can be seen as a larger programming task.

Swing and AWT are supplemented by packages of the Java Foundation Classes (JFC): Including Java2D with further tools for drawing object-graphics. Java Drag and Drop for implementing easy copying or extracting and inserting. And the Java Accessibility infrastructure furnishing so-called assistive program features. These include helper-utilities for existing applications, like additional help menus. There are features for means that widen the number of channels of interaction, like speech recognition so that a wider range of end-users should be enabled to work with a given software product. This may include industrial climbers who cannot use their hands for interaction with a portable computer or physically impaired persons, generally enabling end-user with differing perceptive- and tactile conditions to still be able to use computing facilities.

Two other central aspects have been omitted or only lightly touched: Image-processing and text-processing: Text-processing, down to RTF²⁷³- and HTML-format rendering, is furnished by Swing. The Java Advanced Imaging (`java.jai`) package presents a wealth of means for processing two-dimensional images. The package Java3D makes it possible to construct three-dimensional spaces and to project them into two-dimensional planes.

This has been only an outlook on the programming tools that still are available. The general methods and skills in the design of interfaces for end-users, which are independent of a concrete programming-language, have not been mentioned. Also there are still remaining the means to use hardware like loud-speakers, printers, scanners, cameras, card-readers.

²⁷³“RTF” is an acronym formed from “Rich Text Format” (“HTML” is an acronym formed from “Hypertext Markup Language”)

Part VIII.

Appendix

In this appendix has been collected material which extends the information given in the preceding text, including alternative perspectives on some topics and simple lists with references into the text.

First, the **value -1** means different things in different context:

reading streams no further data available for reading or end of file (page 296, used for example in source-code on page 387 and on page 441)

network port-number use a given default port-number (page 388)

The following chapter about reflection, introspection and profiling compiles means of retrieving information from given entities, describing exactly these entities. With entities are meant objects, classes, the Java Runtime Environment and the underlying native platform.

Then there is presented a fully implemented event-infrastructure essentially given by a highly automated switchboard class. This central switchboard may have a relatively simple programming-interface, nevertheless, reading the overall demonstration-code may require some dedication.

In a succeeding chapter, have been collected some remarks on reading, writing and checking pieces of Java source-code.

Hints on performance, platform-specific programming features, and miscellaneous questions follow.

Finally have been compiled lists of programs, classes, interfaces, syntax-diagrams and sources of information as well as an index.

Reflection, Introspection and Profiling

Reflection, Introspection and Profiling — This text mentioned methods for getting knowledge about the context in which the classes (application-program, applet or servlet) are working. This chapter compiles the initial questions, some answers and references to the pages on which more answers can be found. Four environments are discerned: The Java Runtime Environment (JRE) with its classes and objects, the platform on which the JRE runs, the applet runtime-environment and servlet-engines.

Classes

Classes — How can be loaded a class into a Java Runtime Environment (JRE), this means how can the class made known to the JRE? How can a class, given as byte-code stored in a class-file, be transformed into its runtime-representation residing in the computer's RAM, there being available for use by the Java Runtime Environment? The standard case may be given by a class *name*, with the class-file *name.class* of byte-code inside a directory of the platform's CLASSPATH-variable: The loading takes place automatically when instantiating an object with the **new**-keyword:

```
name x = new name();
```

The class is also loaded by a call of the static factory-method **forName()** of the class `java.lang.Class` which also returns the reference of an object of type `java.lang.Class` from which can be drawn information about the loaded class:

```
Class c = Class.forName("className"); // takes a String-argument!  
// can also be used for pure class-loading from CLASSPATH
```

For a given class, or for the class of a given object (see next section), how to get the associated class-loader, the associated Class-object and the associated protection-domain? A reference to the class-loader can be got easily from a given Class-object by using the method `java.lang.Class.getClassLoader()`:

```
java.lang.ClassLoader cl = c.getClassLoader();
```

For example, if the class-loader is of the type `java.net.URLClassLoader`, then

```
java.net.URLClassLoader cl = c.getClassLoader();
```

gives access to all additional features related to Uniform Resource Locators and network access. If loaded by a(n extension of a) `java.security.SecureClassLoader`, like the `java.net.URLClassLoader`, then its `defineClass()`-method associated a protection-domain with each loaded class²⁷⁴ (page 433). The `java.lang.Class`-object of such a loaded class furnishes the method `getProtectionDomain()` which returns a reference to the associated protection-domain (page 468):

```
java.security.ProtectionDomain pd = c.getProtectionDomain();
```

Without further documentation: What members has the given class? For a given class or object (next section) list the members, variables and method-signatures: Get the associated Class-object and use one of its instance-methods:

```
java.lang.reflect.Method [] array = c.getDeclaredMethods();  
for( int i = 0; i < array.length; i++ ) System.out.println( array[i] );
```

²⁷⁴Classes loaded by the initial class-loader or an instance derived from the `java.lang.ClassLoader` have no protection-domain, or rather a null-protection-domain.

How to get the type-identifier, the class-name, transformed into a String-object out of the associated Class-object?

```
String name = c.getName();
```

How to call, for example, the `main()`-method of another class, even when **NO CLASS-NAME IS KNOWN AT COMPILE-TIME**? Those cases may use a call of the instance-method `java.lang.reflect.Method.invoke()`. In the case of the `main()`-method, a call means to start another application! The following mini-application-runner, an example of a rudimentary runtime-environment — on top of the given standard Java Runtime Environment — demonstrates this:

```
public class MiniApplicationRunner
{
    public static void main( String classNameAndParameters[] )
    {
        String className = classNameAndParameters[0]; // fileNameWithoutClassSuffix
        String[] args = new String[ classNameAndParameters.length-1 ];
        System.arraycopy(
            classNameAndParameters, 1, args, 0, classNameAndParameters.length-1 );
        Class argsGetClass = args.getClass();
        try
        { Class theClass = Class.forName( className );
        // for pure informational purposes:
            java.lang.reflect.Method[] methods = theClass.getDeclaredMethods();
            for( int i = 0; i<methods.length; i++ ) System.out.println( methods[i] );
        // the call: -----
            java.lang.reflect.Method theMethod =
                theClass.getMethod( "main", new Class[]{ argsGetClass } );
            Object result = theMethod.invoke( theClass, new Object[]{ args } );
        }
        catch( Exception e ){ System.out.println( e ); }
    }
}
// the string-array is ONE and the only argument
// in the parameter-list of the main()-method!
/*
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] marycosway% javac MiniApplicationRunner.java
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] marycosway% java MiniApplicationRunner HelloWorldCL
public static void HelloWorldCL.main(java.lang.String[])
public void HelloWorldCL.run()
  an object's ClassLoader-class: class sun.misc.Launcher$AppClassLoader
  an object's ClassLoader-object: sun.misc.Launcher$AppClassLoader@33065f
  a class's class-loader class: class sun.misc.Launcher$AppClassLoader
  a class's ClassLoader-object: sun.misc.Launcher$AppClassLoader@33065f
java.lang.Object has null-class-loader.
[localhost:~/JavaCodeFiles/JRESecurity/JREClassLoading] marycosway%
*/
```

Observe also that this program is run in the “class-loading”-folder as can be seen from the chosen directory in the print-out of the terminal-window. There, in the present example this program runs the “JavaHelloWorldCL”-class (page 435). Furthermore, keep in mind that this program calls a `main()`-method explicitly from some other `main()`-method!

If the name of the class is known at the time the runner is written, then the `main()`-method can be called directly:

```
String[] s = {}; className.main( s );
```

What classes are available for loading? How many and which objects are currently available? These questions seem to be answerable with the `java.lang.Package` class (page 245).

Objects

Objects — Let a reference “*objectName*” to an object be given: How to get the *class-type of that object*, independently of possible type-casts that affected the creation of the reference? The information on the class from which the object actually had been instantiated can be drawn from its associated object of type `java.lang.Class`:

```
Class c = objectName.getClass();
Class c = this.getClass() // inside a non-static method
```

How to get the name of the class of the object? (Answer: `objectName.getClass().getName()`) The class-name of a given object (unaffected by type-casts of its identifiers) can be printed out with the piece of source-code below:

```
try
{ Class info = objectNameOrTHIS.getClass()
  System.out.println( info.getName() );
}catch(Exception e){ e.printStackTrace(); }
```

The try-catch-clause is used in case the *objectName* returns null.

Further developing what has been said on constructors (pages 149, 166) consider a case of constructor-callback: Cross-instantiation as inner objects can be realized with the keyword *this*, that means one object implements the other one as an inner instance.

```
class Ping
{ Pong reflect;
  Ping(){ reflect = new Pong(this); System.out.println("ping"); }
}

class Pong
{ Ping caller;
  Pong(Ping p){ caller = p; System.out.println("pong"); }
}

class ConstructorCallback
{ public static void main( String args [] )
  { new Ping(); }
}
/*
marycosway@linux:~/JavaCodeFiles/Appendix > javac ConstructorCallback.java
marycosway@linux:~/JavaCodeFiles/Appendix > java ConstructorCallback
pong
ping
marycosway@linux:~/JavaCodeFiles/Appendix >
*/
```

Both objects know of each other because they have been given references of each other. Nevertheless one object instantiated the other.

System

System — On what kind of platform resides the current Java Runtime Environment? (page 258) Exists there another Java Runtime Environment? How to contact it? Needs deeper checking of the native platform. Is there a network-connection; where to find the local IP-address(es)? (source-code on page 382) How many free memory is left for the Java Runtime Environment (page 445); how many memory is left in the file-system? Needs deeper checking of the native platform.

Applets

Applets — Where does the document come from in which the applet occurs? Where does the applet's byte-code come from? (in the reflection part of the table on page 615) Where can be found the applet's source-code? If not told explicitly where to find it, then the programmer might hold the source-code back for security- or commercial reasons.

Servlets

Servlets — See the communication- and introspection-servlet on pages 412 and 411.

How to read Programs?

How to read programs? — First read the documentation or the comments, sometimes they contain hints towards the programming idea or a sketch of the concept. Reading programs often can be done economically by first sorting out known programming constructs with little new content. After that, the remaining complexities may be scrutinized. At this point is given an example of infrastructure-programming; a fully-implemented router/switchboard for objects of type `java.util.EventObject`:

```
/*
Alternatives: Method-calls and event-switchboarding
.
ADVANTAGES of using simple method-calls:
--- Simple, fast because of little programming and compiling overhead.
--- Direct contact and insight into the communicating entities.
DISadvantages of using simple method-calls:
--- The calling-class or -object has to be aware of the specific methods,
    which the called entity uses (the second item of the advantages).
--- Usually only one target entity per method-call available.
.
ADVANTAGES of using event-objects with software-switchboarding:
--- The communicating entities may only be coupled by the common knowledge
    of the message-sending protocol (another form of encapsulation).
--- A single event-object dispatched notifies multiple targets.
--- Full freedom of implementing multicast multisource message passing, for example in
    instantiating a switchboard- or adapter-object, that routes events to their handlers.
--- One of the real advantages of the event-model shows when multiple sources
    produce multiple event-objects whose references have to be channeled properly to multiple drains.
DISadvantages of using event-objects with software-switchboarding:
--- Large overhead of implementation of message-sending protocol for the infrastructure-programmer.
--- Additional learning for the application-programmer.
--- The programming overhead also shows in added consumption of processor-time.
*/
/* So a simple unisource unicast message or event passing
    usually can be substituted by a method-call.
The mechanism becomes beneficial in multisource multicast situations and some other circumstances:
-- Changes in design or existence of sources and drains of messages should be possible.
-- Registering and de-registering of events should be easy and localized at one point.
-- Registering should be done with sources and drains instead of messages and handlers.
(Hides the communication implementation.)
That is possible by using an adapter- or switchboard-object as a registry-center
for source- and drain-objects.
This adapter- or switchboard-object receives the event-objects from the registered sources
and forwards them to the appropriate handlers in the drains.
If there is only one drain-object,
then the adapter- or switchboard-object often is attached to the drain
by defining it as an inner class.
*/
// EVENT
// The class EventObject has
// --- a protected (transient(not part of serialization)) member-variable the Object "source",
// --- a constructor with the Object source as a parameter,
// --- and the methods public Object getSource() and public String toString()
// An extension of this class may introduce additional variables that transport data
// (even a reference to a Thread-object).
// An extension may define methods
// that transform data and by their association to the event-object
// are available to source AND drain.
// EVENT
class NumberEvent extends java.util.EventObject
{
    static int noOfEventInstantiation=0;
    int noOfEvent=0;
    int info;
    NumberEvent(Object source, int n)
    {
        super(source);
        noOfEventInstantiation++;
        noOfEvent=noOfEventInstantiation;
        info=n;
        System.out.println( source.getClass().getName() + " instantiated the event " +
                            noOfEvent + " with the number-data=" + info );
    }
}
//NumberEvent(){super(this.getClass().getDeclaringClass());}
// can't reference this before the super-class constructor has been called}
// end NumberEvent
/* The NumberEvent-class can count its instances
    with the static integer-variable noOfEventInstantiation.
    To identify the event each instance of NumberEvent gets an instance-variable noOfEvent
    with the current number. The instance-variable 'info' holds the data of the NumberEvent-object.
*/
```

```

    This event-class is extended:
*/
class TextNumberEvent extends NumberEvent
{ String messageText;
  TextNumberEvent( Object source, int n, String s )
  { super(source,n);
    messageText = s;
    System.out.println( source.getClass().getName()+ " and contains the text="+s );
  } } // end TextNumberEvent
//////////
/*
Now the programmer introduces an interface EventDispatcher,
this requires each class that implements EventDispatcher to implement two methods:
addEventListener() and removeEventListener().
Then two different kinds of sources are introduced,
--- one, SourceA that accesses the relay()-method of a Switchboard object directly;
--- the other, SourceB sends an event just by calling the
    dispatchTo(Switchboard, java.util.EventObject)-method.
*/
////////// SOURCES
interface EventDispatcher
{ void addEventListener( Switchboard l );
  void removeEventListener( Switchboard l );
}
// that interface tags sources and furnishes the names for standard methods
class SourceA implements EventDispatcher
{ SourceA(){
  Switchboard listener = null;
  public void addEventListener( Switchboard l ){ listener=l;};
  public void removeEventListener( Switchboard l ){ listener=null;};
  void activate(){ listener.relay( new NumberEvent(this,5) ); }
}
class SourceB implements EventDispatcher
{ SourceB(){
  Switchboard listener = null;
  public void addEventListener( Switchboard l ){ listener=l;};
  public void removeEventListener( Switchboard l ){ listener=null;};
  void dispatchTo(Switchboard l, java.util.EventObject evt){l.relay(evt);}
  void activate()
  { dispatchTo( listener, new NumberEvent(this,8) );
    dispatchTo( listener, new TextNumberEvent(this,9,"Mail!") );
  }
}
}
////////// DRAINS
/* There are two drain-classes:
   DrainX has two event-handlers that each take a different event-class and
   have different names. The drain-class DrainY has one event-handling method.
*/
class DrainX
{ DrainX(){
  public synchronized void workoverEvent( NumberEvent evt )
  { System.out.println( this.getClass().getName() + " has received event " + evt.noOfEvent +
    " with number=" + evt.info );
  }
  public synchronized void handleEvent( TextNumberEvent evt )
  { System.out.println( this.getClass().getName() + " has received event " +
    evt.noOfEvent + evt.getClass().getName() + " with number=" + evt.info +
    " text=" + evt.messageText );
  };
}
class DrainY
{ public synchronized void handleEvent( NumberEvent evt )
  { System.out.println( this.getClass().getName() + " has received event " + evt.noOfEvent +
    " with: " + (evt.info) );
  };
} // what if static?
class DrainZ
{ static void EventHandler( TextNumberEvent evt )
  { System.out.println
    ( DrainZ.class.getName() + " a static handler has received event " +
    evt.noOfEvent + " with: " + (evt.info) + "0000000000000" );
  };
}
////////// Switchboard
/* The Switchboard-class is the most complex part of this application:
The switchboard routes the event-objects from the sources to their appropriate drains.
The routing-mechanism is programmed by using Vector- and Hashtable-classes:
Arrays are found where a sequence of objects can be indexed by a natural number.
The java.util.Vector-class is used if the length of the sequence should be variable.
The java.util.Hashtable-class is used where variably many objects may be indexable by
references to other objects.
Here the hashtable is used as the center of the switching-process:
The indices (or keys) are event types and the objects are vectors of "objectnames.methods"
that are handlers of these event types.
The central method constructConnectionTableFromDrains(Object [] drains) does this work.

Here the java.lang.Class and the java.lang.reflect.* classes
are used extensively to analyze objects and their members during run-time.
( For a more detailed discussion of the java.lang.reflect package and the functionalities
of the java.lang.Class-class refer to Java Language Specification.)
*/
// Switchboard
class Switchboard implements java.util.EventListener

```

```

{ protected void registerSources(EventDispatcher[] sources)
{ for( int i = 0; i < sources.length; i++ ){ sources[i].addEventListener(this); };
// source-registering-process
}

protected java.util.Hashtable connections = new java.util.Hashtable(10,(float).5);
// capacity, load factor(0.0=fast 1.0=small)

protected java.util.Hashtable constructConnectionTableFromDrains(Object[] drains)
{ java.util.Hashtable table = new java.util.Hashtable();
// keys are Event types, values are vectors with the objectnames.methods
for( int i=0; i < drains.length; i++ )
{ java.lang.reflect.Method [] methodInfoArray = drains[i].getClass().getMethods();
for( int k=0; k < methodInfoArray.length; k++ )
{ if( methodInfoArray[k].getParameterTypes().length == 1 )
{ // only one parameter (extensible to testing on modifier "void")
boolean argumentIsEvent = false;
Class param = methodInfoArray[k].getParameterTypes()[0];
Class check = param;
while( check != null && argumentIsEvent == false )
{ // that parameter is of type sub-class of EventObject
if( check == java.util.EventObject.class )
{ argumentIsEvent = true; }else{ check = check.getSuperclass(); };
}; // loop avoidable if there were a class-method isSuperclass()
if( argumentIsEvent )
{ Object objectref = drains[i];
java.lang.reflect.Method method = methodInfoArray[k];
java.util.Vector vect;
if ( table.containsKey(param) )
{ vect = (java.util.Vector)table.get( param ); }
else{ vect = new java.util.Vector(); };
Object [] pair = new Object[]{ objectref, method }; // can be anonymized
vect.addElement(pair);
table.put( param, vect );
}; // end if argument of method is event
}; // end if method only one parameter
}; // end for methods count
}; // end for drains count
// testing before returning
//
//System.out.println(table.toString());
//for(java.util.Enumeration e=table.keys();e.hasMoreElements());
// {Object[] info={table.get(e.nextElement());System.out.println(" "+info);};
return table;
}
void relay( java.util.EventObject evt )
{
// the event reference is the parameter of the method,
// but the event-class is the argument of the hashtable!
Object [] param = { evt };
java.util.Vector targets = (java.util.Vector)connections.get( evt.getClass() );
// select the target references and their handling methods
for( int i=0; i < targets.size(); i++ )
{ // step through each target
Object [] objrefMethodPair = (Object [])targets.elementAt( i );
// array caster: (type []) - array-initializer: {...}
Object objectref = objrefMethodPair[0];
java.lang.reflect.Method method = (java.lang.reflect.Method)objrefMethodPair[1];
try{ method.invoke( objectref, param ); } // relay event to target handling method
catch( java.lang.reflect.InvocationTargetException e ){ e.printStackTrace(); }
catch(java.lang.IllegalAccessException e){ System.out.println(e); } // unnecessary
}; // end for
} // end relay
Switchboard( EventDispatcher[] sources, Object[] drains )
{ registerSources(sources);
connections = constructConnectionTableFromDrains( drains );
// now relay can route the events to their receiver(s)
} // end Switchboard constructor
} // end Switchboard-class
// if only one global switchboard useful then use static methods and make it uninstantiable
// by making the default constructor private
public class RoutingDemo
{ public static void main( String args[] )
{ System.out.println( "\n\nRouting Demonstration" );
SourceA sourceA = new SourceA();
SourceB sourceB = new SourceB();
DrainX drainX = new DrainX();
DrainY drainY = new DrainY();// or static? does not work
Switchboard router =
new Switchboard( new EventDispatcher[] {sourceA, sourceB},
new Object[] {drainX, drainY, DrainZ.class} );

sourceA.activate();
sourceB.activate();
sourceA.activate();
} // end main
} // end RoutingDemo
/*
marycosway@linux:~/JavaCodeFiles/Appendix > javac RoutingDemo.java
marycosway@linux:~/JavaCodeFiles/Appendix > java RoutingDemo
Routing Demonstration
SourceA instantiated the event 1 with the number-data=5
DrainX has received event 1 with number=5

```

```
DrainY has received event 1 with: 5
SourceB instantiated the event 2 with the number-data=8
DrainX has received event 2 with number=8
DrainY has received event 2 with: 8
SourceB instantiated the event 3 with the number-data=9
SourceB and contains the text=Mail!
DrainX has received event3TextNumberEvent with number=9 text=Mail!
SourceA instantiated the event 4 with the number-data=5
DrainX has received event 4 with number=5
DrainY has received event 4 with: 5
marycosway@linux:~/JavaCodeFiles/Appendix >
*/
```


How to write and check Programs?

How to write and check programs? — Program writing usually happens as an incremental check and build procedure. The more checking and conceptualizing is done at early stages, the less debugging on these levels becomes necessary afterwards. Under ideal circumstances, almost every incremental step, that produces an error-free working prototype of the final program, should be stored and even documented. Later, the development-process can be remembered and if errors occur – for example when changing hardware, operating-system or Java version – then, in some cases, the error might be easier trackable by reiterating the software-production-process.

incremental
process, stored
version-history

Writing a program usually begins with an idea, first developed into an implementation plan, before doing any programming. Especially if the idea involves complexities, conceptualizing and planning should be given extensive care beforehand. Then, the program may be developed in small working steps. Getting experience in the strategy of incremental programming may be frustrating and sometimes it is even easier to begin again with the accumulated knowledge from non-working versions. Even the experienced programmers often have to work by trial and error.

sometimes
beginning from
scratch after
several tries

Object-orientation becomes useful for larger programming projects, those which require extensive planning, and those which reuse pre-written code and of course those which are simply open to object-oriented structuring. Actual testing of code often gets easier using a simple procedural programming style. (Therefore in many of the examples in the text, the object-oriented programming style could be called an unnecessary organizational overhead in achieving the actual programming-task! Only their function as examples justifies that overhead.) But if the code is to be reused in the future, then a thoughtfully structured, well-documented object-oriented programming style may yield a lot of freedom because code — written once —, then might be easily reused.

(Smaller) tests
usually work
better when
programmed
procedurally

Larger and more
general tasks
usually are
programmed in
object-oriented
style

Being able to write Java programs proceduralized, object-oriented, multithreaded, packaged or in any other style, should be a matter of striving for ease of coding and attaining readability of code. THESE STYLES ARE MEANT AS HELPERS, NOT GOALS OR REQUIREMENTS. THE ULTIMATE GOAL IS A WORKING PROGRAM, with a source-code that may be easily revisable or/and extendable. The use of the support-mechanisms furnished by Java ideally has to be part of the individual choice of any programmer.

If Something goes Wrong

Some erroneous results persist? Check the associated identifiers, individually or even use a spell-checker. If found useful, put (temporarily) unwanted source-code into a “compiler option” `if(false){unwanted source-code}` or comment the code out. See also the footnote near page 323. Consider the use of debuggers for Java source-code.

Avoid catch-clauses that do nothing! Being unable to detect a thrown exception is a striking source of confusion in program-development! If an exception is actually caught, then the application-programmer should see what is happening!

```
try{ }catch( Exception e ){ } // AVOID, if possible
try{ }catch( Exception e ){ e.printStackTrace(); } // more helpful
```

The lines above already show effective ways of controlling source-code during development: Using print-out commands has to be seen a **debugging instrument of the utmost importance**:

```
System.out.println(identifier);
```

The following list gives a selection of commented compiler- and runtime messages. Many errors are simple keyboard typing errors, that gives another reason for keeping code as simple as possible.

Can't access protected method clone in class java.lang.Object The `clone()`-method has been called from code from outside a sub-class of the current class. Refer also to the discussion of access modifiers following the program on page 285.

ClassNotFoundException When transferring files of byte-code (the so-called class-files, named `className.class`) from one place to another, keep in mind that *one* text-file of source-code can produce *multiple* class-files. Losing one of them might result easily in a “ClassNotFoundException” error at runtime.

Can't find class: *className* The platform-variable CLASSPATH has to be set to the directory where the class exists OR the name of the class-file has to be identical with name of class that contains the `main()`-method.

No method matching *methodName()* found in class *className*. That means: Method not existent or protected method called in a non-sub-class and non-same-package environment.

No constructor matching *className(possibleArgumentType(s))*. Constructor not found in class *className* but constructor is there! Then it may have been forgotten to declare the constructor protected or public (inside a package).

Method *methodName()* not found in class *className*. Forgotten the `new`-keyword when using the constructor!

Public class *name* must be defined in a file called “*name.java*”. Only one public class per compilation unit, which generally is a file. This compiler message occurs when another public class is defined in the same text-file or if the name of the public class doesn't match the name of the file.

Undefined variable, class or package name. Often the `new`-keyword is missing when instantiating an object.

Undefined variable, Undefined method A default method (`getClassLoader()`) returns the reference of a default Object-type whereas the object itself is of an extended type. Then the reference of default type may have to be widened by casting to make the additional members of the initial class-extension known. To achieve this, the name of the class-extension has to be known in that piece of source-code. How can the name of the extended type made known to this environment? There are several approaches to that task:

1. The expression `objectRef.getClass()` gets the information for widening the type as much as possible²⁷⁵. Use `objectRef.getClass()` and the reflection package to access known extended features of the object.
2. Load the extended class-type with the modifier `public`, if possible.
3. Predefine an interface used both by the referencing-part and the declaring-part. (Example, beginning page 404)

²⁷⁵The expression `(objectRef.getClass()) objectRef` is valid, it gets no type-expression error.

How to make Programs faster?

Making programs faster needs not always contradict the idea of user-friendliness: Avoid unnecessary I/O(input/output)-operations. A simple, unglamorous but well-conceptionalized Graphical User Interface, may be less impressive for the end-user, but it may prove more intuitive to handle and thus reveal itself to be more efficient in the long run.

Keep source-code as small and as terse as possible.

With one loop within another: Keep the innermost loop extremely optimized because the code of the inner loops is re-run most often.

Sort cascading if-statements according to the probability in which their conditional expression is expected to return `true`. The if-statement, with the conditional expression of the highest probability to be affirmative, should appear in the outermost level.

Avoid multithreading or use as few threads as possible. Context-changes and time-slicing usually take processor-time. Calling synchronized methods also takes more time (`java.util.Vector, ...`).

Declare all methods `final` where possible; this makes them unoverridable but calling them takes less time. During compile-time, the identifier of `final` methods is statically bound to the method's code, thus the a method-call can directly access the code. (Late-bound methods, the default case in Java, do have to select their code on runtime, thus such a method-call takes more time. See also page 168.)

Avoid unnecessary structuring of source-code into classes and methods. Try to use as few classes and as few methods/method-calls as possible. This suggestion has to be balanced with the aspiration of writing understandable code.

Use `String`-objects when working with relatively small or constant strings. With larger or changeable strings rather use a `StringBuffer`-object.

Wrap streams into buffered stream-versions. Byte-streams usually are deemed to be faster than two-byte-streams of the `Reader/Writer`-type.

Use a Just in Time (JIT)-compiler, check the code-performance with the JDK-profiler functionalities.

There are numerous other approaches to the task of making Java byte-code run faster.

Further Questions

- **Where does the underlying platform shine through?** The native platform underlying the Java Runtime Environment can be seen when using platform-specific separators in paths of the Hierarchical File System (HFS) (page 48), in multithreading where the thread-scheduling (priority-based or additionally time-sliced) may depend drastically on the underlying (native) platform (page 203). Consider also the information given by the `System`-class and the control given by the `Runtime`-class (page 254 and the program on page 258). A platform-dependency also may be seen when using or importing `com.*`-packages, which are specifically written for that operating-system. And of course cases, unmentioned in this text, where the Java Native Interface (JNI) is used to reference platform-specific-code. How does a Java Runtime Environment and the underlying platform furnish a visible contact for the Java programmer and the end-user? See also the table on page 56 and the Java references to native code that are established by calling factory-like-methods (page 185).
- Which directories are used to store the software of the Java Runtime System? See page 249.
- What file in the file-system contains the implementation of the Java Virtual Machine (JVM), the byte-code interpreter? What file(s) constitute the stored Java Runtime System? `rt.jar` (before Java Version 1.1 also `classes.zip`) Consider the information given by the source-code and its printout on page 258 (property: `java.library.path`) and the printout on page 434 produced by the `-verbose`-option of the `"java"`-command.
- Where in the file-system can be found the pre-written packages and classes? See the question above and refer to page 249.
- How to get a list of all currently loaded classes of the Java Runtime Environment? This may be possible by inspecting the loaded packages (page 245). How to get a list of all loadable classes on the current platform? Listing all "loadable" classes would mean to inspect the current directory, the directories in the `CLASSPATH`-variable and in the `ext`-directory PLUS all the relative directory-paths made possible by package names. This would effectively mean to test virtually each file in the native file-system on being a class-file of Java byte-code.
- Which places in a class-definition can hold programmer-defined code? When is the respective code used? Static-initializer (page 152), instance-initializer (page 152), constructor (page 149), `main()`-method defining an application (page 127), `run()`-method of a Java-thread (page 205), finalizers of objects (page 152), `init()/start()/stop()/destroy()`-methods of applets (the graphic and text around page 398 and the end of the table beginning on page 615), `init()/service()/destroy()`-methods of servlets (page 407), any other triggerable method (page 127) and any other methods called from other methods.
- How is the Java Runtime Environment (including the implementation of the Java Virtual Machine, JVM) started? By entering the `"java className"`-command into the command-line of a terminal-based operating-system (like Unix-derivates or Windows)! By entering the Uniform Resource Locator (URL), pointing to a HTML-file with applet HTML-tags, into a Java-enabled HTML-browser. Or by entering `"appletviewer htmlTextwithAppletTags.html"` into a terminal-window, to run the applets referenced by the applet HTML-tags in the HTML-file. Or by starting a Java-enabled server with configured servlets, any servlet-activating Uniform Resource Identifier (URI) requesting the servlet's services activates the Java Runtime Environment (JRE).
- How to load multiple *public* classes/interfaces into a Java Runtime Environment if only one *public* class/interface per file is allowed?
 1. Put the source-code in multiple files to be found in the current directory. If the original source-code references the respective identifiers, then the various public pieces of source-code are automatically compiled and the respective byte-code is automatically loaded.

2. Define a package, place its class-files into one of the directories indicated by the CLASSPATH-variable, or jarred into the ext-folder, and reference the package members.
 3. Loading with `Class.forName()` only gives the `Class`-object, an instance given by the `newInstance()`-method is only referenced as an object of type `java.lang.Object`. How to get knowledge of members and how to access members? See the section on reflection, above.
 4. Use some `java.lang.ClassLoader`-object.
- What is the difference between an abstract class and an interface in Java? And what are they good for? Both cannot be implemented by a constructor (consider the case of some abstract classes and factory-like-methods, see page 184). A class can implement multiple interfaces but extends only one (or none)(abstract) class. An abstract class may have implemented methods whereas an interface gives only method-signatures and (quasi-)constants. An abstract class is a template for further class extension; interfaces can be used as templates or as markers. More is presented in the sections on these topics beginning on page 183.
 - What are the current platform's IP-address(es)? See the source-code on page 382.
 - Can one Java Runtime Environment (JRE) run more than one application? Yes, see the mini-application-runner on page 630, it could run multiple applications, apart from that it is an application itself, thus already realizing two applications running at the same time: Itself, and the program run by itself. Or see the two communicating applets that are run within one applet runtime-environment on page 404.
 - Who called the method which is currently run? Throw an exception and inspect the stack-trace.
 - How can the Java byte-code actually be seen? Open a class-file with a text-editor to see fragments. Better open a class-file with a hex-editor to see each individual byte of the class-file. Or, to see the byte-code as a sequence of commands for the virtual processor, open a terminal-window of the operating-system, chose the directory in which to find the class-file and enter the command `javap -c fileName`²⁷⁶. (*fileName* without the suffix "class")
 - What is the difference between a constructor and a factory-method? And what are factory-methods good for? Both are static methods, both instantiate a new object. A constructor, space-prefixed by the `new`-keyword, instantiates an object AND returns a reference to that object. Factory-methods can instantiate and return references to objects from anywhere. See also page 149 for constructors and page 151 for factory-methods. (Factory-like methods return a reference, but do not instantiate an object, see also page 185)
 - Why can `java.lang.String`-objects be only assigned another name and not modified, like the `java.lang.StringBuffer`-objects? (String-objects can only change their identifier, their value remains constant!) Non-editable strings apparently can be processed using lesser computing resources.
 - How to transform an `int` -value into a `String`-object and vice-versa? Use the static methods of the class `java.lang.Integer`: `valueOf( String )` and `toString( int )`. See also page 256 and page 257.
 - How can a method be un-overridden? In a Java Runtime Environment, this should be impossible from outside a class or from outside an object, see also page 175.
 - What special meaning may get constructors without arguments? This kind of constructor becomes useful when instantiating an object with the `Class.newInstance()`-method. — When is that necessary? Loading a class during runtime, for example in the `java.net.URL`-protocol and type-extension mechanism.
 - How to check an array on being empty? Using the expression `arrayName == null` doesn't work, but the expression `arrayName.length == 0` does work, see program `WhoIs` on page 382.
 - How can a Java program communicate with the outside world? Java's low-level of communication with the outside world consists of few mechanisms:

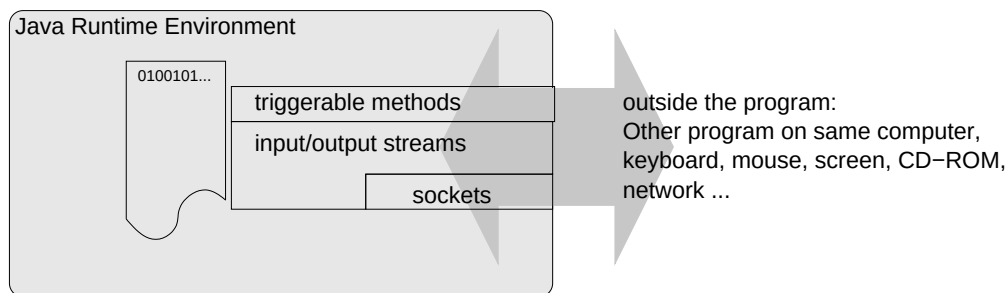
triggerable methods see page 127 for the basic idea, for their use in security-checking see page 462, for application in keyboard- and mouse-communication see page 573 and screen-updating see page 603.

²⁷⁶More on analyzing byte-code or its assembler-representation can be found in the TechTip of August of 2000.

streams one-way channels for data-transfer, see page 295

sockets “dial” into a computer-network, see page 273

other means, like factory-like-method and more, are listed on the table on page 56.



- Are TCP-ports also UDP-ports? Yes, in the sense that such a port-number may be assigned to a TCP-based service exclusive-or a UDP-based service.
- Are there other internal concepts of communication in Java?
The concepts above are extended and supplemented by some more problem-oriented (higher level) concepts of communication:

method calling is described extensively beginning with page 115. The external version is presented as Remote Method Invocation (RMI) on page 414.

thread-synchronization and spawning described in the part of this text on parallel programming, beginning with page 197.

Source-Event-Drain Concept has been applied widely from various implementations for internal communication (page 271) to networking (page 337).

Request and Response Objects Servlet programming uses request and response objects furnished by the servlet container. These request and response objects are interfaces for storing the incoming and outgoing data and additional information. (page 407)

List: Prototypes of Programs

List: Prototypes of Programs — These programs are intended to first have an introductory character, to show the realization of a specific programming feature. The programs are meant furnish a reference to a working implementation or to serve as a reminder. For more of the same kind search in the Internet, especially on *Sun's*-site for downloadable files containing so-called examplets. These are pieces of source-code that demonstrate the use of language- and package-constructs. These examplets also can be found inside the Java Developers Almanac [JDA].

The programs below can be downloaded from <http://www.chartingJava.com> collected into a JAR-file named `JavaCodeFiles.jar`. Unpack this JAR-file, as described around page 325, into your “`user.home`”-directory: Consider a terminal-window opened on this directory and enter the command:

```
user.home# jar -xvf JavaCodeFiles.jar
```

How to identify the “`user.home`”-directory has been shown in the source-code on page 258. The program on the security-related method-call-history (page 472) expects to find the code in the thus created directory-structure. The JAR-file contains software which writes data into the current directory. The JAR-file should ONLY contain source-code and text-files related to this book. Nevertheless check the source-code before compiling it.

Primitive-Types and Control Structures

- HelloWorld, one of the SMALLEST NON-TRIVIAL JAVA APPLICATION-PROGRAMS. Running the program produces the sentence “Hello World” on the standard output device (mostly a terminal-window). See page 71.
- Demonstrates how a Java programmer can define a type and instantiate variables of that type, page 76.
- Shows how characters can be tested for usability in a Java-identifier, page 82.
- Just demonstrates how to instantiate variables of primitive-type, page 87.
- How, one given byte, consisting of EIGHT BITS, REPRESENTS DIFFERENT NUMBERS, depending on interpreting this byte as representing data of different PRIMITIVE-TYPE (`int` or `byte`). How relying on automatic casting of primitive-types can produce surprises. See page 88.
- The use of `break`- and `continue`-statements in loops with a label. See page 112.

Code Structuring: Proceduralization-Methods

- HelloWorld with a (static) method (and a static variable). An elementary demonstration of how to define and to call a method, refer to page 116.
- Java methods take arguments of primitive-type by-value, these parameters are treated as local variables. Changes are valid only in the code-block of the method. Whereas changes to arguments of reference-type remain valid outside the method's code-block. See page 120 for the implementation.
- The first and elementary example of shadowing a variable within a method-block and un-shadowing it by using a qualified identifier. See page 124. More is presented in the source-code on page 167.
- How DIFFERENT METHODS WITH THE SAME NAME (BUT DIFFERENT ARGUMENT-LISTS) can be of use. See page 125.
- How a METHOD CAN “CALL ITSELF” or rather: How a method can call a working copy of itself. See page 126.
- How a METHOD RETURNS BY CALLING ANOTHER METHOD (and not by a `return`-statement). See page 127.

Code- and Data-Structuring: Object-Orientation: Class-Object

- A standard Java-class as a built-in type, here the `String`-class. A Java class as a programmer defined type, here the “`VarInteger`”-class (“`Pair`” has been defined on page 76). Here should be noted explicitly that a Java class also works as a program, given by the “`Program`”-class. See page 142.
- **RUNNABLE CODE:** Where to find it in a class and when that code is actually run. Static-initializer, instance-initializer, finalizer, `main()`-method. See page 153.
- The Java class-construct and its use of class-members (usually called static members) and instance-members. See page 155.
- Implements static- inner and local classes, page 158.
- As arguments of methods and in return-statements: Instantiations of class-definitions without giving the class or the instance a name (creating and using “Anonymous classes” and “anonymous objects”). See page 159.
- Identity, state and behavior are the conceptual characteristics of an object. How to assert the identity of a Java-object? See page 161.
- Comparing objects — asserting their equality. See page 162.

Code Structuring: Object-Orientation: Class-Extension

- Extending a class — adding new functionality, reusing code. See page 163.
- Extension and member-access, page 165.
- Extension, for shadowing variables and overriding methods see the source-code on page 167. (Remember also the shadowing of variables in method-declarations as demonstrated on page 124.)
- Overriding methods, shadowing variables **FROM INSIDE** THE SOURCE-CODE OF A CLASS-EXTENSION: How to un-override methods and how to un-shadow the variables using the keywords `super` and `this`. See page 174.
- Overriding methods, shadowing variables **FROM OUTSIDE** THE SOURCE-CODE OF A CLASS-EXTENSION: How to un-shadow variables by casting. See page 175.
- From within the same class-type, private members can modify each other. See the footnote on page 176.
- Demonstrates the different effects of the different access-modifiers `public`, `protected` and `private`. Beginning on page 177.
- A class-template (abstract class) and co-templates (interfaces) for a class. See page 186.
- Simple cloning with Java-specific overriding to assure a proper `clone()`-method. See page 192.

Code Structuring: Multithreading

- Determine whether the current platform supports **TIME-SLICED SCHEDULING OR NOT**. See page 204.
- **RUNNABLE CODE:** The extended `Thread`-object contains the `run()`-method. See page 207.
- **RUNNABLE CODE:** The argument of the `Thread`-constructor is an anonymous implementation of the `Runnable`-interface. See page 208. Useful for starting small anonymous threads.
- **RUNNABLE CODE:** The argument of the `Thread`-constructor references an object with a `run()`-method, thus implementing the `Runnable`-interface (an object of interface-type `Runnable`). See page 209.
- **RUNNABLE CODE:** The `Thread`-constructor references the enclosing object with the `this`-keyword. The enclosing class-type implements the `Runnable`-interface, that means it contains a `run()`-method. See page 209. There is shown an object that encapsulates the complete thread and its management.

- Demonstrates how a communication-error between two threads (a so-called race-condition) can occur, page 217.
- **EXCLUSIVE ACCESS TO AN OBJECT** permits to fix the order of changes on the objects. See page 219.
- **WAITING** for another thread to get ready. See page 221.
- Priority and Starvation (a thread is deprived of processor-time) illustrated. See page 224.
- **Deadlock** — each thread waits for the other to react. See page 226.
- **Lifelock** — each thread waits for the other to finish, actually a more agitated form of a deadlock, see page 226.
- Asynchronous thread-communication. Threads communicating over a buffer. See page 228.

Code Structuring: Aggregation: Arrays and Container-Classes

- How to use an **ARRAY**, an indexed aggregation of objects. See page 235.

Code Structuring: Packaging and Libraries

- Declaring a simple package, just furnishing a “Print(String s)”-constructor which calls the `System.out.println()`-method. See page 241.
- Using the class of a self-declared package. See page 245.
- Retrieving and setting information about the current Java platform, see page 258.
- Retrieving information about the files and the file-system of the current Java platform. See page 259.
- How to acquire and format-, date- and time-data, see page 262.
- International HelloWorld with **text**-files as database, see page 263.
- International HelloWorld with **class**-files as database, see page 264.
- Lists the fonts available on the current platform for use in the Java Runtime Environment, see page 266, continued on page 569.

Communication: Messaging

- Branching by throwing objects, or rather by throwing the references of objects of class-type **Throwable** (`java.lang`). See page 283.
- Communication using `java.util.Observable/Observer` programming-interfaces. See page 285.
- Communication using `java.util.EventObjects`. Actually this program could be reduced to a simple method-call, but the program gives an impression of the programming overhead necessary when implementing a messaging-infrastructure of one's own. See page 288.
- Communication using `java.util.EventObjects`, and a switchboard-object with full flexibility, see page 633.

Communication: Streams and Pipes, Serialization (package `java.io`)

- Stream-class `java.io.StringReader`: The characters of a string are printed out - one-by-one with a time-delay, see page 291.
- Stream-objects `System.in/out`: The first character of a 'return'-terminated sequence of keyboard key-presses has its ASCII/Unicode-code-number returned. This is the first example of entering data into a running Java program! See page 297.
- Stream-class `java.io.FileInputStream`: How files are opened and made ready to be read. See page 298.
- Stream-class `java.io.FileOutputStream`: Reading and writing a file, see page 298.

- Class `java.io.RandomAccessFile`: Handling a random-access-file, see page 299.
- Stream-classes `java.io.PipedWriter/PipedReader`: Demonstration of a pipe, see page 300.
- Stream-class `java.io.FilterReader`: A generic parser, searching for a character in a stream, see page 304.
- Stream-class `java.io.ObjectOutputStream/ObjectInputStream`: Serializing an object, transforming an object into a byte-sequence, see page 309. The graph of object-references used here is also a simple example of type-recursion.

Communication: Component/Bean-Based Programming, Documentation

- An example of a software-component according to the Bean-specification in Java. (An object that communicates by messaging and that follows special conventions.) See page 313.
- A property-change is communicated to registered listeners, on page 315.
- A property change is aborted by throwing an exception if one of the registered listeners does not consent to the change, see page 316.
- How to generate an automated documentation, see page 324.

Communication: Computer-to-Computer: IP (Internet Protocol)-Networks (package `java.net`)

- Using the Internet Domain Name Service (DNS) to resolve IP-address and domain-name relations. See page 382. This needs a connection to an IP-network with working DNS-servers! Some Intranets may not furnish this service, but the Internet usually will.
- A client of the WHOIS-protocol, a program that asks for information about the holder of the domain-name. See page 382. This needs an IP-network with WHOIS-services, the Internet usually will provide this service.
- A UDP (User Datagram Protocol) demonstration of sending and receiving packets of data that contain a `String`-object. See page 384. This only needs a connection to an IP-network.
- One of the smaller demonstration of TCP-clients that does nothing but establish a connection. The working program can be recognized only by an absence of error-messages. See page 385. This only needs a connection to an IP-network.
- One of the smaller demonstration TCP-servers that does nothing but accept a connection of the client mentioned above. The working program can be recognized only by an absence of error-messages. See page 386. This only needs a connection to an IP-network.
- A rudimentary HTTP-server: The client-request string "`GET /pathAndFileName HTTP 1.0`" is cut down to its `pathAndFileName` part; with which the file is retrieved and sent as a stream of bytes. See page 386.
- A few-liner of a workable HTTP-client; simply enter a domain-name and get the HTTP-answer printed into the standard output (window). See page 387.
- A simple client that uses the `java.net.URL`-class and its automatically configured application-protocol handlers to receive pure content-data. The URL-strings (like `http://java.sun.com`) are entered as arguments when running the "`URLClient`"-class in a terminal-window of the platform (`java URLClient http://java.sun.com`). The received application-data is printed into the standard output window as a sequence of strings. See page 390.
- A demonstration of the full `java.net.URL`-mechanism, with protocol- and content-handlers which can be introduced at run-time. See page 394.
- One of the smaller Java applets. See page 400.
- One of the smallest threaded Java applets. See page 401.
- An applet that manages a thread. See page 401.

- An applet that gathers information about itself and its environment. See page 403.
- An applet that communicates with another applet. See page 404.
- One of the smaller *servlets*. See page 408.
- A *servlet* that gathers information about itself and its environment. See page 411.
- A *servlet* that communicates with another *servlet*. See page 412.
- A simple *servlet* that communicates over the HTTPProtocol. See page 413.
- A demonstration of the RMI (Remote Method Invocation)-mechanism. A sort of “HelloOtherJavaPlatform” application-program. See page 417.

Java Runtime Environment, Class-Loading and Security: Class-Loading

- Identifying the class-loaders of the `java.lang.Object`-class and the current application. See page 435.
- An object of the `java.net.URLClassLoader`-class is a fully-functional class-loader: The NAME OF A CLASS plus a list of UNIFORM RESOURCE LOCATORS and a CALL OF ITS `loadClass()`-METHOD is all this class-loader needs to make this class available to the Java Runtime Environment (JRE). See page 439.
- A simple class-loader that transforms an array of bytes into a runtime-representation of the class. In this example a thread is run. See page 441.
- Calling methods whose names and parameter-lists have not been known previously, see page 442.
- Get information about used space of the computer’s Random Access Memory, while trying to let it overflow by producing a lot of `String`-objects. See page 445.

Java Runtime Environment, Class-Loading and Security: Policy-Based Security

- This program reads files as Unicode-text-files (every character is coded by two bytes) and appends the contents of one file to the other file. This needs reading and writing of files and can be used by users to test the user-related security properties of the Java Runtime Environment (JRE). At the end this program reads a system-property. See page 456.
- An simple example of a custom security-manager. See page 464.
- Reads out a list of code-sources and associated permissions to be found in the current valid policy. See page 466.
- A very simple example of a custom-written permission. See page 467.
- Reads out the list of code-sources and associated permissions to be found for the current class-file; contrary to any standard policy, this includes also the permission to read from the code source’s directory and its subdirectories. See page 468.
- Demonstration of the `PrivilegedAction`-interface and the use of an `AccessControlContext`. The directory-structure containing the class-files has to be rooted in the “`user.home`”-directory. The `.java.policy` text-file has to be edited too. See page 472.
- How to make an object a policy-protected system resource. See page 474.

Java Runtime Environment, Class-Loading and Security: Security by Data-Processing

- A simple program that shifts the letters of the alphabet in an input so that it has the effect of encrypting a message. This is also an implementation of the so-called ROT13 algorithm. See page 478.
- Which implementations of engine-classes and corresponding providers are present in this Java Runtime System? See page 486.
- How to generate a key-pair. How to take the private key and a file and produce a signature-file. How to export the public key and the signature of the given file. See page 493.

- How to use the previous public key, the signature-file and the given file to authenticate the file. See page 494.
- How to sign an object. (How to sign a class? With the jar- and the jarsigner-tools or by signing the class-file.) See page 494
- How to read any key-store that furnishes an instantiation of the class `java.security.KeyStore`. See page 495.
- This shows how a string can be encrypted. See page 497.
- Here is implemented the ROT-algorithm within the Java Provider-Infrastructure. See page 498.
- Here the encryption-process using the ROT-algorithm within the Java Security Provider Infrastructure is demonstrated. CANNOT BE RUN. See page 499.

Single User Interface: Graphical User Interface (GUI) (package `java.awt`, `javax.swing`)

- A picture-viewer in Java: Takes picture-files and displays them on the screen. See page 527. A version that informs about the loading status can be seen on page 607.
- Constructing an image from single bytes of pixel-information, where the values are chosen at random. See page 530.
- Constructing an image from single bytes of pixel-information, where the values are chosen so that the image displays rings. See page 531.
- Painting lines and strings by using a graphics-toolbox, see page 538. The drawing-surface is given by an EXTRA CLASS.
- Painting ovals and strings by using a graphics-toolbox, see page 539. The drawing-surface is given by EXTENDING THE CLASS that contains the `main()`-method.
- Painting ovals and strings by using a graphics-toolbox, see page 539. The drawing-surface is given by AN INNER CLASS of that class which contains the `main()`-method.
- Adding a screen-component (`JLabel`) to a top-level-container (`JFrame`). See page 544.
- Adding a screen-component (`JButton`) to a top-level-container (`JFrame`). See page 544.
- Adding a screen-component (`JList`) to a top-level-container (`JWindow`). See page 545. Here the class that contains the `main()`-method extends the class `JWindow` right away.
- Adding a screen-component (`JMenuBar`) to a top-level-container (`JFrame`). See page 546. Here the class that contains the `main()`-method extends the class `JFrame` right away and implements an interface `ActionListener`.
- Adding TWO screen-components (`JSlider`, `JProgressBar`) to a top-level-container (`JFrame`). See page 547. Here the class that contains the `main()`-method extends the class `JWindow` right away and implements an interface `ChangeListener`.
- The official "HelloWorld" window given by a `JFrame`-object. See page 561.
- An example of Swing's `JRootPane`-object and its various functionalities (layering), see page 568.
- Shows the available text-fonts on-screen, see page 569, prepared on page 266.
- Prints out all the message-objects generated within the current Java Runtime Environment, listed on page 579.
- Event propagation of the computer's mouse, fully demonstrated with interfaces like `MouseListener` and `MouseMotionListener`, see page 582.
- Event propagation of the computer's keyboard, demonstrated with the interface `KeyListener`, see page 584.
- Handler-methods running in the event-dispatch-thread: Large, resource-consuming methods block event-management with its `EventQueue`-object. See page 588.

- Handler-methods running in the event-dispatch-thread: Small handler-methods that call non-blocking methods, or explicitly spawn threads of their own, ease event-management. See page 589
- An object that “throws event-objects”, or rather instantiates an event-object and throws its references. See page 592.
- Interaction: For a label that listens to mouse-events see page 600.
- Interaction: Tracing mouse-movements in Java is shown on page 601.
- Animation: How to produce concentric waves is shown on page 603.
- A picture-viewer like the one introduced on page 527, but this one informs about the loading status. See page 607.
- A rudimentary Java-specific desktop on top of the operating-system’s one is introduced on page 610.
- For how to program a simple kind of applet runtime-environment, see page 619.

Appendix

- A program which can call the `main()`-method of another class, an application-runner, see page 630.
- An example of the implemented framework around the class `java.util.EventObject` (page 286). Also an example of a larger piece of source-code, see page 633.

When application-programs, those that are usually started from a terminal-window of the operating-system with the “java”-command, take arguments, they can be given a minimal user-interface like in the following pseudo-code:

```
public static void main( String args [] )
{ if( args.length != 2 )
    System.println("Usage: java programName argument1 argument2");
  else { theRestOfTheSourceCode }
}
```

This may be refined according to optional arguments. (Instead of using an `else`-clause, the `if`-clause may simply use a `System.exit(0);` statement, but that may be forbidden by a security policy.) For an example see page 382 or see page 610.

List: Mentioned Packages, Classes and Interfaces

List: Mentioned Packages, Classes and Interfaces — Classes and objects have been introduced in iglu notation on page 131. In this notation the notion of a class-structure and that of a class-hierarchy could be clarified (page 135). The class-structure has been visualized also by simplified extension diagrams (introduced on page 186, and used extensively beginning with page 562).

The following constitutes only a selection of classes. This selection solely reflects the striving for showing key functions in the programming context, not application-related proficiency. Java programming may require to carefully consider the details of the classes, which could not be anticipated in this introductory text. As already mentioned, for an up-to-date information consult the ➡ ➡ Java Platform API Specification [JPAPIS] or specialized books.

Check, if the package, class or interface of interest have been mentioned here, then consult the index to locate the place in this text, or consult the index right away: (sorted alphabetically)

java.applet Applet, AppletContext

java.awt AWTEvent, AWTEventMulticaster, BorderLayout, Button, Canvas, CardLayout, Color, Component, Container, Dialog, Dimension, EventQueue, FlowLayout, Font, Frame, Graphics, GraphicsEnvironment, GridBagLayout, GridLayout, Image, Inset, Label, LayoutManager, List, Menu, MenuBar, Panel, Point, PopupMenu, Rectangle, Scrollbar, ScrollPane, SystemColor, Toolkit, Window

java.awt.event ActionEvent, ActionListener, ComponentEvent, ComponentListener, ContainerEvent, ContainerListener, FocusEvent, FocusListener, InputEvent, ItemEvent, ItemListener, KeyAdapter, KeyEvent, KeyListener, MouseAdapter, MouseEvent, MouseListener, MouseMotionListener, PaintEvent, Event, WindowListener

java.awt.image BufferedImage, ColorModel, DirectColorModel, FilteredImageSource, ImageConsumer, ImageObserver, ImageProducer, IndexColorModel, MemoryImageSource, PixelGrabber

java.beans BeanInfo, PropertyChangeEvent, PropertyChangeListener, PropertyChangeSupport, VetoableChangeListener, VetoableChangeSupport

java.io BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, DataInputStream, DataOutputStream, File, FileInputStream, FileOutputStream, FilePermission, FileReader, FileWriter, InputStream, InputStreamReader, IOException, ObjectInputStream, ObjectOutputStream, OutputStream, OutputStreamWriter, PipedInputStream, PipedOutputStream, PipedReader, PipedWriter, PrintStream, PrintWriter, RandomAccessFile, Reader, Serializable, StringBufferInputStream, StringReader, StringWriter, Writer

java.lang ArithmeticException, ArrayIndexOutOfBoundsException, Boolean, Byte, Character, Class, ClassCastException, ClassLoader, ClassNotFoundException, Cloneable, CloneNotSupportedException, Compiler, Double, Error, Exception, Float, IllegalAccessError, IllegalAccessException, IndexOutOfBoundsException, Integer, InternalError, Long, Math, NoClassDefFoundError, NullPointerException, Number, Object, OutOfMemoryError, Package, Process, Runnable, Runtime, RuntimeException, RuntimePermission, SecurityException, SecurityManager, Short, StackOverflowError, String, StringBuffer, System, Thread, ThreadGroup, Throwable, UnknownError, VirtualMachineError, Void

java.lang.reflect Array, Constructor, Field, Method, Modifier

java.net ContentHandler, ContentHandlerFactory, DatagramPacket, DatagramSocket, InetAddress, ServerSocket, Socket, SocketPermission, URL, URLClassLoader, URLConnection, URLStreamHandler, URLStreamHandlerFactory

java.rmi Naming, Remote

java.security AccessControlContext, AccessController, AlgorithmParameterGenerator, AlgorithmParameters, AllPermission, BasicPermission, Certificate, CodeSource, GuardedObject, Key, KeyFactory, KeyPair, KeyPairGenerator, KeyStore, MessageDigest, Permission, PermissionCollection, Permissions, Policy, PrivateKey, PrivilegedAction, PrivilegedActionException, PrivilegedExceptionAction, ProtectionDomain, Provider, PublicKey, SecureClassLoader, Security, SecurityPermission, Signature, SignedObject, UnresolvedPermission

java.security.cert Certificate, CertificateFactory

java.util AbstractList, Arrays, Calendar, Collection, Date, EventListener, EventObject, GregorianCalendar, HashTable, List, Locale, Map, Observable, Observer, Properties, PropertyPermission, Random, ResourceBundle, StringTokenizer, Vector

java.util.zip ZipFile

javax.swing AbstractButton, JApplet, JButton, JComponent, JDesktopPane, JDialog, JFrame, JInternalFrame, JLabel, JLayeredPane, JList, JMenu, JMenuBar, JMenuItem, JPanel, JPopupMenu, JProgressBar, JRootPane, JSlider, JToolTip, JWindow, RepaintManager, RootPaneContainer, SwingUtilities, Timer

javax.swing.event ChangeEvent, ChangeListener, ListSelectionEvent, ListSelectionListener

List: Syntax-Diagrams

List: Syntax-Diagrams — Syntax- or railroad-diagrams are the graphical equivalent of Backus-Naur formed (BNF) specifications of the syntax of a language. These specifications outline combinations of Java language-keywords and literals. Disobeying these rules should immediately produce an un-permitted syntactic construct. But solely following these rules means “to use a grammar-book for writing a novel”. These specifications should serve as a means for reference and for guidance when in doubt with a concrete formulation, but they tell very little about the actual programming-process. The first classification of that kind occurs on page 76, the last on page 323.

Backus-Naur Form (BNF)

(o) means the symbol “o” occurs **ZERO** or **ONE** time, this means also that the occurrence of “o” is optional:

(none)

o

{ m } means the symbol “m” occurs **ZERO** or **MANY** times, thus the symbol “m” can occur 0, 1, 2, 3, ... times:

(none)

m

m m

...

< n > means the symbol “n” occurs **AT LEAST ONCE**, thus the symbol “n” can occur 1, 2, 3, ... times:

n

n n

...

e | o means **either** the symbol “e” **or** the symbol “o” occurs once

(...) parentheses indicate precedence

“(”, “)”, “{”, “}”, “<”, “>”, “|”, “(”, “)” denote literal occurrences of bracket, braces, larger and smaller symbols, bar and parentheses

Syntax- or Railroad-Diagrams

Syntax- or railroad-diagrams should be understandable intuitively: They are read from left to right by following the lines. Arrows pointing from right to left reserve that line for following in the non-standard direction. Keywords or literals are given a rounded frame, whereas names of other syntactic entities are given a frame with edges.

List

This lists the denotations and links of what is also called the grammar of the Java programming-language:

AbstractMethodModifier (184), AdditiveExpression (94), AmbiguousName (82), AndExpression (98), ArgumentList (119), ArrayAccess (235), ArrayCreationExpression (234), ArrayInitializer (236), ArrayType (234), Assignment (99), AssignmentOperator (99), Block (105), BooleanLiteral (79), BreakStatement (110), CastExpression (101), CatchClause (283), CharacterLiteral (80), ClassBody (146), ClassDeclaration (146), ClassInstanceDeclarationExpression (151), ClassLiteral (437), ClassMemberDeclaration (147), ClassModifier (146), ClassName (241), ClassOrInterfaceType (143), ClassType (146), Comment (323), CompilationUnit (246), ConditionalAndExpression (98), ConditionalExpression (99), ConditionalOrExpression (99), ConstantDeclaration (188), ConstructorBody (150), ConstructorDeclaration (150), ContinueStatement (110), DoStatement (110), EmptyStatement (103), EqualityExpression (96), EscapeSequence (80), ExclusiveOrExpression (98), ExplicitConstructorInvocation (166), Expression (90), ExpressionName (82), ExpressionStatement (103), FieldAccess (156), FieldDeclaration (147), FieldModifier (147), FloatingPointLiteral (79), FloatingPointType (86), ForStatement (109), FormalParameter (118), Identifier (81), IfStatement (106), ImportDeclaration (244), InclusiveOrExpression (98), InstanceInitializer (152), IntegerLiteral (78), IntegralType (85), InterfaceDeclaration (187), InterfaceMemberDeclaration (188), InterfaceModifier (188), InterfaceType (187), IterationStatement (108), Keyword (81), LabeledStatement (111), Literal (77), LocalVariableDeclaration (121), MethodDeclaration (117), MethodInvocation (118), MethodModifier (117), MethodName (82), MultiplicativeExpression (94), NullLiteral (145), NumericType (84), PackageDeclaration (240), PackageName (241), PackageOrTypeName (82), Primary (91), PrimitiveType (84), PostFixExpression (92), ReferenceType (143), RelationalExpression (96), ReturnStatement (118), ShiftExpression (96), Statement (103), StatementExpression (91), StaticInitializer (152), StringLiteral (81), SwitchStatement (107), SynchronizedStatement (216), ThrowStatement (279), TryStatement (283), Type (76), TypeName (82), UnaryExpression (93), UnaryExpressionNotPlusMinus (93), UnicodeEscape (80), VariableDeclarator (121), WhileStatement (108).

Little Classification of some Notions

Use the index to find the places in the text.

Hardware-related: CPU, processor, RAM (Random Access Memory), interrupt, port, mouse, keyboard

Operating-System related: socket, exception, file-system, Hierarchical File-System (HFS), terminal-window

Operating-System related - Multithreading: scheduling, round-robin, priority, semaphore, task, process, thread

Network related: socket, protocol, packet, frame, address, IP-Address, connection-oriented protocols, packet-oriented protocols, transport-protocols host-name, domain-name

Software-development related: syntax-diagrams, Backus-Naur-Notation, parser, compiler, interpreter, class-loader, (Integrated) Development Environment (IDE), debugger

Software entities related: standalone code, handler, plug-in, middleware, brokers, client, server, browser, editor, Application Programming Interface (API)

Language related: null-keyword, factory-method, factory-like-method, wrapper, exception

Sources

Sources that necessarily supplement this text have been mentioned especially beginning with page 59. Up-to-date information on definitions of vocabulary can be seen at the site <http://webopedia.internet.com> or in the glossary-gray-boxes throughout the text. A selection of texts of norms like the ISO-norms is given in the following list.

Sources On Java

Archive and Newsletter service from *Sun Microsystems Inc.*: Java Developer Connection (JDC) Tech Tips

- <http://java.sun.com>
The central source on Java: Java Language Specification 2nd Edition, TechTips, Short Courses, Tutorials, Java Development Kits, a glossary, Java User Group information and much more can be downloaded from there. The TechTips that are part of the newsletter-service furnish short concise practical information about various basic and advanced issues of Java programming.
- <http://www.gamelan.com>
A database of existent Java programs.
- comp.lang.java.*
Newsgroups on Java-related issues.
- http://dir.yahoo.com/Computers_and_Internet/Programming_Languages/Java
- CD-ROM's bundled with magazines or operating-systems often contain software and documentation.
- <http://www.kaffe.org>
An effort to devise an open-source Java Runtime System (JRS).

Sources on the Internet

The organization called Internet Activity Board (IAB) = "Internet Engineering Taskforce" (IETF) has a subdivision, the "Internet Engineering Steering Group" (IESG) that approves the publishing of documents which are the initial source of information for any Internet development. These documents are upgraded from "draft" to "Request for Comment" (RFC) and finally to "Internet standard" (STD). Many techniques and protocols that are in common use in the Internet still have the RFC status. View the documents at <http://ds.internic.net> or <http://www.ietf.org>

Secondary information can be found as usual at http://www.yahoo.com/Computers_and_Internet/Standards.

For some basic information about the Java language consult the Java Language Specification 2nd Edition. And especially refer to the Java Development Kit Documentation.

The Unicode-encoding of characters is documented extensively at <http://www.unicode.org>.

Sources on Programming

As one of the first addresses in that respect can be taken the java Tutorials from the *Sun* site (<http://java.sun.com>).

<http://www.wdvl.com> The Web Developert's Virtual Library

<http://webopedia.internet.com> This dictionary contains the up-to-date information about many more notions related to computing.

<http://www.oreilly.com> FTP site examplets.

<http://www.dejay.org> A Java variant that extends Java programming to multiple Virtual Machines.

Literature

Ladd, Scott Robert; **Java Algorithms**; McGraw Hill

[OOT] Taylor, David A.; **Object-Oriented Technologies**; Addison-Wesley

Boger, Marko; **Java in verteilten Systemen**; 1999; dpunkt.verlag (in German)

[JSDK] Java Standard Edition Documentation (online)

[JPAPIS] Java Platform API Specification found for example in [JSDK]

[JDA] Chan, Patrik; **The Java Developers Almanac**; 2000; Addison-Wesley

[JLS2] Gosling, James et. al.; **The Java Langage Specification**; 2000; Second Edition; (online)

[JT] Oaks, Scott and Wong, Henry; **Java Threads**; O'Reilly & Associates

László Böszörményi, Jürg Gutknecht, Gustav Pomberger, **The School of Niklaus Wirth**; 2000; dpunkt.verlag

[Oberon] Niklaus Wirth, Jürg Gutknecht; **Project Oberon**; 1992; ACM Press and Addison-Wesley, ISBN 0-201-54428-8

Index

- * in policy-file, 472
- in policy-file, 472
- 1, default TCP-port, 388, 627
- 1, end of file, 441, 627
- 1, end of stream, 296, 627
- .java.policy (text-file), 458, 461, 462, 467, 469
- 100 Percent Java, 72

- Absolute Path-Name, 48
- absolute path-name (example), 510
- absolute positioning and layout-manager, 542
- Abstract Class, 184
- abstract class (non-instantiable class), 184
- abstract class (non-instantiable class, concept), 134, 138
- abstract class and static variables, 184
- abstract class-type, type-casting identifiers to-, 552
- Abstract Method, 184
- abstract method (non-implemented method), 184
- abstract method (non-implemented method, concept), 134, 136, 138
- abstract modifier, 181
- Abstract Window Toolkit (AWT), applets, 613
- Abstract Window Toolkit (AWT), as API, 52, 67, 274
- Abstract Window Toolkit (AWT), as package, 53, 184, 246, 251, 266, 505
- Abstract Window Toolkit (AWT), class-hierarchies, 564
- Abstract Window Toolkit (AWT), description, 520
- Abstract Window Toolkit (AWT), event-handling, 276, 331, 440, 572
- Abstract Window Toolkit (AWT), image drawing, 519, 595
- Abstract Window Toolkit (AWT), image processing, 520
- Abstract Window Toolkit (AWT), image processing (extensive), 527
- Abstract Window Toolkit (AWT), peer-classes, 353
- Abstract Window Toolkit (AWT), screen-components, 538, 547, 551
- Abstract Window Toolkit (AWT), time-delayed data, 606
- Abstract Window Toolkit (AWT), versus Swing, 520, 548
- AbstractButton (javax.swing), 574
- AbstractList (java.util), 237
- Access (security-related), 449
- access, random-, 233, 291
- access-area, 430
- access-area and scope in an Hierarchical File System (HFS), 172
- Access-Area of an identifier, 172
- access-area, extended by overriding, 175, 286
- access-control, in security context, 449
- access-control, locking data in threads, 213
- access-control, with modifiers, 171, 175
- access-modifier, 177, 179, 180, 429
- access-modifier (protocol character), 345
- AccessControlContext (java.security), 470
- AccessController (java.security), 427, 453, 458, 462, 463, 465, 470, 476
- accessing class-members, 155
- accessing methods by overriding, 192, 286
- accessor (of a software-component), 312
- accessor-method (of a software-component), 312
- ActionEvent (java.awt.event), 544, 573, 574, 578, 603
- ActionListener (java.awt.event), 578
- actionPerformed()-method (java.awt.event.ActionListener), 127, 544, 576, 592
- activated window, 577
- activated window, concept of-, 509
- active graphics-card (hardware), 198
- active modem (hardware), 198
- adapter, 272, 289
- adapter-class, 187, 190, 580, 584, 601
- adapter-class with software-components, 318
- additive-operators, 94
- address of a computer in a network, 342
- address-space and port, 231, 271, 337
- address-space of a processor, 28, 291, 522, 535
- address-space, in threading, 217
- addressing resources network-wide: URL, 373
- administrator, 458, 475
- administrator (key-management and signing files), 488
- administrator (using a security-manager), 465
- administrator and jar-archive, 326
- administrator and security, 448, 453, 457
- administrator and tools, 54
- administrator, cannot change a security-manager, 463
- administrator, selecting policies, 460
- aggregates to streams, 237
- algebra, boolean-, 97
- algorithm, 20, 41, 140
- algorithm, compression-, 534
- algorithm, cryptographic-, 478
- algorithm, cryptographic- or encryption-, 480
- algorithm, ROT-encryption-, 478
- algorithm, Run Length Encoding (RLE)-compression-, 522
- algorithm, sorting-, 195, 249
- algorithm, strength or quality of a cryptographic-, 479
- algorithm, ZIP-compression-, 326, 522
- AlgorithmParameterGenerator (java.security), 485

- AlgorithmParameters (java.security), 485
- algorithms, implementation of cryptographic-, 485
- algorithms, modes of encryption-, 484
- algorithms, names of encryption-, 479
- algorithms, names of key-pair-, 481
- algorithms, names of- for digital signatures, 483
- algorithms, names of- for hashfunctions, 482
- AllPermission (java.security), 459, 461
- alpha-value/-channel, 526, 527, 556
- ambiguity and multiple inheritance, 136
- ambiguous identifier, 187
- ambiguous variable-name, 187, 239
- AND-operator (&&), 98
- AND-operators (&, &&), 84
- anonymous class, 159
- anonymous constructor-call, 159
- anonymous object, 159, 279, 581
- anonymous package, 241
- anonymous screen-window objects, 159
- API (Application Programming Interface), 51, 55, 129, 131, 190, 249
- API (Application Programming Interface) as Java software library, 54, 59
- API (Application Programming Interface, definition), 43
- append()-method (java.lang.StringBuffer), 120, 257
- applet, 67, 205, 396, 430, 613
- Applet (java.applet), 68, 396, 397
- Applet (java.applet) heavyweight, 565
- applet and application, difference of-, 460, 613
- applet and distributed computing, 338, 406, 423
- applet communication, 403
- applet HTML-tag, 68, 397, 399, 401, 431, 614
- applet introspection, 402
- applet methods, 529, 615
- applet methods, triggerable-, 397, 400, 614, 617
- applet runtime-environment, 67, 398, 564, 618
- applet runtime-environment as security-sandbox, 463
- applet runtime-environment, example of a-, 619
- applet runtime-environment, start, 401, 432, 614
- applet security, 460, 475
- applet source-code, 399, 615
- applet threading, applicational-, 401
- applet threads, supporting-, 205, 232
- applet with a main()-method, 619
- applet's triggerable methods, 618
- applet's uniqueness, 67, 333, 460, 613
- applet's window, 400
- applet's window (security mark), 475
- applet, concept of an, 67, 397
- Applet, description, 68
- applet-container (= applet-sandbox), 398
- AppletContext (java.applet), 404, 617
- appletviewer, 67, 397, 431, 475
- appletviewer as a Java program, 619
- appletviewer, controlling security externally with-, 457, 461, 463
- Application, 38, 507, 512, 613
- application and applet, difference of-, 460, 613
- application and main()-method, 38, 65, 127, 141, 142, 154
- application icon, 512
- Application Layer (OSI), 355
- Application Programming Interface (API), 51, 55, 129, 131, 190, 249
- Application Programming Interface (API) as Java software library, 54, 59
- Application Programming Interface (API, definition), 43
- application protocol and TCPport-number, 363
- Application User Interface, 512
- application with Graphical User Interface, 515
- application, HelloWorld-, 71
- application, inside named packages, 241, 246
- application, inter-application-communication, 353
- application, MIME-type, 369
- application, security control by programmer, 462
- application, security control by user, 457
- application, starting an-, 65, 431
- application, stopping an-, 432
- Application, Web-Application, 406
- application-development, 313
- application-programmer, 27, 448
- Application-Programming, 38
- application-protocol, 348, 367, 378
- application-protocol and servlet, 405
- application-protocol and URL, 373
- application-protocol and URL-class, 388
- application-specific programming interface of Java-beans, 311
- application-triggered-programming, 599
- applications with system tasks, 423
- applications, multiple in one JRE, 443, 630
- applications, security-related-, 488
- Archiving-Tool: jar, 325
- archiving-tool: jar, 54, 242, 243, 250, 489, 492
- ARGB-color-coding applied, 530
- ARGB-pixel-format, 526
- arguments of a method-call, 116
- ArithmeticException (java.lang), 92, 280, 281
- array, 234
- Array (java.lang.reflect), 250
- array, dimension of an -, 234
- array-initializer, 235
- array-type, 143, 234
- array-type and class-loader, 236, 443
- array-variable, 234
- arraycopy()-method (java.lang.System), 236, 254, 630
- ArrayIndexOutOfBoundsException (java.lang), 235, 281, 451
- Arrays (java.util), 236, 238, 249
- ASCII (American Standard Code for Information Interchange), 20, 33, 34, 73, 265
- ASCII-format and HTTPProtocol, 369
- ASCII-format and policy-file, 451, 458
- ASCII-format and security-files, 457
- ASCII-format and security/policy-files, 455
- ASCII-format and URL encoding, 375
- assignment, 74

- assignment by-reference, 143
- assignment by-value, 83, 143
- assignment operators, 99
- assignment-operator (=), 99
- assignment-operator (primitive- and reference-type), 144
- associative, left-/right-, 101
- asymmetric-key-cryptography, 479
- asynchronous method, 231
- atomic block of code, 214, 216
- Atomic Operation, 214
- AUI (Application User Interface), 512
- Authentication, 477
- authentication, 477, 488
- AWT (Abstract Window Toolkit) versus Swing, 520
- AWT (Abstract Window Toolkit), applets, 613
- AWT (Abstract Window Toolkit), as API, 52, 67, 274
- AWT (Abstract Window Toolkit), as package, 53, 184, 246, 251, 266, 505
- AWT (Abstract Window Toolkit), class-hierarchies, 564
- AWT (Abstract Window Toolkit), description, 520
- AWT (Abstract Window Toolkit), event-handling, 276, 331, 440, 572
- AWT (Abstract Window Toolkit), image drawing, 519, 595
- AWT (Abstract Window Toolkit), image processing, 520
- AWT (Abstract Window Toolkit), image processing (extensive), 527
- AWT (Abstract Window Toolkit), peer-classes, 353
- AWT (Abstract Window Toolkit), screen-components, 538, 547, 551
- AWT (Abstract Window Toolkit), time-delayed data, 606
- AWT (Abstract Window Toolkit), versus Swing, 548
- AWTEvent (java.awt), 331, 555, 573
- AWTEvent, class-hierarchy, extensions, 575
- AWTEvent, event-dispatch-thread, 586, 587
- AWTEvent-objects, triggering dispatch, 591
- AWTEventListener (java.awt.event), 587
- Backus-Naur Form (BNF), 76, 655
- Base-Class, 55
- base-classes, 54, 247, 249, 389
- base-classes (loading), 434
- base-classes, peeking inside-, 243, 244
- BasicPermission (java.security), 467
- BasicPermission (java.security) in footnote, 467
- Bean, 251, 313
- Bean Class, 313
- BeanInfo (java.beans), 318
- Beans, 311
- behavior and method (in object-orientation), 133
- behavior of an object, 131, 161
- big-endian order of bytes, 533
- binary number-system, 34
- binary operators, 93
- bit, 33, 78, 79, 87
- bit-length of words in the Java Virtual Machine, 216
- Bit-Mask, 533
- bit-mask, 287, 555, 563, 576, 579, 582, 591, 593
- bit-redundancy checks, 341
- Bitmap of image, 521
- bitmap of image, 96, 519, 525, 530, 535
- bitmap of image and screen-component, 540
- bitmap of image in rendering, 535
- bitmap of image, updated loading, 563
- bits for monitor screen-image, 537
- bits of IP-addresses, 359
- bits of IP-addresses (in footnote), 377
- bitwise operators, 95, 97
- bitwise shift-operators, 95
- bitwise-negation-operator (), 97
- blank-final, 122, 182, 186, 190
- block, 105
- block, (machine-)synchronized-, 216
- block, atomic-, 214, 216
- block, empty-, 184, 187, 190, 581
- block-cipher, 484
- Blocked thread, 199, 211
- Blocking Method, 129
- blocking method, 231
- BNF (Backus-Naur form), 76, 655
- Boolean (java.lang), 169, 250, 256
- boolean (primitive-type), 84, 93, 96, 97, 106, 108, 169, 253, 312
- boolean algebra, 97
- boolean literal, 79
- boolean negation-operator (!), 93, 97, 98
- BorderLayout (java.awt), 543, 558
- bound property (of software-component), 315
- branching, 89, 283
- branching (if-statement), 105
- branching (switch-statement), 106
- branching (switch-statement), example, 401
- branching (try-catch clause), 112
- branching in Java, 105
- break (Java-keyword), 106
- break-keyword, 110
- break-statement, 110
- break-statement and try-statement, 282
- breaking a cryptographic algorithm, 478
- bridge, 29, 353, 354
- broadcasting, 277, 343
- Browser, 371
- browser, 38, 197, 205, 243, 274, 324, 370, 373, 430, 520
- browser and applets, 67, 68, 72, 396, 397, 431, 564, 614
- browser and servlets, 406
- BufferedImage (java.awt.image), 520, 530
- BufferedInputStream (java.io), 304, 441
- BufferedOutputStream (java.io), 304
- BufferedReader (java.io), 300, 304
- BufferedWriter (java.io), 304, 386
- builder-tool (for software-components), 311
- bus (communication-line), 29, 87, 337
- bus (network topology), 340

- Button (java.awt), 566
- by-reference, assignment-, 143
- by-reference, passing arguments-, 120, 256
- by-value, assignment-, 83, 143
- by-value, passing arguments-, 120, 225, 256
- byte, 33, 73, 74, 78
- Byte (java.lang), 169, 181, 250, 256
- byte (primitive-type), 73, 85, 169
- byte (primitive-type), behavior in a cast, 87, 100, 101
- byte (primitive-type), byte-code as array of-, 441
- byte (primitive-type), key-data as array of-, 493
- byte (primitive-type), length of an array, 234
- byte order, big-endian- or network-, 533
- byte, 16 byte IP-address, 358
- byte, 8 to 16 bytes as hashCode, 481
- byte, four byte int-value, 388
- byte, four byte int-value as ARGB-format, 526
- byte, four byte IP-address, 357
- byte, most significant-, least significant- of a given byte order, 533
- byte, one byte ASCII-format, 33, 265, 374
- byte, one byte unit of stream-data, 296
- byte, represented by a pair of hexadecimal digits, 36
- byte, two byte port-address, 231, 363
- byte, two byte Unicode-format, 34
- byte, two byte unit of stream-data (=char), 296, 302
- Byte-Code, 65
- byte-code-verifier, 281, 441, 450, 451
- ByteArrayInputStream (java.io), 238
- ByteArrayOutputStream (java.io), 238
- CA (certification authority), 484
- Calendar (java.util), 262
- callback, 228, 288, 631
- callback of methods, 127
- callback-model, 276, 586
- Canvas (java.awt), 536, 538, 597
- CardLayout (java.awt), 543, 559
- case (Java-keyword), 106
- case of letters, 34, 66, 81, 82, 491
- case of letters (in domain-names), 361
- case of letters (in URIs), 375, 394
- case of letters, in integer-literal of type long, 78
- case-sensitive identifiers, 81
- case-sensitive, not-(applet's parameter-name), 616
- cast potential, testing type-, 253
- cast, implicit type- (example), 544, 580
- cast, narrowing type-, 169
- cast, widening type-, 169
- casting identifiers to an abstract class-type, 552
- casting identifiers to interface-type, 194, 404
- casting of variable of primitive-type, 88
- casting of variable of reference-type, 169
- casting, implicit type-, 86, 87, 100, 106, 169, 194, 253
- Casting, Type-, 88
- casting, type-, 257, 450
- CD-ROM, 298
- certificate, 489
- Certificate (java.security.cert), 494
- Certificate interface (java.security) in footnote, 494
- Certificate Revocation List (CRL), 484
- Certificate Signing Request (CSR), 484
- certificate, a signed public key, 484
- certificate, self-signed, 484
- CertificateFactory (java.security.cert), 485
- certificates in key-store, 488
- certification authority (CA), 484
- chain-model, 276, 440, 574
- ChangeEvent (javax.swing.event), 547, 574, 577
- ChangeListener (javax.swing.event), 547, 577
- channel, virtual- (in network), 346
- channel-switching, 277, 345
- char (primitive-type), length of an array, 234
- char (primitive-type), 85, 169
- char (primitive-type), as TCPport, 385
- char (primitive-type), as UDPport, 384
- char (primitive-type), character-encoding, 86, 584
- char (primitive-type), stream-data, 296, 302, 309
- Character (java.lang), 82, 169, 257
- character-literal, 73, 79
- CharArrayReader (java.io), 238
- CharArrayWriter (java.io), 238
- charAt()-method (java.lang.String, StringBuffer), 257
- check...()-method (java.lang.SecurityManager), 128, 256, 456, 462, 464, 465
- check...()-methods, triggerable-, 462
- checkbox, 510, 521, 548
- Checked Exception, 281
- checked exception, 280, 330
- checked exceptions and throws-keyword, 281
- checkPermission()-method (java.security.AccessController), 128, 453, 456, 465, 467
- checksum, 481
- child-component, 542, 543, 551
- child-component, layout, 558
- chime-like counting (NON-), 458
- chime-like counting in random-access-filing, 299
- chime-like counting of character-positions in strings, 257
- chime-like counting of coordinates, 537
- chime-like counting, (0,1,2,...) in footnote, 234
- Cipher, 478
- Cipher (javax.crypto), 497
- cipher, block-, 484
- ciphers, names of-, 496
- class, 138
- Class (java.lang), 151, 250, 252, 255, 256, 435, 437, 629
- Class (java.lang) acquired for reflection, 252, 318
- Class (java.lang) as class-loader, 443
- Class (java.lang) its origin, 441
- Class (java.lang) references ProtectionDomain, 468
- Class (java.lang) references super-class, 558
- Class (java.lang) references the class-loader, 435
- class as program/application, 141
- class as type, 73, 140
- class declaration/-definition, 145

- class final and constructors non-public: no instance, 255, 257, 382, 417
- class locked by one of its static synchronized methods, 214
- class without a super-class (java.lang.Object), 164
- class, abstract- (non-instantiable class), 184
- class, abstract- (non-instantiable class, concept), 134, 138
- class, abstract-, and static variables, 184
- class, adapter-, 187, 190, 580, 584, 601
- class, adapter-, with software-components, 318
- class, anonymous-, 159
- class, constant-, 183
- class, final- (non-extensible), 146, 183
- class, final- (non-extensible) example, 257
- class, inner-, 157
- class, interface- (RMI), 416
- class, loading of a-, 152, 433
- class, local-, 157
- class, member-, 157
- class, nested, 157
- class, public-, extended over different packages, 438
- class, public-, has to be in file of same name, 146, 241, 242, 245, 246
- class, runtime-representation of a-, 54, 255, 433, 441, 443, 629
- class, signed- = signed class-file, 494
- class, skeleton- (RMI), 416
- class, static member-, 158, 182
- class, stub- (RMI), 416
- class, top-level-, 157
- class, virtual- (non-instantiable class), 138
- class, wrapper-, 169, 171
- class-extension, 138
- class-hierarchy, 133, 135, 138, 164, 247
- class-hierarchy (AWT/Swing), 564
- class-hierarchy and casting, 171
- class-hierarchy and multiple extension, 186
- class-literal, 252, 437
- class-loader (secure), 469
- class-loader delegation, 440
- class-loader's name-space, 436, 469
- class-loader, secure-, 454, 458, 471
- Class-Loading, 433
- Class-Member, 148, 166
- class-member, 147, 148, 154
- class-member and static-keyword, 154
- class-member, static-, 140, 148, 154
- class-members and interfaces, 190
- class-members, accessing-, 155, 156, 171
- class-method, 138, 154, 182
- Class-Name, 146
- Class-Structure, 135, 164
- class-structure, 134, 135, 164, 653
- class-structure (example), 399
- class-structure and class-hierarchy, 135, 166, 194
- class-structure and interface, 190
- class-structure and java.lang.reflect-package, 250
- class-structure and programmer, 164
- class-structure as index, 142, 194
- class-structure of JApplet, 615
- class-type, 120, 143, 146
- class-variable (static variable), 138, 148, 154, 182
- class/object-model, 131
- class/object-model (in programming language), 140
- ClassCastException (java.lang), 171, 253, 281
- classes, list of mentioned-, 653
- classes.zip, 243, 244, 247, 249, 389, 434, 641
- ClassLoader (java.lang), 250, 255, 435, 436
- ClassLoader (java.lang) extensive, 440
- ClassLoader (java.lang) referenced, 629
- ClassLoader (java.lang) simple, 438
- ClassNotFoundException (java.lang), 281, 436
- CLASSPATH-variable (operating-system), 61, 66, 243, 254, 436, 451, 629
- Client, 276
- client-side computing, 406
- Clonable interface (java.lang), 189
- Clone, 191
- clone, 192
- clone()-method (java.lang.Object), 144, 165, 252
- clone()-method, a factory-method, 151
- clone()-method, needs Cloneable-interface, 192, 258, 637
- clone()-method, unoverridden, copies shallow, 193
- clone()-method, usually overridden, 168
- Cloneable interface (java.lang), 192, 252, 281
- CloneNotSupportedException (java.lang), 192, 281
- clones are shallow equal, 162, 191, 193
- cloning a variable of primitive-type, 83
- cloning a variable of reference-type, 144
- Cloning, Deep-, 191
- Cloning, Shallow-, 191
- close()-method (java.io.FilterInputStream, java.io.FilterReader), 302
- close()-method (java.io.InputStream, java.io.Reader), 296, 305, 456, 493
- close()-method (java.io.OutputStream, java.io.Writer), 296, 456, 493
- close()-method (java.io.RandomAccessFile), 299
- close()-method (java.net.Socket), 383, 385
- close()-method (javax.servlet.ServletOutputStream), 409
- CMY-model, 524
- CMYK-model, 525
- code, machine-, 37
- code, object-, 38
- code, P-, 39
- code, Pseudo-, 39
- code, Source-, 38
- code, source-, 37
- code, standalone- (=program), 160
- codeBase in a policy-text-file, 459–461, 472
- codebase of applet's byte-code, 403, 614
- CodeSource (java.security), 454, 455, 465, 501
- coding- and content-transformation, 35, 520
- coding-transformation, 534, 535
- Collection (java.util), 237
- Color (java.awt), 550, 556
- Color and transparency, 527
- color concept, 522

- color-cube, 523
- color-model, 522
- color-model, CMY-, 524
- color-model, CMYK-, 525
- color-model, direct-, 526
- color-model, HSV and HSB, 525
- color-model, HSV-, ..., 525
- color-model, index-, 526
- color-model, indirect-, 526
- color-model, RGB-, 523
- color-space, 523
- ColorModel (java.awt.image), 520, 556, 596
- Command-Line, 44
- command-line, 44, 250, 324, 400, 407, 430, 431, 496
- command-line of browser, 68, 373, 614
- comment in source-code, 37, 71, 323
- communication infrastructure, three-tiered-, 277, 352, 406, 414
- communication suggesting concurrency, 231
- Comparable interface (java.lang), 189
- compareTo()-method (java.lang.Comparable) abstract, 189
- comparison operators (==, !=), 96
- Compilation Unit, 246
- compileClass()-method (java.lang.Compiler), 255
- Compiler, 38, 39
- compiler (byte-code to native code), 255
- Compiler (java.lang), 250, 255
- compiler, cross-, 45
- compiler, Java-, 52, 53, 124, 184, 257, 280, 431
- compiler, Java-, and -generated names, 83
- compiler, Java-, and character-encoding, 71, 79
- compiler, Java-, and comparison-operator, 163
- compiler, Java-, and compilation unit, 246
- compiler, Java-, and nested classes, 159
- compiler, Java-, and security, 450
- compiler, Java-, and unreachable statements, 106
- compiler, Java-, import-on-demand, 244
- compiler, Java-, in JDK, 54
- compiler, Java-, late-binding, 168, 183
- compiler, Java-, letter-case sensitive, 66
- compiler, Java-, searching files, 63, 65
- compiler, rmic- (RMI), 415
- Compiler-tool: javac, 72, 431
- compiler-tool: javac, 60, 246, 333, 397, 407, 473
- Component (java.awt), 538, 551, 559, 563
- Component as drawing surface, 528
- Component as source of event-objects, 577, 591
- Component method isLightweight(), 550
- Component methods, 553
- Component's paint()-method, 595
- Component, as source of event-objects, 331, 573
- Component, class hierarchy, 552, 565
- component, screen-, 331, 399, 509, 519
- component, screen- as a painting-surface, 538
- component-model, 138, 311, 381, 542
- ComponentEvent (java.awt.event), 573, 575, 577, 598
- ComponentListener (java.awt.event), 577
- compression, 240, 325, 368
- Compression, destructive-, 522
- Compression, non-destructive-, 522
- compression, RLE-, 522
- compression, ZIP-, 326, 522
- compression-algorithm, 534
- Computer, 17
- computer-model, 19, 23
- computer-platform, 44
- computer-system, 29
- computing, distributed-, 423
- concatenation, string-, 95
- conditional compilation, 106
- conditional-operator (?), 99
- connection, logical-, 343
- connection, multipoint-, 343
- connection, point-to-point-, 277, 343
- connection-oriented network, 345
- Connection-Oriented Protocol, 350
- connection-oriented protocol, 275, 351
- connectionless network, 345
- connectionless-protocol, 350
- constant (final variable in interfaces), 180, 187
- constant (final variable), 83, 85, 86, 106, 122, 182, 256, 450, 497, 563, 577, 582, 584
- constant class, 183
- constant object, 183
- constant, String-, 383
- constrained property (of software-component), 315
- constructor, 145, 165, 642
- Constructor (java.lang.reflect), 250, 253
- constructor (object-orientation), 134, 138
- constructor of core-class and security behavior, 463
- constructor without arguments, 642
- constructor, declaration, 149
- constructor, default-, 149, 166
- constructor, default-accessible-, 181
- constructor, not a class-member, 148, 166
- constructor, overloaded-, 151, 174
- constructor, private-, 150, 181, 182, 382, 417
- constructor, private-, and singleton, 160
- constructor, protected-, 151, 181
- constructor, public-, 180
- constructor, super()- (class-extension), 166
- constructor, this()-, 167
- constructor-call, 145, 151
- constructor-call chaining (class-extension), 166
- constructor-call sequence, 152
- constructor-call, anonymous-, 159
- constructor-callback, 631
- constructors and factory-methods, 151
- constructors non-public and class final: no instance, 255, 257, 382, 417
- Container (java.awt), 551, 556, 559
- container and multiple layouts, 559, 566
- Container as content-pane, 536, 538
- Container as content-pane (Swing), 567
- Container as drawing surface, 528
- Container as layered-pane (Swing), 567
- Container as source of event-objects, 578

container of screen-components, 512, 542, 551, 556, 576, 598
 Container's paint()-method, 595
 container, applet- (=applet-sandbox), 398
 Container, class hierarchy, 556, 564, 565
 container, fifo-, 293
 container, servlet- (=servlet-sandbox), 406, 407, 409, 430
 container-file (=jar-file), 491
 container-object, 138
 ContainerEvent (java.awt.event), 576, 578, 598
 ContainerListener (java.awt.event), 578
 containment-hierarchy (screen-components), 440, 541, 543
 content- and coding-transformation, 35, 520
 Content-Handler, 369
 content-stream, 389, 394
 content-type, 368, 405
 content-type and encoding, 368
 content-type and portability, 265
 ContentHandler (java.net), 394
 ContentHandlerFactory (java.net), 394
 context-free-/generic event-object, 572, 573
 context-independent-event, 575
 contextual event, 575
 continue-keyword, 110, 112
 continue-statement, 110
 Control Structure, Statement and Expression (concept), 89
 Controller and MVC, 515
 conventions, 83, 140
 Cookies, 351, 369, 413
 coordinate-system for colors, 525
 coordinate-system for graphics-objects, 519, 534, 536
 coordinate-system of devices (pixel-oriented), 536
 coordinates, generic-, 536
 coordinates, pixel-, 536, 542
 copy — deep copy of an object-graph, 162
 copying a variable of primitive-type, 83
 copying a variable of reference-type, 144
 Core-Class, 55
 core-classes, 54, 247, 249, 389
 core-classes (loading), 434
 core-classes, peeking inside-, 243, 244
 country-code top-level-domains (DNS), 361
 CPU (Central Processing Unit), 28
 cross-compiler, 45
 cryptographic algorithm, 478, 480
 cryptographic algorithm, strength or quality of an-, 479
 cryptographic algorithms, implementations of-, 485
 cryptographic-engine, 486
 cryptography, 477
 cryptography with key-pairs or asymmetric keys, 479
 cryptography with secret- or symmetric keys, 478
 Current Directory, 48
 currentThread()-method (java.lang.Thread), 207, 212, 232, 401

currentTimeMillis()-method (java.lang.System), 254, 261
 cursor (GUI-pointer), 509, 536, 542, 556, 605

 Daemon-thread, 223
 daemon-thread, 215, 432, 586
 Data Link Layer (OSI), 354
 Database, 321
 database, 20, 40, 138, 193, 235, 246
 database and resource-bundle, 263
 database, distributed, 360
 database, file-system as, 46
 database, key-store-, 489
 database, non-persistent-, 233
 Datagram, 351
 DatagramPacket (java.net), 384
 DatagramSocket (java.net), 384, 421
 DataInput (java.io) interface, 300, 307
 DataInputStream (java.io), 382
 DataOutput (java.io) interface, 300, 307
 DataOutputStream (java.io), 382
 Date (java.util), 261
 deadlock (thread-specific problem), 226
 decimal number-system, 34, 79, 257
 declaration, 74, 84
 declaration-assignment-sequence, 122, 155
 decrement operators (pre-, post-), 92
 Deep Cloning, 191
 deep copy of an object-graph, 162
 deep equal object-graphs, 162, 192
 default access, 177
 default assignment, 86, 87
 default assignment of member-variables: YES, 148
 default assignment with local variables: NONE, 121
 default constructor, 149, 166
 default type of number literals: int (primitive-type), 86
 default-accessible constructor, 181
 default-keyword (switch-statement), 107
 default-package, 180
 default-package (idea), 241
 default-package, in footnote, 193
 Delegation, 440
 delegation event-model, 19, 276, 314, 440, 571, 573, 576, 586
 delegation event-model (Concept in Java), 278
 delegation, class-loader-, 440
 design-pattern, 251, 311, 312
 design-pattern (event-framework), 287
 desktop, 44, 509
 desktop in Swing, 610
 destroy()-method (java.applet.Applet), 205, 398, 400, 618, 641
 destroy()-method (java.lang.Process), 255
 destroy()-method, triggerable-, 398
 development-environment, 64, 65, 311, 313, 318, 450
 Dialog (java.awt), 565
 Dialog (java.awt) top-level screen-component, 564, 598

- dialog window, 540
- dialog, modal-, 564
- difference of applet and application, 460
- digest, message-, 481
- digital fingerprint, 481
- digital signature, 483
- digital signature and certification, 484
- digital signatures, names of algorithms for-, 483
- Dimension (java.awt), 555
- dimension of an array, 234
- dimension, length of a - (array), 234
- dining humans (thread-specific problem), 215
- direct color-model, 526
- DirectColorModel (java.awt.image), 526
- directory-/folder-/file-separator, 46, 254
- directory-/folder-/file-separator (in packaging), 242
- directory-/folder-/file-separator (policy-files), 458, 462
- directory-/folder-/file-separator (URL), 364, 388
- disableEvents()-method (java.awt.Component), 287
- disassembler-tool: javap, 250, 642
- Discrimination (security-related), 449
- dispatchEvent()-method (custom-defined), 287, 288, 331
- dispatchEvent()-method (java.awt.Component), 592
- dispatchEvent()-method (java.awt.Component, ...), 591
- disposing of a top-level screen-component, 598
- Distinguished Name (DN), 481, 488, 489
- Distributed Computing, 338
- distributed computing, 423
- distributed computing (example), 362
- distributed database, 360
- distributed garbage-collector, 251, 415
- distributed programming, 419, 423
- dithering, 535
- division by zero with integers: ArithmeticException, 86
- division-operator, 93
- DN (Distinguished Name), 481, 488, 489
- DNS (Domain Name Service), 348, 354, 359, 362, 381
- do-statement (usually a pre-condition loop), 109
- documentation-tool: javadoc, 324
- Domain Name Service (DNS), 348, 354, 359, 362, 381
- Domain-Name, 362
- domain-name (case of letters), 361
- dots per inch (dpi) resolution, 536, 537
- Double (java.lang), 86, 169, 181, 194, 250, 256
- Double (java.lang) constants (NaN), 256
- double (primitive-type) and JVM, 87, 216
- double (primitive-type), 169
- double (primitive-type) and equality operator, 97
- double (primitive-type) and strictfp modifier, 118, 146, 183
- double (primitive-type), argument in mathematical functions, 94
- double (primitive-type), behavior in a cast, 169
- double (primitive-type), default type of floating-point-literals, 79, 86
- double (primitive-type), return-value of mathematical functions, 106
- double (primitive-type), value of a StreamTokenizer, 301
- double buffering (= off-screen rendering), 535, 542, 621
- double-click (GUI concept), 510
- downloading a Java Runtime System (client FROM server), 59
- Downloading files (client FROM server), 371
- downloading files (client FROM server), 19, 372, 374
- downloading files (client FROM server) and distributed computing, 338
- downloading files (client FROM server) and monitoring, 606
- downloading files (client FROM server) and security, 57
- downloading mail (client FROM server), 371
- downloading package-files (client FROM server), 243
- downloading the Java Cryptography Extension (client FROM server), 496
- downloading the Java Servlet Web Development Kit (client FROM server), 409
- dpi (dots per inch) resolution, 536, 537
- drag and drop (GUI concept), 510
- dragging (GUI-concept), 510
- drain of message-model, 271
- drawing-method, 536
- drawing-surface, 528, 536
- drawing-surface, a JComponent, 528, 530, 531, 538, 602
- drawing-surface, a JPanel, 608
- drawing-surfaces, 538
- driver (system-software), 38, 43, 45, 273, 378, 508, 512, 516, 522, 572
- drivers and operating-system, 508
- duplicating objects, 191
- dynamically-typed language, 75
- Early-Bound Method, 168
- editor, hex-, 449, 489
- editor, hex-, see Java byte-code, 642
- editor, text-, for editing security- and policy-files, 453, 457, 458, 496
- editor, text-, for writing source-code, 53, 54, 57, 62, 63, 65, 323
- email-client, 371
- empty statement, 103
- empty-block, 184, 187, 190, 255, 581
- enableEvents()-method (java.awt.Component), 287, 579, 591, 592, 594
- encapsulation, 115, 122, 138, 139, 377
- encapsulation and non-local variables, 195
- Encoding, 35
- encoding, 40, 73
- encoding and content-type, 368
- encoding of image-information, 521
- encoding transformation with images, 535
- encoding, ASCII-, 34
- encoding, Unicode-, 34

encoding, URL-, 375
 encodings, 265
 Encryption, 477
 encryption algorithm, 478, 480
 encryption algorithms, implementations of-, 485
 encryption algorithms, modes of-, 484
 encryption algorithms, names of-, 479
 encryption-engine, 478
 endian-order of bytes, little-/big-, 533
 Engine, cryptographic-, 486
 engine, encryption-, 478
 environment, development-, 64, 65, 311, 313, 318, 450
 equal — deep equal object-graphs, 162, 192
 equal — shallow equal object-graphs, 162, 192
 equal sign (=), assignment-operator, 99
 equal sign (=), copying or referencing, 143
 equal sign (=), tool option, 461
 equality operator (==), 161–163
 equality operator (==) and reference-type, 162
 equality operator (==) with float and double, 97
 equals()-method (java.lang.Object), 165, 190, 252, 256
 equals()-method (java.lang.String), 162
 Error (java.lang), 281
 escape-sequence, 80
 Ethernet (network hardware), 20, 29, 30, 190, 339–341, 348, 354, 360
 Ethernet (network hardware) -address, 342
 Ethernet (network hardware) installation hint, 378
 event, context-independent-, 575
 event, contextual-, 575
 event-delegation model (Concept in Java), 278
 Event-Dispatch-thread, 586
 event-dispatch-thread, 205, 228
 event-dispatch-thread, and painting-methods, 595
 event-dispatch-thread, running code on the-, 590
 event-example (footnote), 275
 event-handler, 287, 575, 577, 586
 event-mask, 287, 555, 576, 579, 582, 591, 593
 event-messages (message-model), 272
 event-messaging, unicast-, 287, 289, 633
 event-model, 278, 329
 event-model, delegation-, 19, 276, 314, 440, 571, 573, 576, 586
 event-model, subscriber-, 276
 event-queue, 586
 event-scheduling, 572, 574
 event-type-hierarchy, 578
 EventListener interface (java.util), 189, 287, 288, 314, 331, 332, 573
 EventListenerList (javax.swing.event), 579
 EventObject (java.util), 250, 276, 287, 288, 331, 586, 633
 EventQueue (java.awt), 332, 571, 586, 587
 EventQueue reference, 587
 EventQueue-object, 595
 Exception, 113
 Exception (java.lang), 280, 281, 283, 330, 637
 Exception, Checked-, 281

exception, checked-, 280, 330
 exception-handler, 283
 exec()-method (java.lang.Process), 255
 exec()-method (java.lang.Runtime), 254
 exit()-method (java.lang.Runtime), 254
 exit()-method (java.lang.System), 254, 432, 581, 651
 exit()-method (java.lang.System), example, 226, 382
 exp()-method (java.lang.Math), 94
 expansion, property- with text-files, 458, 459
 explicit type-casting, 100
 exponential function (mathematical function), 94
 expression, 90
 Expression, Statement and Control Structure (concept), 89
 ext-folder, 61, 243, 244, 247, 250, 410, 429, 487, 496, 499, 641
 extended file-name, 47
 extending access-area by overriding, 175, 286
 extends-keyword, 146, 187
 extends-keyword (example), 163, 186
 eXtensible Markup Language (XML), 410, 414
 extension, class-, 138
 extension-hierarchy, 135

 Factory-Like-Method, 185, 551
 factory-like-method, 56, 266, 527, 551, 569, 596, 641, 642
 Factory-Method, 151
 factory-method, 145, 185, 273, 381, 392, 485, 629, 642
 factory-method (static), 151
 factory-method (static) in footnote, 382
 factory-methods and constructors, 151
 Field, 148
 Field (java.lang.reflect), 250, 253
 field, final- (quasi-constant), 148
 field, final-, example, 160
 field, transient- (non-serializable), 183, 309
 field, volatile- (uncached), 148, 183
 field-variable, 155
 fifo-container, 293
 fifo-container (stream), 310
 fifo-storage (first-in-first-out), 233
 File (java.io), 259, 463
 File Transfer Protocol (FTP), 349, 354, 371, 374, 389
 file, container- (=jar-file), 491
 file-/folder-/directory-separator, 46, 254
 file-/folder-/directory-separator (in packaging), 242
 file-/folder-/directory-separator (policy-files), 458, 462
 file-/folder-/directory-separator (URL), 364, 388
 file-hierarchy, 510
 file-name, extended-, 47
 File-System, 46
 file-system as database, 46
 file-system, flat-, 46
 FileInputStream (java.io), 297, 298
 FileInputStream (java.io) security, 463
 FileOutputStream (java.io), 297, 298

- FilePermission (java.io), 459–461, 463, 472
- FileReader (java.io), 298, 456, 463
- FileReader (java.io) security, 453
- FileWriter (java.io), 298, 456
- filter-stream, 292
- FilteredImageSource (java.awt.image), 610
- final class (non-extensible), 146, 183
- final class (non-extensible) example, 257
- final field (quasi-constant), 148
- final field, example, 160
- final local variable, 122
- final method called faster, 118, 168, 182
- final method, unoverridable, 168, 182
- final modifier, 117, 182
- final variable (quasi-constant), 182
- final, implicitly - variables in interfaces, 186
- finalize()-method (java.lang.Object), 127, 152, 165, 232, 252, 415
- finalize()-method and destroy()-method, 398
- finalize()-method with own thread, 228, 232
- finalize()-method, triggerable-, 127, 153
- finalize-thread, 228
- finalizer, not a class-member, 148, 166
- finally-keyword, 113, 282, 330
- fingerprint, digital-, 481
- fire...()-method (custom-defined), 288, 331
- fire...()-method (Java beans, Swing, ...), 287, 312, 314, 331, 574, 591
- fireActionPerformed()-method (javax.swing.AbstractButton), 574
- firePropertyChange()-method (java.awt.Component), 315, 319, 331, 578, 603
- fireStateChanged()-method (javax.swing.AbstractButton, ...), 574, 577
- fireVetoableChange()-method (javax.swing.JComponent, java.beans.VetoableChangeSupport), 315, 331
- flat file-system, 46
- Float (java.lang), 86, 169, 181, 250, 256
- Float (java.lang) constants (NaN), 256
- float (primitve-type), 86, 169
- float (primitve-type) and equality operator, 97
- float (primitve-type) and strictfp modifier, 118, 146, 183
- float (primitve-type) as color-value, 556
- float (primitve-type), behavior in a cast, 101, 169
- float (primitve-type), postfixing literals, 100
- floating-point-literal, 78
- FlowLayout (java.awt), 543, 558
- FlowLayout, default layout of many containers, 559
- focus, concept of-, 509
- focus, screen-component on-, 542, 552, 575, 577
- focus-cycle, 509
- FocusEvent (java.awt.event), 575, 577
- FocusListener (java.awt.event), 577
- folder-/file-/directory-separator, 46, 254
- folder-/file-/directory-separator (in packaging), 242
- folder-/file-/directory-separator (policy-files), 458, 462
- folder-/file-/directory-separator (URL), 364, 388
- font, 34, 265
- Font (java.awt), 569
- fonts, Java and, 266, 569
- for-statement (usually post-condition loops), 109
- Format, 35
- format, 521
- format, text-based, 414
- forName()-method (java.lang.Class), 185, 252, 318, 394, 435, 437, 443, 468, 619, 629, 642
- FP-strict, 118
- Frame (java.awt), 562
- Frame (java.awt) top-level screen-component, 564, 598
- frame (message-model), 272
- Frame (of packet-oriented protocol), 349
- frame or image (of a movie, animation-context), 602
- frame or packet (networking-context), 349
- frame or window (as modal screen-component), 540
- frame or window (GUI container), 512, 573
- frame or window (GUI-context), 509
- frame or window (size-change), 573
- frame- or window-states (concept), 509
- Framework, 237
- framework, 287
- freeMemory()-method (java.lang.Runtime), 254, 258, 445
- frequency, 31
- front-end, 507
- front-end and kernel, 515
- front-end, a Java desktop, 610
- front-end/kernel model, 515, 516, 542
- FTP, 364
- FTP (File Transfer Protocol), 349, 354, 371, 374, 389
- fully-qualified class-name, 244
- Function, Hash-, 482
- Functions, One-Way-, 480
- gamma-correction (screen-hardware related), 523
- garbage-collection (Concept), 444
- garbage-collection (distributed), 415
- garbage-collection-thread, 223, 228, 444
- garbage-collector, 153, 223, 254, 415, 450, 618
- garbage-collector (distributed), 251
- gateway, 353
- GB (GigaBytes) more than a billion bytes, 35
- gc()-method (java.lang.Runtime), 254
- gc()-method (java.lang.System), 153, 223, 254, 445, 598
- generic coordinates, 536
- generic top-level-domains (DNS), 361
- generic-/context-free event-object, 572, 573
- get...()-method, 312, 313, 318
- getByName()-method (java.net.InetAddress), 151, 382, 385
- getClass()-method (java.lang.Object), 245, 252, 318, 437, 630, 633, 638
- getClassLoader()-method (java.lang.Class), 435, 437, 439, 468, 629

- getContentPane()-method (javax.swing.JRootPane), 528, 536, 567
- getDefaultToolkit()-method (java.awt.Toolkit), 185, 528, 579, 596, 604, 608
- getDefaultToolkit()-method (java.awt.Toolkit, Component), 551
- getErrorStream()-method (java.lang.Process), 255
- getImage()-method (java.awt.Toolkit, java.applet.Applet, ...), 232, 528, 529, 596, 608
- getInputStream()-method (java.lang.Process), 255
- getInputStream()-method (java.net.Socket), 297, 302, 383, 385, 386, 395
- getInputStream()-method (java.net.URLConnection), 393, 395
- getInputStream()-method (javax.servlet.ServletRequest), 412
- getInstance()-method (custom-defined), 160
- getInstance()-method (java.security.KeyFactory), 494
- getInstance()-method (java.security.KeyPairGenerator), 493
- getInstance()-method (java.security.KeyStore), 495
- getInstance()-method (java.security.SecureRandom), 493
- getInstance()-method (java.security.Signature), 493
- getInstance()-method (java.util.Calendar), 262
- getInstance()-method (javax.crypto.Cipher), 497
- getInstance()-method (javax.crypto.KeyGenerator), 497
- getInstance()-method (some engine-class), 485, 487
- getMessage()-method (java.lang.Throwable), 280
- getMessage()-method (java.lang.Throwable, ...), 282
- getName()-method (java.io.File), 259
- getName()-method (java.lang.Class), 630
- getName()-method (java.lang.Thread), 211
- getOutputStream()-method (java.lang.Process), 255
- getOutputStream()-method (java.net.Socket), 297, 302, 382, 385, 386, 395
- getOutputStream()-method (java.net.URLConnection), 393
- getOutputStream()-method (javax.servlet. ServletResponse), 409, 412
- getPackage()-method (java.lang.Package), 245
- getPriority()-method (java.lang.Thread), 211, 224, 232
- getProperties()-method (java.awt.Image), 607
- getProperties()-method (java.lang.System), 254, 258, 556
- getProperty()-method (java.awt.Image), 608
- getProperty()-method (java.lang.System), 254, 259, 456
- getRuntime()-method (java.lang.Runtime), 254, 258, 445
- getSecurityManager()-method (java.lang.System), 254, 255
- getSource()-method (java.awt.Image), 607
- getSource()-method (java.util.EventObject), 288, 315, 545, 546, 611
- GIF (Graphics Interchange Format), 369, 522, 529
- GigaBytes (GB) more than a billion bytes, 35
- glyph of a font, 34, 82, 265
- graph of a network, 340
- graph of references, 161, 162, 191, 233, 307, 441, 444
- graph of references (cloning), 191
- graph, processing-, 520
- Graphical User Interface (GUI), 44, 190, 507, 508, 520
- Graphical User Interface and applets, 615
- Graphical User Interface and applications, 512
- Graphical User Interface and Java-Beans, 313
- Graphical User Interface and locale, 263
- Graphical User Interface and MVC, 515
- Graphical User Interface and operating-system, 508
- Graphical User Interface and policytool, 458
- Graphical User Interface, design of, 513
- graphical-object, 507, 519
- graphical-object, concept, 534
- Graphics (java.awt), 400
- Graphics (java.awt) as software-library, 505, 519, 534, 536, 540, 556
- Graphics (java.awt) typed reference to native-code, 185, 527, 556
- Graphics Interchange Format (GIF), 369, 522, 529
- graphics toolbox, 538
- graphics toolbox/-context/-library, 519, 536, 540, 599
- graphics-card (hardware), 522
- graphics-card, active- (hardware), 198
- graphics-library, 538
- graphics-object, 505, 597
- GraphicsEnvironment (java.awt), 266, 551, 569
- GregorianCalendar (java.util), 262
- GridBagLayout (java.awt), 543, 559
- GridLayout (java.awt), 543, 558, 569
- GuardedObject (java.security), 474
- GUI (Graphical User Interface), 44, 190, 507, 508, 520
- GUI and applets, 615
- GUI and applications, 512
- GUI and Java-Beans, 313
- GUI and locale, 263
- GUI and MVC, 515
- GUI and operating-system, 508
- GUI and policytool, 458
- GUI, design of, 513
- Handler, Content-, 369
- handler, exception-, 283
- Handler, Protocol-, 349
- Handler-code, 113
- handler-method, 113
- hard-disk (hard-drive), 17
- hard-disk (hard-drive) and persistent storage, 28, 30, 46, 238, 306, 311, 321, 433, 447, 510
- Hardware, 36
- Hash-Function, 482
- Hashcode, 481
- hashcode, 319
- hashCode()-method (java.lang.Object), 161, 165, 191, 252, 256, 482

- hashCode()-method and identity of an object, 131, 161, 419
- hashCode-value of type int, 161
- hashCode-value: 10 bytes, 483
- hashtable, 233, 237
- Hashtable (java.util), 254, 486, 634
- heavyweight screen-component, 559, 562, 564, 566, 595, 597
- heavyweight screen-component (concept), 353, 548
- Heavyweight-Code, 550
- HelloWorld-application, 71
- Hertz (unit of frequency), 31
- hex-editor, 449, 489
- hex-editor, see Java byte-code, 642
- hexadecimal number-system, 34, 100
- hexadecimally coded, 36, 78, 80, 87, 96, 100, 342, 375, 556
- HFS (Hierarchical File System), scope and access-area, 172
- hierarchical file system, 298
- Hierarchical File System (HFS), scope and access-area, 172
- hierarchical structures, their importance, 543
- hierarchy of a domain-name, 361
- hierarchy of classes, 133, 135, 164, 247
- hierarchy of classes (AWT/Swing), 564
- hierarchy of containment (screen-components), 440, 541, 543
- hierarchy of event-types, 578
- hierarchy of files, 247, 510
- hierarchy of packages, non-existent, 240
- hierarchy of sub-types of Throwable, 281
- hierarchy, class-, 138
- hierarchy, domain-name, 361
- hierarchy, of exceptions, 113
- host of a network, 339
- host-application (of an applet), 613
- Host-Name, 339
- HSB-model, 525
- HSV-like color-models, 525
- HSV-model, 525
- HTML (HyperText Markup Language), 414
- HTML-tags, 21, 324
- HTML-tags (applet), 68, 402, 431, 614
- HTML-tags (servlet), 406, 414
- HTTP, 364
- HTTP (HyperText Transfer Protocol), 369
- HTTP (Hypertext Transfer Protocol), 348, 349, 354, 363, 373, 413
- http://webopedia.internet.com, 660
- hyperlink, 371
- hypertext, 371
- HyperText Markup Language (HTML), 414
- HyperText Transfer Protocol (HTTP), 369
- Hypertext Transfer Protocol (HTTP), 348, 349, 354, 363, 373, 413
- icon, 44, 508
- IDE (Integrated Development Environment), 64, 313, 323, 411
- identifier, 81
- identifier, (variable-), 40
- identifier, access-area of-, 172
- identifier, ambiguous-, 187
- identifier, case-sensitivity, 81
- identifier, scope of-, 122, 171
- identifier, variable-, 74
- identity of a person, 477, 479, 481–483
- identity of an object, 131, 161
- identity of an object and hashCode()-method, 131, 161, 419
- identity of key-pair-owner, 481
- IEEE 754 Standard (for floating-point-numbers), 85
- if-statement (branching), 105
- IllegalAccessError (java.lang), 281
- IllegalAccessException (java.lang), 281
- Image (java.awt), 527, 530, 556
- Image typed reference to native code, 184, 528
- Image, manipulation, 596, 607
- ImageConsumer (java.awt.image), 563, 610
- ImageObserver (java.awt.image), 607
- ImageObserver interface (java.awt.image), 563
- ImageProducer (java.awt.image), 610
- imageUpdate()-method (java.awt.image.
ImageObserver), 530, 563, 607, 608
- imageUpdate()-method, triggerable-, 563
- implements-keyword, 146, 187
- implements-keyword (example), 186
- implements-keyword and multiple super-interfaces, 188
- import-keyword, 244
- import-keyword, invisible-, 244
- Inactive (=Interrupted) thread, 200
- increment operators (pre-, post-), 92
- incremental development (class-extension), 164
- incremental development (trying and retrying), 599, 637
- index color-model, 526
- IndexColorModel (java.awt.image), 526
- IndexOutOfBoundsException (java.lang), 281
- indirect color-model, 526
- inequality operator (! =), 96
- InetAddress (java.net), 151, 381, 421
- information-hiding, 122, 129, 377
- infrastructure, three-tiered communication-, 277, 352, 406, 414
- infrastructure-programmer, 27
- Infrastructure-Programming, 38
- infrastructure-programming (security), 455, 486, 497
- infrastructure-programming (switchboard example), 633
- inheritance, 139
- inheritance, multiple-, 139
- init()-method (java.applet.Applet), 205, 398, 400, 618, 619, 641
- init()-method (javax.crypto.Cipher), 497
- init()-method (javax.servlet.GenericServlet), 411, 641
- init()-method, triggerable-, 398
- Initialized thread, 211
- initializer, array-, 235

- initializer, static- and instance- (member sequence matters), 122
- initializer, static-/instance-, 152
- inner class, 157
- input-stream, 296
- InputEvent (java.awt.event), 579, 581, 582
- InputStream (java.io), 254, 310, 385, 429
- InputStream (java.io) abstract, 296
- InputStreamReader (java.io), 304
- insert()-method (java.lang.StringBuffer), 257
- Inset (java.awt), 558
- instance, 139
- instance and object and variable, 149
- instance-initializer, 152, 441, 641
- instance-initializer and blank-finals, 182
- instance-initializer, not a class-member, 148, 166
- instance-method, 139, 182
- instance-variable, 139, 148, 182
- instance-variable, unshadowing an-, 172
- instanceof-keyword, 191, 498
- instantiation, 74
- int (primitive-type), 85, 95, 169, 642
- int (primitive-type), argument in switch-statement, 106
- int (primitive-type), behavior in a cast, 87
- int (primitive-type), bitmask, 563, 582
- int (primitive-type), buffer-size, 296
- int (primitive-type), coding-constants, 563, 575, 584
- int (primitive-type), default type of number literals, 86
- int (primitive-type), identity-tag of objects, 161
- int (primitive-type), JRE-exit-status, 254, 432
- int (primitive-type), pixel-data, 526
- int (primitive-type), priority of thread, 211
- int (primitive-type), stream-data, 296
- int (primitive-type), TCPport, 388
- int (primitive-type) as color-value, 556
- Integer (java.lang), 169, 181, 252, 256
- integer-literals, 77
- Integrated Development Environment (IDE), 64, 313, 323, 411
- Inter Process Communication (IPC), 230, 231
- inter-application communication, 353
- Interface, 190
- interface, peer-, 59, 251
- interface-class (RMI), 416
- interface-type, 143, 185
- interface-type, arguments of-, 190
- interface-type, type-casting identifiers to-, 194, 404
- interfaces and class-members, 190
- interfaces, list of mentioned-, 653
- InternalError (java.lang), 281
- internationalization, 259
- Internet, 344
- Internet (definition), 372
- Internet Protocol (IP), 344, 348, 349, 354, 357, 359, 372, 377
- Internet Protocol Version 4 (IPv4), 357
- Internet Protocol Version 6 (IPv6), 357
- Internet, its defining protocol-stack, 348
- Internet-Address, 359
- Internet-address (=IP-address), 273, 358
- Interpreter, 39
- Interrupt, 212
- interrupt()-method (java.lang.Thread), 204, 212, 213, 232
- Interrupted (=Inactive) thread, 200
- interrupted()-method (java.lang.Thread), 212, 232
- intialized by assignment: local variable of primitive-type, 121
- intialized to default value: member-variable of primitive-type, 148
- Intranet, 344, 372, 377
- introspection, 629
- introspection (software-components), 311, 318
- introspection: applet, 402
- introspection: package, 245
- introspection: servlet, 411
- invisible extends-Object-clause, 164, 169
- invisible file (in Unix-like operating-systems), 63, 458, 469, 492
- invisible import-java.lang.-clause, 244
- invokeAndWait()-method (javax.swing.SwingUtilities), 590
- invokeLater()-method (javax.swing.SwingUtilities), 590
- IOException (java.io), 281, 318, 330
- IP (Internet Protocol), 344, 348, 349, 354, 357, 359, 372, 377
- IP-Address, 359
- IP-address, 339, 357, 360, 381, 410, 421
- IP-address (assignment organization), 359
- IP-mask, 358, 378
- IPC (Inter Process Communication), 230, 231
- IPv4 (Internet Protocol Version 4), 357
- IPv6 (Internet Protocol Version 6), 357
- is...()-method, 312, 313
- isAlive()-method (java.lang.Thread), 211, 218, 226, 227, 232
- isArray()-method (java.lang.Class), 253
- isAssignableFrom()-method (java.lang.Class), 253
- isDaemon()-method (java.lang.Thread), 224, 232
- isEventDispatchThread()-method (javax.swing.SwingUtilities), 538, 587, 588, 600
- isInstance()-method (java.lang.Class), 253
- isInterface()-method (java.lang.Class), 253
- isInterrupted()-method (java.lang.Thread), 212, 232
- isPrimitive()-method (java.lang.Class), 253
- ItemEvent (java.awt.event), 547, 575, 578
- ItemListener (java.awt.event), 547, 578
- iteration, 108
- J2EE (Java 2 Enterprise Edition), 18, 20, 277, 319, 352, 407, 414, 501
- J2ME (Java 2 Micro Edition), 18, 20
- J2SE (Java 2 Standard Edition), 18, 20, 381
- JApplet (javax.swing), 68, 396, 397
- JApplet (javax.swing) heavyweight, 549, 550, 562, 565, 597
- JApplet as AWT-component, 618

- JApplet is a RootPaneContainer, 566
- jar, Archiving-Tool, 325
- jar, archiving-tool, 54, 242, 243, 250, 489, 492
- JAR- (Java ARchive) file-format, 326
- JAR-File, 326
- JAR-file, 54, 246, 326
- JAR-file (and applet), 403, 614
- JAR-file (and directory-path), 247
- JAR-file (and ext-directory), 64, 179, 243, 245, 487
- JAR-file (and package), 240
- JAR-file (and Provider Infrastructure), 487
- JAR-file (and running applications), 327
- JAR-file (signed), 491
- JAR-file (with prototypes of Java source-code), 645
- JAR-file, signed-, 491
- jarsigner, signing-tool, 489, 492
- Java, 51, 71
- Java 2 Enterprise Edition (J2EE), 18, 20, 277, 319, 352, 407, 414, 501
- Java 2 Micro Edition (J2ME), 18, 20
- Java 2 Standard Edition (J2SE), 18, 20, 381
- Java application, 38, 51, 613
- Java ARchive (JAR) file-format, 326
- Java compiler, 52, 53, 124, 184, 257, 280, 431
- Java compiler and -generated names, 83
- Java compiler and character-encoding, 71, 79
- Java compiler and comparison-operator, 163
- Java compiler and compilation unit, 246
- Java compiler and nested classes, 159
- Java compiler and security, 450
- Java compiler and unreachable statements, 106
- Java compiler searching files, 63, 65
- Java compiler, import-on-demand, 244
- Java compiler, in JDK, 54
- Java compiler, late-binding, 168, 183
- Java compiler, letter-case sensitive, 66
- Java DataBase Connectivity (JDBC), 321
- Java Development Kit (JDK), 54
- Java Development Kit (JDK), installing-, 59
- Java Foundation Classes (JFC), 505, 623
- Java keywords, 81
- Java Runtime Environment (JRE), 51, 52, 250, 431
- Java Runtime Environment (JRE) and abstract references accessing native code, 184, 551
- Java Runtime Environment (JRE) and applets, 397, 613
- Java Runtime Environment (JRE) and application, 72
- Java Runtime Environment (JRE) and daemon-threads, 223
- Java Runtime Environment (JRE) and Java Runtime System (JRS), 55
- Java Runtime Environment (JRE) and security, 128, 448
- Java Runtime Environment (JRE) and servlets, 406
- Java Runtime Environment (JRE) and threading portability, 204
- Java Runtime Environment (JRE) and triggerable methods, 128
- Java Runtime Environment (JRE) as sandbox, 463
- Java Runtime Environment (JRE) communication through streams, 297
- Java Runtime Environment (JRE) DEFINITION, 55
- Java Runtime Environment (JRE) extended by packages, 240
- Java Runtime Environment (JRE), class-loading, 433
- Java Runtime Environment (JRE), communication by method-calls, 414
- Java Runtime Environment (JRE), communication with event-objects, 571
- Java Runtime Environment (JRE), errors and exceptions, 279, 329
- Java Runtime Environment (JRE), garbage-collection, 444
- Java Runtime Environment (JRE), introspection, 254
- Java Runtime Environment (JRE), multiple- on one native platform, 231
- Java Runtime Environment (JRE), tools for single user interfacing, 505
- Java Runtime Environment, Creation-tool: java, 51, 431
- Java Runtime Environment, creation-tool: java, 54, 55, 64, 65, 72, 231, 236, 246, 434, 435, 451, 457, 461, 465, 473, 505
- Java Runtime System (JRS), 51, 53, 54, 65, 249
- Java Runtime System (JRS) and Java Runtime Environment (JRE), 55
- Java Runtime System (JRS) DEFINITION, 56
- Java Virtual Machine, 40
- Java Virtual Machine (JVM), 36, 40, 52, 54, 55, 444
- Java Virtual Machine (JVM) and threads, 203, 216, 232
- Java Virtual Machine (JVM), 32bit bus, 87
- Java Virtual Machine (JVM), How to start?, 431, 641
- java, tool for Creating a Java Runtime Environment, 51, 431
- java, tool for creating a Java Runtime Environment, 54, 55, 64, 65, 72, 231, 236, 246, 434, 435, 451, 457, 461, 465, 473, 505
- Java-Bean, 251, 313
- Java-Beans, 311
- java-tool as trigger for a method-call, 127
- java.beans, switchboarding, 314
- java.lang package automatically imported, 244
- java.policy (text-file), 458, 462, 467, 469, 501
- java.security (text-file), 455, 457, 458, 460, 462, 465, 467, 487, 489, 501
- javac, Compiler-tool, 72, 431
- javac, compiler-tool, 60, 246, 333, 397, 407, 473
- javadoc, documentation-tool, 324
- javap, disassembler-tool, 250, 642
- JavaScript, 338

- JButton (javax.swing), 516, 544, 576, 580, 621
- JComponent (javax.swing), 536, 538, 551
- JComponent (javax.swing) as super-class, 565
- JComponent (javax.swing) methods, 559
- JComponent as drawing-surface, 528, 530, 531, 538, 602
- JComponent as source of event-objects, 591
- JComponent method isLightweightComponent(), 550
- JComponent method paintComponent(), 552
- JComponent opacity, 550
- JDBC (Java DataBase Connectivity), 321
- JDesktopPane (javax.swing), 610
- JDialog (javax.swing), 565, 566
- JDialog (javax.swing) heavyweight, 549, 550, 562, 597
- JDialog (javax.swing) top-level screen-component, 531, 564, 598
- JDK (Java Development Kit), 54
- JDK (Java Development Kit), installing-, 59
- JFC (Java Foundation Classes), 505, 623
- JFrame (javax.swing), 528, 561, 563, 565, 566, 568, 597, 621
- JFrame (javax.swing) heavyweight, 549, 550, 562, 597
- JFrame (javax.swing) top-level screen-component, 531, 598
- JFrame (javax.swing) top-level-screen-component, 564
- JInternalFrame (javax.swing), 566, 567, 598, 610
- JLabel (javax.swing), 544, 566, 621
- JLayeredPane (javax.swing), 567, 568
- JList (javax.swing), 545, 566
- JMenu (javax.swing), 546, 566
- JMenuBar (javax.swing), 546, 567
- JMenuItem (javax.swing), 546
- join()-method (java.lang.Thread), 211, 213, 216, 219, 221, 227, 232, 255, 329
- Joint Photographic Experts Group (JPEG), 369, 522, 529
- JPanel (javax.swing), 566–568, 597, 599, 621
- JPanel as drawing-surface, 608
- JPEG (Joint Photographic Experts Group), 369, 522, 529
- JPopupMenu (javax.swing), 598, 610
- JProgressBar (javax.swing), 547, 548, 566
- JRE (Java Runtime Environment), 51, 52, 250, 431
- JRE (Java Runtime Environment) and abstract references accessing native code, 184, 551
- JRE (Java Runtime Environment) and applets, 67, 397, 613
- JRE (Java Runtime Environment) and application, 72
- JRE (Java Runtime Environment) and applications, 65
- JRE (Java Runtime Environment) and daemon-threads, 223
- JRE (Java Runtime Environment) and security, 128, 448
- JRE (Java Runtime Environment) and servlets, 406
- JRE (Java Runtime Environment) and threading portability, 204
- JRE (Java Runtime Environment) and triggerable methods, 128
- JRE (Java Runtime Environment) as sandbox, 463
- JRE (Java Runtime Environment) DEFINITION, 55
- JRE (Java Runtime Environment) extended by packages, 240
- JRE (Java Runtime Environment), class-loading, 433
- JRE (Java Runtime Environment), communication by method-calls, 414
- JRE (Java Runtime Environment), communication through streams, 297
- JRE (Java Runtime Environment), communication with event-objects, 571
- JRE (Java Runtime Environment), errors and exceptions, 279, 329
- JRE (Java Runtime Environment), garbage-collection, 444
- JRE (Java Runtime Environment), introspection, 254
- JRE (Java Runtime Environment), multiple- on one native platform, 231
- JRE (Java Runtime Environment), tools for single user interfacing, 505
- JRE, Creation-tool: java, 51, 431
- JRE, creation-tool: java, 54, 55, 64, 65, 72, 231, 236, 246, 434, 435, 451, 457, 461, 465, 473, 505
- JRootPane (javax.swing), 561, 566, 568
- JRS (Java Runtime System), 51, 53, 54, 65, 249
- JRS (Java Runtime System) DEFINITION, 56
- JSlider (javax.swing), 547, 548, 574
- JToolTip (javax.swing), 560
- JVM (Java Virtual Machine), 36, 40, 52, 54, 55, 444
- JVM (Java Virtual Machine) and threads, 203, 216, 232
- JVM (Java Virtual Machine), 32bit bus, 87
- JVM (Java Virtual Machine), How to start?, 431, 641
- JWindow (javax.swing), 545, 565, 566, 599, 610, 621
- JWindow (javax.swing) heavyweight, 549, 550, 562, 597
- JWindow (javax.swing) top-level screen-component, 531, 564, 598
- kB (kiloBytes) 1024 bytes, 35, 537
- kernel and front-end, 515
- kernel/front-end model, 515, 516, 542
- Key (java.security), 497
- key of hashtable, 233, 263
- key-chain, 484
- key-pair algorithms, names of-, 481
- key-pair-cryptography, 479
- key-store (definition), 488

- key-store (for public keys), 488
- key-store database, 489
- KeyAdapter (java.awt.event), 581
- Keyboard, 572, 584
- keyboard, 29, 271, 314, 507
- KeyEvent (java.awt.event), 577, 581, 584
- KeyFactory (java.security), 485, 493
- KeyListener (java.awt.event), 577, 581
- keymanagement-tool: keytool, 489, 492
- KeyPair (java.security), 493
- KeyPairGenerator (java.security), 485, 493, 497
- keyPressed()-method (java.awt.event.KeyEvent), 585
- KeyStore (java.security), 492, 494
- keytool, keymanagement-tool, 489, 492
- keywords, Java-, 81
- kiloBytes (kB) 1024 bytes, 35, 537

- Label (java.awt), 566
- labeled statement, 111
- labeled while-statement (a pre-post-condition loop), 111
- language, JavaScript-, 338
- language, non-procedural-, 40
- language, procedural-, 40
- late-binding language, 168
- late-binding of code, 75
- Late-Bound Method, 168
- Latency, 588
- layer-model, 353
- layer-model (RMI), 417
- layer-model of computer-platform, 52, 353, 550
- layer-model of protocols, 348
- layer-model, OSI-, 353
- layers for painting and screen-components, 567, 615
- layers of card-layout, 559
- layout-manager, 543
- layout-manager and absolute positioning, 542
- LayoutManager interface (java.awt), 556, 558
- layouts, multiple- in one container, 559, 566
- least significant byte, 533
- left-associative, 101
- length of a dimension (array), 234
- length()-method (java.io.File), 259
- length()-method (java.io.RandomAccessFile), 299
- length()-method (java.lang.String), 227, 257, 384, 387, 394, 456
- length()-method (java.lang.StringBuffer), 257
- letter, case of, 34, 66, 81, 82, 491
- letter, case of (in domain-names), 361
- letter, case of (in URIs), 375, 394
- letter, case of, in integer-literal of type long, 78
- lifelock (thread-specific problem), 226
- lifo-storage (last-in-first-out), 233
- lightweight process, 230
- lightweight screen-component, 559, 565, 566
- lightweight screen-component (concept), 353, 549
- Lightweight-Code, 550
- line-separator, 254, 304
- List (java.awt), 566, 578
- List (java.util), 237, 238
- List (java.util) (java.awt), 239
- list of listener-objects (with Swing-components), 579
- list()-method (java.io.File), 259
- list()-method (java.rmi.Naming), 417
- list()-method (java.security.Provider), 486
- list()-method (java.util.Properties), 254, 258
- listenerList (variable of javax.swing.JComponent), 579
- ListSelectionEvent (javax.swing.event), 547
- ListSelectionListener (javax.swing.event), 547
- literal, boolean-, 79
- literal, character-, 73, 79
- literal, class-, 252, 437
- literal, floating-point-, 78
- literal, integer-, 77
- literal, literal form, 40, 73, 74
- literal, null-, 144
- literal, number-, 73
- literal, number- (default type: int), 86
- literal, string-, 81, 257
- literals in Java, 77
- literate programming, 323
- little-endian-order of bytes, 533
- loadClass()-method (java.lang.ClassLoader), 255, 438–441
- loading of a class, 152, 433
- local class, 157
- local type, 124
- Local Variable, 121
- local variable, 121
- local variable (and field), 148
- local variable of primitive-type: initialized by assignment, 121
- local variable, final-, 122
- local variable, scope of-, 122
- local variable-declaration, 122
- local variables and threads, 217
- Locale (java.util), 260–263, 402, 617
- locale of screen-component, 542
- localization, 259
- Locking data, 214
- log()-method (java.lang.Math), 95
- logical connection, 343
- Long (java.lang), 169, 181, 256
- long (primitive-type), 85, 95, 169
- long (primitive-type) and JVM, 87, 216
- long (primitive-type), bitmask, 287, 555, 594
- long (primitive-type), byte-count, 296
- long (primitive-type), milliseconds, 223, 254, 261
- long (primitive-type), NEVER length of an array, 234
- long (primitive-type), postfixing literals, 100
- loop/code, post-condition-, 90
- loop/code, pre-condition-, 90
- looping, 90, 108
- looping (do-statement), 109
- looping (for-statement), 109
- looping (while-statement), 108
- lower-case letter, 34, 82, 257, 394

- machine-code, 37, 38

- machine-synchronization, 216
- main()-method characterizes an application, 38, 65, 127, 141, 142, 154
- main()-method, explicitly called-, 630
- main()-method, static-, 71, 142, 154
- main()-method, triggerable-, 72, 127, 185
- main- or main()-thread, 205, 207, 228, 232
- Map (java.util), 237
- Mask, 533
- mask, 287, 555, 563, 576, 579, 582, 591, 593
- mask, IP-, 358, 378
- mask, subnet-, 358, 378
- Math (java.lang), 95, 106, 182, 194, 227, 250, 257
- Math (java.lang) private constructor, 150, 181
- math (java.math package), 251
- mathematical function: exponential function, 94
- mathematical function: natural logarithm, 95
- mathematical function: power function, 95
- mathematical function: square root, 94
- MB (MegaBytes) more than a million bytes, 35, 537
- MegaBytes (MB) more than a million bytes, 35, 537
- MegaHertz (unit of frequency), 31
- Member of a class, 148
- member, accessing class-, 171
- Member, Class-, 148, 166
- member, class-, 147, 148, 154
- member, package-, 240
- member-class, 157
- member-variable, 148
- member-variable of primitive-type: initialized to default value, 148
- members, accessing class-, 156
- members, interfaces and class-, 190
- MemoryImageSource (java.awt.image), 530, 596, 610
- menu, 511, 546
- Menu (java.awt), 566
- menubar, 511
- MenuBar (java.awt), 566
- message, 113
- message (in message-model), 271
- message (in object-orientation), 139
- Message-Digest, 482
- message-digest, 481
- message-model, 271, 606
- message-objects, 137
- MessageDigest (java.security), 485
- method, 139
- Method (java.lang.reflect), 250, 253, 629, 630, 635
- method and behavior (in object-orientation), 133
- method, (static) factory-, 151
- method, (static) factory- (in footnote), 382
- method, abstract- (non-implemented method), 184
- method, abstract- (non-implemented method, concept), 134, 136, 138
- method, accessor- (of a software-component), 312
- method, asynchronous-, 231
- Method, Blocking-, 129
- method, blocking-, 231
- method, class-, 138, 182
- method, content-handler-, 369
- Method, Early-Bound- and Late-Bound-, 168
- method, event-handler-, 287, 575, 577, 586
- Method, Factory-, 151
- method, factory-, 145, 185, 273, 381, 392, 485, 629, 642
- Method, Factory-Like-, 185
- method, factory-like-, 56, 527, 551, 569, 596, 641, 642
- method, final-, called faster, 118, 168, 182
- method, final-, unoverridable, 168, 182
- method, handler-, 113
- method, instance-, 139, 182
- method, native-, 183
- Method, Non-Blocking-, 231
- method, non-blocking-, 231
- method, overloaded-, 139, 168
- method, overridden-, 139, 168
- Method, Overriding-, 168
- method, protocol-handler-, 349
- method, static-, 154, 182
- method, strictfp-, 183
- method, synchronized-, 183
- method, the caller of the, 642
- Method, Triggerable-, 128
- method-call, 118
- method-call history, 470
- method-declaration, 117
- method-Signature, 117, 124
- method-signature (in abstract methods), 184
- method-signature (in interfaces), 187
- method-signature (in overriding methods), 168
- methods for rendering (an image), 536
- methods in sub-classes overriding methods in super-classes, 168
- middleware, 353
- MIME-type, 389, 393
- MIME-types, 368
- minus-operator (binary), 94
- minus-operator (unary), 93
- modal dialog, 564
- modal screen-component, 540, 542
- Model and MVC, 515
- Model Delegate model, 516, 542
- model of thread-states, 210
- Model View Controller model, 515
- model, callback-, 276
- model, chain-, 276, 440, 574
- model, color-, 522
- model, component-, 138, 381, 542
- model, computer-, 19, 23
- model, delegation-event-, 19, 276, 314, 440, 571, 573, 576, 586
- model, event-, 278, 329
- model, event-delegation- (Concept in Java), 278
- model, kernel/front-end-, 515, 516, 542
- model, layer, 353
- model, layer- of computer-platform, 52, 353, 550
- model, message-, 271, 606
- model, Model View Controller, 515

- model, object/class-, 131
- model, object/class- (in programming language), 140
- model, processor-, 39
- model, protocol-, 274
- model, pull-, 274, 520
- model, push-, 520, 574
- model, stream-, 272
- model, subscriber-event-, 276
- model, three-tier-, 352, 406
- model-variable (MVC-concept), 516
- modifier, 117
- Modifier (java.lang.reflect), 250
- modifier, abstract-, 181
- modifier, access-, 177, 179, 180, 429
- modifier, access- (protocol character), 345
- modifier, final-, 182
- modifier, native- (for methods), 183
- modifier, no-, 176, 181
- modifier, private-, 176, 181
- modifier, protected-, 176, 180
- modifier, public-, 176, 180
- modifier, static-, 182
- modifier, strictfp- (for classes and methods), 183
- modifier, synchronized- (for methods), 183
- modifier, transient- (for fields), 183, 309
- modifier, volatile- (for fields), 183
- Modularization, 240
- modulo-operator, 93
- most significant byte, 533
- Mouse, 508, 571, 581
- mouse, 29, 44, 127, 212, 271, 314, 371, 507
- MouseAdapter (java.awt.event), 581, 584, 599
- mouseClicked()-method (java.awt.event.MouseListener), 582, 584, 587, 595, 599, 600
- MouseEvent (java.awt.event), 576–578, 581
- MouseEvent constants, 577
- MouseListener (java.awt.event), 577, 581
- MouseMotionListener (java.awt.event), 577, 581, 601
- mouseMoved()-method (java.awt.event.MouseMotionListener), 583, 587, 595, 601, 602
- multi-processing, 199
- multi-threading, 199
- multicasting, 277, 343
- multiple inheritance, 139
- multiple inheritance and ambiguity, 136
- multiple super-classes (concept), 136, 139, 187
- multiplication-operator, 93
- multipoint connection, 343
- Multipurpose Internet Mail Extension-types, 368
- multithreading (long and double), 87
- MVC, Model View Controller, 515
- name, ambiguous variable-, 187, 239
- Name, Variable-, 40, 74
- name, variable-, 73, 83, 119
- name, variable-, representing a reference, 145
- Name-Space, 123, 429
- name-space, 430
- name-space and security, 450
- name-space considerations, 472
- name-space of class-loader, 436, 469
- name-space of package, 240
- name-space, flat- (of packages), 241, 246
- Naming (java.rmi), 417
- naming conventions, 83, 140
- NaN (Not a Number) constant of Float- and Double-class, 92, 257
- NaN (Not a Number) usage, 86
- NaN-expression (Not a Number), 86, 96
- narrowing an identifier's type, 170
- narrowing cast, 169
- native method, 183
- native modifier (for methods), 183
- Native Software, 53
- natural logarithm (mathematical function), 95
- negation-operator, bitwise- (), 97
- negation-operator, boolean- (!), 93, 97, 98
- NEGATIVE_INFINITY usage, 86
- NEGATIVE_INFINITY, constant of Float- and Double-class, 257
- nested class, 157
- Network Layer (OSI), 354
- network, connection-oriented-, 345
- network, connectionless-, 345
- network, sub-, 343
- network, virtual-, 343, 357
- network-order of bytes, 533
- network-protocol, technology-specific-, 348
- network-topology, 340
- new-keyword and anonymous object of interface-type Runnable, 198
- new-keyword and anonymous objects, 159
- new-keyword and array-type, 234
- new-keyword and factory-methods, 151
- new-keyword and reference-type, 77, 142, 145, 150, 253
- newInstance()-method (java.lang.Class), 151, 253, 439, 499, 619, 642
- no-name-package, 241
- NoClassDefFoundError (java.lang), 61, 171, 281, 436
- node of a network, 339
- Non-Blocking Method, 231
- non-blocking method, 231
- non-local variable, 123
- non-local variable (definition), 117
- non-local variables and encapsulation, 195
- non-localized switching-mechanism, 283
- Nonvisible (state of screen-component), 598
- nonvisible (state of screen-component), 541
- NOT-operator (!), 97, 98
- notify()-method (java.lang.Object), 211, 213, 216, 221, 223, 232, 252, 329
- notify...()-method (custom-defined), 287, 331
- notify...()-method (Swing, ...), 314
- notifyAll()-method (java.lang.Object), 223, 232, 252
- notifyObservers()-method (java.util.Observable), 285, 330
- null-literal, 144
- NullPointerException (java.lang), 281, 289

- Number abstract (java.lang), 181, 256
- number-literal, 73
- number-literal (default type: int), 86
- number-system, binary-, 34
- number-system, decimal-, 34, 79, 257
- number-system, hexadecimal-, 34, 100
- number-system, octal-, 100
- object, 131, 139
- Object (java.lang), 161, 164, 252
- Object (java.lang) array of -, 235
- Object (java.lang) as message-object's type, 284, 285
- Object (java.lang) as super-class, 164, 166, 252, 564
- Object (java.lang) class without a super-class, 164
- Object (java.lang) clone()-method, 151, 168, 192, 258
- Object (java.lang) equals()-method, 163
- Object (java.lang) finalize()-method, 127, 153, 398
- Object (java.lang) getClass()-method, 437
- Object (java.lang) hashCode()-method, 161, 482
- Object (java.lang) in casting, 169
- Object (java.lang) super-class of Throwable, 280, 281
- Object (java.lang) wait() and notify()-methods, 223, 329
- object and instance and variable, 149
- object locked by one of its synchronized instance-methods, 214
- object's state, behavior and identity, 131, 161
- object, anonymous-, 159, 279, 581
- object, constant-, 183
- object, container-, 138
- object, graphics-, 505, 597
- object-code, 38
- object-graph, 161, 191, 233, 307, 441
- object-graph (cloning), 191
- object-graph (in cloning), 191
- Object-Orientation, 131
- object-orientation, 637
- object-orientation (event-messaging), 278
- object-orientation (software-components), 311
- object-orientation and Java, 193
- object-orientation in Java, 142
- object-orientation, behavior and method, 133
- object-orientation, variable and state, 133
- object/class-model, 131
- object/class-model (in programming language), 140
- ObjectInput interface (java.io), 308
- ObjectInputStream (java.io), 298, 308, 318
- ObjectOutput interface (java.io), 308
- ObjectOutputStream (java.io), 298, 307, 308, 318
- Observable (java.util), 284, 330
- Observable-Observer switchboarding, 285
- Observer interface (java.util), 285, 330
- octal number-system, 100
- off-screen rendering (=double buffering), 535, 542, 621
- on-screen rendering, 535
- One-Time-Server, 386
- One-Way-Functions, 480
- open protocol, 347
- open-source, 659
- Operating-System (OS), 17, 44
- operating-system (OS), 338, 516
- operating-system and drivers, 508
- operator, AND- (&&), 98
- operator, assignment- (=), 99
- operator, assignment- (primitive- and reference-type), 144
- operator, bitwise-negation- (), 97
- operator, boolean negation- (!), 93, 97, 98
- operator, conditional- (?), 99
- operator, division-, 93
- operator, equality- (==), 161–163
- operator, inequality- (! =), 96
- operator, minus- (binary), 94
- operator, minus- (unary), 93
- operator, modulo-, 93
- operator, multiplication-, 93
- operator, NOT- (!), 97, 98
- operator, plus- (binary), 94
- operator, plus- (unary), 93
- operator, pre-deprecation-, 127
- operator, qualification-, 182
- operator, remainder-, 93
- operator, shift- (example), 533
- operator, string-concatenation-, 95
- operators, additive-, 94
- operators, assignment-, 99
- operators, binary-, 93
- operators, bitwise shift-, 95
- operators, shift-, 95
- order of bytes, big-endian-, 533
- order of bytes, little-endian-, 533
- order of bytes, network-, 533
- OS (Operating-System), 44
- OS (operating-system), 338, 516
- OSI layer-model, 353
- OutOfMemoryError (java.lang), 281, 444
- output-stream, 295
- OutputStream (java.io), 255, 310, 385
- OutputStream (java.io) abstract, 296
- OutputStreamWriter (java.io), 304
- overflow/underflow warning with floating-point operations: NONE, 86
- overflow/underflow warning with integers: NONE, 86
- overloaded constructor, 151, 174
- overloaded method, 139, 168
- Overloading Methods, 125
- overloading methods and polymorphism, 194
- overridden method, 139, 168
- overriding a static method, 168
- overriding has to extend accessibility, 168
- overriding makes methods accessible, 192, 286
- Overriding Method, 168
- overriding methods and final modifier, 168
- overriding methods and polymorphism, 194
- overriding variables?, 139
- P-code, 39
- pack()-method (java.awt.Window), 562, 598–600

- Package, 240
- Package (java.lang), 245
- package introspection, 245
- package naming, 241
- package, anonymous-, 241
- package, default-, 180
- package, default- (idea), 241
- package, default-, in footnote, 193
- package, install a -, 242
- package, no-name-, 241
- package, unnamed-, 241
- package, write a -, 240
- package-hierarchy non-existent, 240
- packages, application inside named-, 241, 246
- packages, list of mentioned-, 653
- packet (message-model), 272
- Packet (of packet-oriented protocol), 349
- packet or frame (networking-context), 349
- packet-example (footnote), 275
- Packet-Oriented Protocol, 349
- packet-oriented protocol, 275, 351
- packet-routing, 345
- packet-routing, IP-, 358
- packet-segmentation in networks, 351
- paint()-method (java.awt.Component, ...), 128, 185, 205, 397, 527, 536, 539, 552, 595
- paint()-method (javax.swing.JComponent, ...), 559
- paint()-method importing a reference to native code, 185, 527
- paint()-method of applet, 397
- paint()-method, triggerable-, 397, 536, 552, 576, 595, 597
- paint()-method, triggerable- (fixes view), 540
- paint()-method, triggerable- calls the paintComponent()-method, 539, 559
- paintComponent()-method (javax.swing.JApplet) NONE, 615
- paintComponent()-method (javax.swing.JComponent, ...), 128, 185, 527, 528, 531, 539, 540, 551, 559, 576, 595–597, 621
- paintComponent()-method importing a reference to native code, 185, 527
- paintComponent()-method, triggerable-, 528, 531, 536, 539, 551, 559, 576, 595, 596
- paintComponent()-method, triggerable- (fixes view), 540
- PaintEvent (java.awt.event), 573, 576, 577, 595, 598
- painting screen-components, 597
- painting-methods, 595
- Pairing-Quandary, 480
- pairing-quandary, 483
- Panel (java.awt), 559, 597
- Panel heavyweight, 566
- Panel, super-class of Applet, 251, 614, 615
- Parallel-Server, 386
- parameter of a method-declaration or -definition, 116
- parent in containment-hierarchies, 543
- parent of screen-component, 543
- parent-class, 139
- parser, 648
- partial transparency, 527, 550
- passing arguments by-reference, 120, 256
- passing arguments by-value, 120, 225, 256
- path in a network, 342
- Path-Name, 47
- path-name and package-name, 240, 242, 244
- path-name and property expansion, 458
- path-name and property expansion (example), 459
- path-name and URI, 409, 410
- path-name in URL, 364, 374
- path-name separator, 48, 254
- Path-Name, Absolute-, 48
- path-name, absolute- (example), 510
- Path-Name, Relative-, 48
- path-name, relative- (example), 393
- path-separator, 48, 254
- PATH-variable (operating-system), 60, 409
- PDF (Portable Document Format), 369, 414
- Peer, 353
- peer, 353, 548
- peer-based screen-components (concept), 548
- peer-code, 598
- peer-interface, 59, 251
- peer-to-peer communication, 275, 352, 516
- peer-to-peer connections, 273
- peer-to-peer HTTPProtocol, 369
- Permission (java.security), 453, 455, 459, 465, 476, 501
- PermissionCollection (java.security), 455, 466
- Permissions (java.security), 455, 469
- Persistence, 238, 321
- persistence and serialization of objects, 306, 318
- persistent database, non- (aggregation), 138, 233
- persistent path-name-setting (operating-system), 63
- persistent storage (example), 298
- persistent storage and communication over time-intervals, 231
- persistent storage and databases, 321
- persistent storage and file-systems, 46, 238
- persistent storage and hard-disk (hard-drive), 28, 30, 46, 238, 306, 311, 321, 433, 447, 510
- persistent storage and packages, 249
- persistent storage and policy-files, 455
- persistent storage and random-access-files, 299
- persistent storage and serialization of objects, 443
- persistent storage and software-components, 311, 318
- persistent storage and transient member-variables/fields, 148, 183
- persistent storage of cryptographic keys (key-store), 488
- persistent storage of registered security-providers, 487
- persistent storage, non- (Random Access Memory), 28
- persistent storage-device and class-loading, 433
- physical connection, 340
- Physical Layer (OSI), 354
- pipe, 292

- pipedReader (java.io), 300
- PipedOutputStream (java.io), 300
- PipedReader (java.io), 300
- PipedWriter (java.io), 300
- Pixel (picture-element), 519
- pixel's partial transparency, 527
- pixel, screen-, 523
- pixel-coordinates, 536, 542
- pixel-format, ARGB, 526
- pixel-format, indirect, 526
- PixelGrabber (java.awt.image), 610
- pixelgrabbing, 535
- Platform, 17, 44
- platform, virtual-, 17, 36, 51, 55, 151, 333
- plug-in (handler), 113, 325, 369
- plus-operator (binary), 94
- plus-operator (unary), 93
- Point (java.awt), 555
- point-to-point connection, 277, 343
- Policy (java.security), 455, 457, 458, 460, 462, 465, 469
- policy-file and public key, 459, 492
- policy-file, conventions for the, 458
- policytool (footnote), 458
- polymorphism, 194
- POP, 364
- POP (Post Office Protocol), 371, 373
- popup-menu, 511
- PopupMenu (java.awt) top-level screen-component, 564
- Port, 231
- portability, 51
- portability (and Java language), 333
- portability (AWT and Swing), 548
- portability and 100 Percent Java (footnote), 72
- portability and multithreading, 203
- portability and native methods, 118, 183
- portability and pre-written packages, 249
- portability and runtime environments, 55
- portability of content-types, 265
- portability-problem, 45, 47, 51
- Portability, 53
- Portable Document Format (PDF), 369, 414
- Portable Software, 53
- portable source-code, 45
- POSITIVE_INFINITY usage, 86
- POSITIVE_INFINITY, constant of Float- and Double-class, 257
- Post Office Protocol (POP), 371, 373
- post-condition loop/code, 90, 109
- postEvent()-method (java.awt.Component, java.awt.EventQueue, ...), 587, 591, 592
- pow()-method (java.lang.Math), 95
- power function (mathematical function), 95
- pre-condition loop/code, 90, 109
- pre-condition loop/code (example), 297
- pre-deprecation-operator, 127
- precedence, 101
- preemptive thread-scheduling, 200, 203, 213
- prepareImage()-method (java.awt.Component), 232, 529, 596
- Presentation Layer (OSI), 354
- primitive-type, 83, 87, 142, 191
- primitive-type and atomic operations, 214
- primitive-type and comparison-operator (==), 162
- primitive-type, type-casting of-, 100
- println()-method (java.lang.System.out), 37, 71, 254, 637
- println()-method (java.lang.System.out) (footnote), 72
- printStackTrace()-method (java.lang.Throwable), 282
- PrintStream (java.io), 253
- PrintWriter (java.io), 310, 413
- priority-based thread-scheduling, 200, 202, 213, 641
- private access, 176
- private class-member, 148
- private constructor, 150, 181, 182, 382, 417
- private constructor and singleton, 160
- private key (for decrypting), 481
- private key (for signing), 482, 488, 492
- private key (for signing), an example, 490
- private key format (PKCS8), 481
- private key in key-store, 488
- private key producing a one-way-function, 480
- private modifier, 176, 181
- PrivateKey (java.security), 493
- PrivilegedAction interface (java.security), 470, 471
- PrivilegedExceptionAction (java.security), 471
- problem tutoring, 512
- Process (java.lang), 56, 250, 255
- process and applet, 613
- process and thread, 230, 254, 273, 293, 321
- process...()-method, triggerable-, 287
- processEvent()-method, triggerable-, 591, 593
- processing-graph, 520
- Processor, 28
- processor's address-space, 28, 291, 522, 535
- processor-model, 39
- profiling, 629
- Program, 17, 35
- program, 160
- program and applet, 67
- program and application, 38
- program and compiler, 39
- program and interpreter, 39
- program and operating-system, 43, 44
- program and runtime-environment, 55
- program, Java-, 51
- program, native-, 53
- programmer, application-, 27
- programmer, infrastructure-, 27
- programmer, system-, 27
- programmer, utility-, 27
- Programming, 38
- programming language, non-procedural-, 40
- Programming, Application-, 38
- programming, distributed-, 419, 423
- Programming, Infrastructure-, 38

- programming, infrastructure- (security), 455, 486, 497
- programming, infrastructure- (switchboard example), 633
- programming, literate-, 323
- programming, symbol-oriented-, 40, 71
- programming, visual- (with software-components), 311, 313
- programming-language, JavaScript-, 338
- programming-language, procedural-, 40
- Properties (java.util), 254, 258, 444, 486
- properties and variables, 312
- property (of a software-component), 312
- property expansion with text-files, 458, 459
- property, bound- (of software-component), 315
- property, constrained- (of software-component), 315
- Property-Expansion, 458
- PropertyChangeEvent (java.beans), 314, 331, 578, 603
- PropertyChangeListener (java.beans), 314, 319, 331, 578
- PropertyChangeSupport (java.beans), 314, 319, 331
- PropertyPermission (java.util), 469
- protected access, 177
- protected constructor, 151, 181
- protected modifier, 176, 180
- Protection (security-related), 449
- protection-domain, 454, 466, 469, 471
- ProtectionDomain (java.security), 453, 454, 458, 465, 468, 469, 501, 629
- Protocol, 274, 275
- protocol, 122, 190, 231, 279, 289, 347
- Protocol (message-model), 275
- protocol and type, 347
- protocol, application-, 348, 367, 378
- protocol, application-, and servlet, 405
- protocol, application-, and TCPport-number, 363
- protocol, application-, and URL, 373
- protocol, application-, and URL-class, 388
- Protocol, Connection-Oriented-, 350
- protocol, connection-oriented-, 275, 351
- protocol, connectionless-, 350
- protocol, DNS, 382
- protocol, open-, 347
- Protocol, Packet-Oriented-, 349
- protocol, packet-oriented-, 275, 351
- protocol, service- (or application-), 378
- protocol, signaling-, 342
- protocol, state-bound-, 351
- protocol, stateless-, 351
- protocol, transport-, 342, 348
- protocol, virtual-network-, 348
- Protocol-Handler, 349
- protocol-model, 274
- protocol-stack, 348, 512
- protocols, layer-model of, 348
- Provider (java.security), 486, 499
- Provider Infrastructure, 487
- provider-infrastructure, 485
- Pseudo-code, 39
- public access, 177
- public class extended over different packages, 438
- public class has to be in class of same name, 146, 241, 242, 245, 246
- public constructor, 180
- public key (distribution problem), 481, 483
- public key (for authentication), 482
- public key (for encrypting), 481
- public key and identity of owner, 481
- public key and policy-file, 459, 492
- public key producing a one-way-function, 480
- public keys in key-stores, stored as certificates, 488
- public modifier, 176, 180
- public-key-file (example), 493
- PublicKey (java.security), 493, 494
- pull-model, 274, 520
- push-model, 520, 574
- QoS (Quality of Service), 345, 351, 354, 423, 444
- QoS (Quality of Service) concept, 447
- qualification-operator, 182
- quality of a cryptographic algorithm, 479
- Quality of Service (QoS), 339, 345, 351, 354, 423, 444
- Quality of Service (QoS) concept, 447
- queue, event-, 586
- race-condition (thread-specific problem), 225
- railroad-diagram, 76
- RAM (Random Access Memory), 28, 73, 76
- RAM-address, 73, 143
- Random (java.util), 257
- Random Access Memory (RAM), 28, 73, 76
- random()-method (java.lang.Math), 106, 112, 219, 229, 257, 531
- random-access, 233, 291
- random-access-file, 299, 321
- RandomAccessFile (java.io), 299, 321
- rasterization, 535
- read()-method (java.io.FilterInputStream, java.io.FilterReader), 302
- java.io.FilterReader) triggerable, 306
- read()-method (java.io.InputStream, java.io.Reader), 127, 232, 291, 295
- read()-method (java.io.InputStream, java.io.Reader) abstract, 296
- Reader (java.io), 304, 310
- Reader (java.io) abstract, 296
- readLine()-method (java.io.BufferedReader), 300, 387, 390, 394, 456
- readObject()-method (java.io.ObjectInputStream), 308–310, 318
- real-life problems, 512
- Rectangle (java.awt), 555, 607
- Recursion, 125
- recursive data-structures (not treated), 233
- redundant information, 522
- reference, passing arguments by-reference, 120, 256
- Reference-Type, 143

- reference-type, 143, 191
- reference-type and atomic operation, 214
- reference-type and equality operator (==), 162
- reference-type, type-casting of-, 169
- reference-type, variable of-, 143
- references, concept of weak-, 445
- reflection (Java language), 252, 442, 629
- reflection (software-/screen-components), 542
- reflection (software-components), 251, 318, 319
- registry (message-model), 272
- Relative Path-Name, 48
- relative path-name (example), 393
- remainder-operator, 93
- Remote interface (java.rmi), 189, 417
- Remote Method Invocation (RMI), 19, 64, 251, 273, 310, 338, 373
- Remote Method Invocation (RMI), introduction, 416
- rendering an image, 519
- rendering an image (off-screen), 535, 542, 621
- rendering an image (on-screen), 535
- rendering as coding-transformation, 535
- rendering of images, 587
- rendering-method, 536
- repaint()-method (java.awt.Component), 128, 405, 596–598, 601, 602
- repaint()-method (java.awt.Component), example, 602
- RepaintManager (javax.swing), 560, 599
- repository (message-model), 272
- resolution in dpi (dots per inch), 536, 537
- Resource, 44
- ResourceBundle (java.util), 260, 263
- return-keyword, 118
- return-statement, 118
- reusability (of software-components), 313
- RGB-model, 523
- right-associative, 101
- RLE-compression (Run Length Encoding), 522
- RMI, 414
- RMI (Remote Method Invocation), 19, 64, 251, 273, 310, 338, 373
- RMI (Remote Method Invocation), introduction, 416
- RMI layer-model, 417
- RMI-compiler-tool: rmic, 415
- RMI-registry-tool: rmiregistry, 415
- rmic, RMI-compiler-tool, 415
- rmiregistry, RMI-registry-tool, 415
- RootPaneContainer (javax.swing), 563, 566, 615, 618
- round-robin-scheme of scheduling threads of same priority, 202
- router, 290
- rt.jar, 54, 247, 249, 389
- rt.jar (loading), 434
- rt.jar, peeking inside-, 243, 244
- run()-method (java.lang.Runnable), 160, 198, 204, 205, 207–210, 255, 256, 258, 329, 386, 401, 590, 641
- run()-method (java.lang.Runnable) abstract, 189
- run-time typed language, 75
- Runnable interface (java.lang), 189, 209, 228, 232, 255, 256, 258, 386, 401, 590
- Runnable interface (java.lang) instantiated anonymously, 208
- Runnable interface (java.lang), introduction, 205
- Runnable thread, 200, 211
- Running thread, 211
- Runtime (java.lang), 56, 250, 254, 258, 402, 432, 445, 641
- Runtime Environment, 55
- runtime-representation of a class, 54, 255, 433, 441, 443, 629
- Runtime.exit()-method (java.lang), 254
- Runtime.gc()-method (java.lang), 254
- RuntimeException (java.lang), 279, 281
- RuntimePermission (java.lang), 468, 472
- Safe Channel, 483
- safe channel, 449, 479, 481, 483
- Sandbox, 463
- sandbox, 68, 333, 398, 406, 432, 475, 613
- scheduling of events, 572, 574
- scheduling of threads, 53, 200, 213, 216, 228, 230
- scheduling threads in Java, 203
- SCOPE, 122
- scope, 430
- scope and access-area, 172
- scope and access-area in an Hierarchical File System (HFS), 172
- scope and name-space, 123
- scope of identifier, 122, 171
- scope of local variable, 122
- screen-component, 331, 399, 509, 510, 519
- Screen-Component (concept), 540
- screen-component and its GUI-related states, 509, 598
- screen-component as a painting-surface, 538
- screen-component as container, 542
- screen-component on focus, 542, 577
- screen-component, heavyweight-, 559, 562, 564, 566, 595, 597
- screen-component, heavyweight- (concept), 353, 548
- screen-component, lightweight-, 559, 565, 566
- screen-component, lightweight- (concept), 353, 549
- screen-component, modal-, 540, 542
- screen-component, visible, showing, nonvisible, 598
- screen-pixel, 523
- Scrollbar (java.awt), 548, 559
- ScrollPane (java.awt), 559
- ScrollPaneLayout (javax.swing), 543
- secure class-loader, 454, 458, 469, 471
- secure stream, 501
- SecureClassLoader (java.security), 465, 469, 629
- Security (java.security), 485–487, 499
- security-manager, 462
- security-manager (and access-control-mechanism), 454, 455
- security-manager (and applets), 475
- security-manager (and class-loading), 440

- security-manager (autistic and free), 255
- security-manager (its various uses), 475
- security-manager (run Java code with and without a-), 460, 465
- security-manager and (security-)sandbox, 463
- security-related-applications, 488
- Security-Sandbox, 463
- security-sandbox, 68, 475
- SecurityException (java.lang), 281, 463
- SecurityManager (java.lang), 250, 254, 255, 427, 440, 453, 456, 476
- SecurityManager (java.lang) core, 462
- SecurityPermission (java.security) in footnote, 465
- segmentation of packets in networks, 351
- self-signed certificate, 484
- semantic event, 578
- semantic manipulation, 35, 73
- semi-transparency, 550
- separator, folder-/directory-/file-, 46, 254
- separator, folder-/directory-/file- (in packaging), 242
- separator, folder-/directory-/file- (policy-files), 458, 462
- separator, folder-/directory-/file- (URL), 364, 388
- separator, line-, 254, 304
- separator, path-, 48, 254
- separators, 641
- separators, table of-, 48
- sequence of constructor-calls, 152
- Sequential-Server, 386
- secret-key-cryptography, 478
- Serializable interface (java.io), 189, 307, 312, 318, 563
- Serialization, 307
- Server, 276
- Server, One-Time-, 386
- Server, Parallel-, 386
- Server, Sequential-, 386
- server-side computing, 406
- ServerSocket (java.net), 273, 385, 421
- service()-method (javax.servlet.Servlet, ...) abstract, 128, 407, 408, 411
- service()-method, triggerable-, 407
- service- (or application-) protocol, 378
- Servlet, 406
- servlet introspection, 411
- servlet's triggerable method, 407
- servlet-container (=servlet-sandbox), 406, 407, 409, 430
- servlets and web-applications, 414
- Session, 277
- Session Layer (OSI), 354
- set...()-method, 312, 313, 318
- setContentHandlerFactory()-method (java.net.URLConnection), 190, 393, 394
- setDaemon()-method (java.lang.Thread), 224, 232
- setName()-method (java.lang.Thread), 212
- setPriority()-method (java.lang.Thread), 212, 224, 232
- setSecurityManager()-method (java.lang.System), 254, 255, 463–465
- setURLStreamHandlerFactory()-method (java.net.URL), 190, 391, 394
- setVisible()-method (java.awt.Component), 528, 552, 595, 597, 598
- shadowing a static variable, 168
- shadowing variables, 123, 126, 167, 168
- Shallow Cloning, 191
- shallow equal object-graphs, 162, 192
- shift operators, 95
- shift-operator (example), 533
- Short (java.lang), 169, 181, 256
- Short (java.lang), un/wrapping, 256
- short (primitive-type), 85, 169
- short (primitive-type), length of an array, 234
- short (primitive-type), un/wrapping, 256
- Showing (state of screen-component), 598
- showing (state of screen-component), 541, 598
- signaling-protocol, 342
- Signature (java.security), 485, 492, 494
- signature, digital-, 483
- signature, digital- and certification, 484
- Signature, method-, 117, 124
- signature, method- (in abstract methods), 184
- signature, method- (in interfaces), 187
- signature, method- (in overriding methods), 168
- signature-file (example), 493
- signed class = signed class-file, 494
- signed data (example), 493
- signed data and authentication (concept), 482
- signed data and policy-files, 459, 492
- signed data and trust, 478
- signed data in class-loading, 469
- signed data-files, by user, 488
- signed data-files, byte-code, classes, objects, 494
- signed JAR-file, 491
- SignedObject (java.security), 494
- signing data by programmer, 492
- signing public keys (certification), 483
- signing-tool: jarsigner, 489, 492
- Simple Mail Transfer Protocol (SMTP), 371, 373, 389
- Singleton, 160
- singleton, 463
- skeleton-class (RMI), 416
- sleep()-method (java.lang.Thread), 204, 211, 212, 223, 232
- SMTP, 364
- SMTP (Simple Mail Transfer Protocol), 371, 373, 389
- Socket, 273
- Socket (java.net), 273, 382, 385
- socket, TCP-, 273, 383, 385
- socket, TCP/UDP-, 364
- SocketPermission (java.net), 459
- Software, 36
- Software, Native-, 53
- Software, Portable-, 53
- software-component (concept), 311
- software-components: introspection, 311
- source of message-model, 271
- Source-Code, 38

- source-code, 37
- source-code written with a text-editor, 53, 54, 57, 62, 63, 65, 323
- source-code, portable-, 45
- source-code, programmer comment, 37, 71, 323
- space, address-, 28
- space, address-, and port, 231, 271, 337
- space, address-, of a processor, 291, 522, 535
- spawn a thread, 206
- SQL (Structured Query Language), 20, 40, 252, 321
- sqrt()-method (java.lang.Math), 94
- square root (mathematical function), 94
- stack (aggregate of protocols), 348, 512
- stack (aggregate), 138, 233, 292
- stack (aggregate, processor-state), 217, 271, 281
- stack of cards, 202
- stack, protocol-, 348
- StackOverflowError (java.lang), 281
- standalone code (=program), 160
- start()-method (java.applet.Applet), 205, 398, 400, 618, 641
- start()-method (java.lang.Thread), 204, 206–209, 211, 212, 232, 386, 401
- start()-method, triggerable-, 398
- starvation (thread-specific problem), 224
- state and variable (in object-orientation), 133
- state of an object, 131, 161
- state-bound protocol, 351
- stateless protocol, 351
- statement, 103
- statement, break-, 110
- statement, continue-, 110
- statement, do- (looping), 109
- statement, empty-, 103
- Statement, Expression and Control Structure (concept), 89
- statement, for- (looping), 109
- statement, if- (branching), 105
- statement, labeled-, 111
- statement, return-, 118
- statement, switch- (branching), 106
- statement, synchronized- (multithreading), 216
- statement, throw- (messaging with Throwable-objects), 279
- statement, try- (messaging with Throwable-objects), 282
- statement, unreachable-, 106
- statement, while- (looping), 108
- states of a screen-component, conceptual-, 509, 598
- static blank-final, 182
- static class-member, 140, 148, 154
- static class-members remain unserialized, 309
- static factory-method, 151
- static factory-method (footnote), 382
- static member-class, 158, 182
- static method, 138, 154, 182
- static method, overriding a-, 168
- static modifier, 182
- static synchronized methods lock their class, 214
- static variable (class-variable), 138, 148, 154, 182
- static variable in an abstract class, 184
- static variable, unshadowing a-, 173
- static-initializer, 152, 441, 641
- static-initializer and static blank-finals, 182
- static-initializer, not a class-member, 148, 166
- static-keyword and class-member, 154
- statically-typed language, 75
- stop()-method (java.applet.Applet), 205, 398–400, 618, 641
- stop()-method, triggerable-, 398
- Stream, 273
- stream, 19, 233
- Stream (Java), 295
- stream (message-model), 272
- stream, content-, 389, 394
- stream, filter-, 292
- stream, I/O-, 292
- stream, input-, 296
- stream, output-, 295
- stream, secure-, 501
- stream, wrapper-, 292
- stream-example (footnote), 275
- stream-model, 272
- stream-objects and -classes, 295
- streams and variables, 292
- streams in Java, 291
- streams to aggregates, 237
- streams, piped-, 292
- strength of an cryptographic algorithm, 479
- strictfp, 118
- strictfp method, 183
- strictfp modifier (for classes and methods), 183
- String (java.lang), 81, 194, 250, 257, 520, 642
- String (java.lang) casting, 170
- String (java.lang) comparing, 162
- String (java.lang) exceptionality, 77, 145
- string-concatenation, 95
- string-concatenation-operator, 95
- string-literal, 81, 257
- StringBuffer (java.lang), 120, 250, 257, 642
- StringBufferInputStream (java.io), 237
- StringReader (java.io), 291, 310
- StringTokenizer (java.util), 233, 237, 301, 386
- StringWriter (java.io), 310
- Structured Query Language (SQL), 20, 40, 252, 321
- stub-class (RMI), 416
- sub-class (concept), 135
- sub-classes with methods overriding methods in super-classes, 168
- sub-network, 343
- subdomain (DNS), 360
- subnet-mask, 358, 378
- subscriber event-model, 276
- super()-constructor, 149
- super()-constructor (class-extension), 166
- super()-constructor, example, 304
- super-class, 164
- super-class (concept), 135

- super-class, class without a- (java.lang.Object), 164
- super-classes and casting, 169, 253
- super-classes and class-loading, 441
- super-classes containing methods overridden by methods of sub-classes, 168
- super-classes in graphical representation, 562
- super-classes, multiple- (concept), 136, 139, 187
- super-interfaces and implements-keyword, 188
- super-keyword, 153, 173, 593
- super-keyword for un-overriding from OUTSIDE a class, 175, 192, 286
- super-keyword for un-overriding from WITHIN a class, 174
- super-keyword undefined in static methods, 182
- super-keyword used for unshadowing, 173
- Swing, 505
- Swing (basics), 548
- Swing as API, 52
- Swing versus AWT (Abstract Window Toolkit), 520, 548
- Swing, applets, 613
- Swing, class-hierarchies, 564
- Swing, desktop, 610
- Swing, image drawing, 519, 595
- Swing, screen-components, 551
- SwingUtilities (javax.swing), 538, 588–590
- switch-statement (branching), 106
- switch-statement (branching), example, 401
- switchboard (message-model), 337
- switchboard (OSI layer-model), 354
- switchboard and pairing-quandary, 480
- Switchboard routes messages (message-model), 272
- switchboard, custom-written, 633
- switchboard-mechanism, advanced AWT-, 594
- switchboard-mechanism, AWT-, 571
- switchboarding, java.beans, 314
- switchboarding, Observable-Observer-, 285
- switchboarding, try-catch clause, 282
- switchboarding-structures, 276
- switching-mechanism, non-localized-, 283
- symbol-oriented-programming, 40, 71
- symmetric-key-cryptography, 478
- synchronized block, 216
- synchronization, machine-, 216
- synchronized instance-methods lock their object, 214
- synchronized method, 183
- synchronized modifier (for methods), 183
- synchronized-statement (multithreading), 216
- Syntax Diagrams, 655
- syntax-diagram, 76
- System (java.lang), 56, 250, 253, 254, 258, 402, 505
- System (java.lang) stream, 274
- system-programmer, 27
- System-Property, 259
- system-property and property-expansion, 458
- System.arraycopy()-method (java.lang), 236, 630
- System.currentTimeMillis()-method (java.lang), 261
- System.exit()-method (java.lang), 254, 432, 581, 651
- System.exit()-method (java.lang), example, 226, 382
- System.gc()-method (java.lang), 153, 223, 254, 445, 598
- System.getProperties()-method (java.lang), 258
- System.in (java.lang) stream, 297
- System.out (java.lang) stream, 297
- System.out.println()-method (java.lang), 71, 254, 637
- System.out.println()-method (java.lang) (footnote), 72
- System.setSecurityManager()-method (java.lang), 255, 463
- SystemColor (java.awt), 527
- TCP (Transmission Control Protocol), 363, 378, 385
- TCP-port, 643
- TCP-port 80 (HTTP), 371
- TCP-port in URL, 374
- TCP-socket, 273, 383, 385
- TCP/UDP-port, 363
- TCP/UDP-socket, 364
- technology-specific network-protocol, 348
- TELNET, 364, 371, 372
- terminal-based user interface, 507
- terminal-window, 44, 54, 55, 64
- terminal-window (and appletviewer-command), 396, 431, 613, 641
- terminal-window (and CTRL-C-command), 432
- terminal-window (and jar-command), 326, 645
- terminal-window (and java-command with jar-option), 327
- terminal-window (and java-command with security-option), 462
- terminal-window (and java-command with String-arguments), 236
- terminal-window (and java-command with verbose-option), 250, 434
- terminal-window (and java-command with version-option), 59, 65, 250
- terminal-window (and java-command), 51, 65, 431, 505, 619
- terminal-window (and java-command, triggers main()-method), 116, 127, 142
- terminal-window (and javac-command), 65, 72, 431
- terminal-window (and javadoc-command), 324
- terminal-window (and javap-command), 642
- terminal-window (and keytool-command), 489
- terminal-window (and policytool-command), 458
- terminal-window (and RMI-related commands), 418
- terminal-window (and servlet-related commands), 407
- terminal-window (and Uniform Resource Locator (URL)), 373
- terminal-window (as single user interface), 508
- terminal-window (its current directory), 48
- terminal-window (Linux operating-system), 62

- terminal-window (Mac OS X), 61
- terminal-window (multiple- for multiple JREs), 52, 231
- terminal-window (typical java-commands), 37
- terminal-window (Windows operating-system), 63
- text-based document format, 414
- text-based user interface, 507
- text-editor for editing security- and policy-files, 453, 457, 458, 496
- text-editor for writing source-code, 53, 54, 57, 62, 63, 65, 323
- text-editors and text-model, 543
- this()-constructor, 167
- this()-constructor, example, 568
- this-keyword, 173, 207, 217, 223, 580, 596, 630
- this-keyword undefined in static methods, 182
- this-keyword used for unshadowing, 172
- this-keyword with little sense, 227
- Thread (java.lang), 206, 210, 216, 232, 256, 329
- Thread (java.lang) cast, 442
- thread communication, with objects, 225
- thread, Blocked-, 199, 211
- thread, Daemon-, 223
- thread, daemon-, 215, 432, 586
- thread, Event-Dispatch-, 586
- thread, event-dispatch-, 205, 228
- thread, event-dispatch-, and painting-methods, 595
- thread, event-dispatch-, running code on the-, 590
- thread, finalize-, 228
- thread, garbage-collection-, 223, 228, 444
- thread, Inactive (=InterruptedException), 200
- thread, Initialized-, 211
- thread, Interrupted- (= Inactive), 200
- thread, main- or main()-, 205, 207, 228, 232
- thread, method-call history of a-, 470
- thread, Runnable-, 200, 211
- thread, Running-, 211
- thread, spawn a -, 206
- thread-context-switch, 230
- thread-scheduling, 53, 200, 213, 216, 228, 230, 641
- thread-scheduling in Java, 203
- thread-scheduling, preemptive, 200, 203, 213
- thread-scheduling, priority-based-, 200, 202, 213, 641
- thread-scheduling: round-robin-scheme, 202
- thread-scheduling: time-slicing, 201, 203, 213, 641
- thread-specific problem (deadlock), 226
- thread-specific problem (dining humans), 215
- thread-specific problem (livelock), 226
- thread-specific problem (race-condition), 225
- thread-specific problem (starvation), 224
- thread-states, 210
- ThreadGroup (java.lang), 250, 256
- threads and local variables, 217
- three-tier-model, 352, 406
- three-tiered communication infrastructure, 277, 352, 406, 414
- throw-statement, 279, 330
- throw-statement (messaging with Throwable-objects), 279
- throw-statement (security-exception), 463
- Throwable (java.lang), 256, 280, 284, 329
- Throwable (java.lang) hierarchy, 281
- throws-keyword and checked exceptions, 281, 330
- throws-keyword in constructor-declaration, 150
- throws-keyword in method-declaration, 117, 280
- tiered communication infrastructure, three-, 277, 352, 406, 414
- time-slicing, a form of thread-scheduling, 201, 203, 213, 641
- Timer (javax.swing), 576, 603
- toCharArray()-method (java.lang.String), 291, 495
- tool, 54
- tool tutoring, 512
- tool(Archiving): jar, 325
- tool(archiving): jar, 54, 242, 243, 250, 489, 492
- tool(Compiler): javac, 72, 431
- tool(compiler): javac, 60, 246, 333, 397, 407, 473
- tool(disassembler): javap, 250, 642
- tool(documentation): javadoc, 324
- tool(Java Runtime Environment Creation): java, 51, 431
- tool(Java Runtime Environment creation): java, 54, 55, 64, 65, 72, 231, 236, 246, 434, 435, 451, 457, 461, 465, 473, 505
- tool(keymanagement): keytool, 489, 492
- tool(RMI-compiler): rmic, 415
- tool(RMI-registry): rmiregistry, 415
- tool, application as tool for solving real-life-problems, 507
- tool, Bean-Box, 313
- tool, builder- (for software-components), 311
- tool: jarsigner, 489, 492
- toolbar (tool-bar), 511
- Toolkit (java.awt), 254, 332, 529, 556, 587, 603, 607
- Toolkit typed reference to native-code, 185
- Toolkit typed reference to native-code, getting a-, 528, 551, 555, 579, 596
- Tooltip, 601
- tooltip, 559, 599
- top-level window, 577
- top-level window, concept of-, 509
- top-level windows and heavyweights, 550, 565
- top-level-class, 157
- top-level-domains, country-code- (DNS), 361
- top-level-domains, generic- (DNS), 361
- topology of a network, 340
- toString()-method (java.lang.Object), 161, 165, 252, 256
- toString()-method (java.lang.StringBuffer), 257
- toString()-method (java.lang.Throwable), 282
- toString()-method (java.util.EventObject), 288
- totalMemory()-method (java.lang.Runtime), 254, 258, 445
- toUpperCase()-method (java.lang.String), 102
- transformation, coding-, 35, 520, 534, 535
- transformation, content-, 35, 520
- transient field (non-serializable), 183, 309

- transient modifier (for fields), 183, 309
- Transmission Control Protocol (TCP), 363, 378, 385
- transparency, 139, 377, 526, 550, 556
- transparency close to encapsulation, 139, 367, 377, 382
- transparency, partial, 527
- Transparent, 367
- Transport Layer (OSI), 354
- transport-protocol, 342, 348
- tree-structure, 233, 543
- tree-structure of a class-hierarchy, 135
- tree-structure of a Hierarchical File System, 46, 326
- tree-structure of texts, 516
- triggerable check...()-methods, 462
- triggerable destroy()-method, 398
- triggerable finalize()-method, 127, 153
- triggerable imageUpdate()-method, 563
- triggerable init()-method, 398
- triggerable main()-method, 72, 127, 185
- Triggerable Method, 128
- triggerable method of servlets, 407
- triggerable methods of applets, 397, 614, 618
- triggerable paint()-method, 397, 536, 552, 576, 595, 597
- triggerable paint()-method (fixes view), 540
- triggerable paint()-method calls the paintComponent()-method, 539, 559
- triggerable paintComponent()-method, 528, 531, 536, 539, 551, 559, 576, 595, 596
- triggerable paintComponent()-method (fixes view), 540
- triggerable process...()-method, 287
- triggerable processEvent()-method, 591, 593
- triggerable read()-method, 306
- triggerable service()-method, 407
- triggerable start()-method, 398
- triggerable stop()-method, 398
- triggerable validate()-method, 598
- trim()-method (java.lang.String), 257, 387, 394
- TRUST, 478
- try-catch clause, 113, 279
- try-catch clause (branching), 112
- try-catch clause, switchboarding, 282
- try-statement (messaging with Throwable-objects), 282
- try-statement and break-statement, 282
- tunneling (data in network), 346
- tutoring, problem-, 512
- tutoring, tool-, 512
- Type, 73
- type, 35, 40, 71, 74, 75, 521
- type and protocol, 347
- type safety, 450
- type, array-, 143, 234
- type, class-, 143, 146
- type, content-, 368, 405
- type, interface-, 143, 185
- type, local-, 124
- type, MIME-, 368, 389, 393
- type, primitive-, 83, 87, 142, 191
- type, programmer-defined-, 131, 140, 142
- Type, Reference-, 143
- type, reference-, 143, 191
- type-cast potential, testing-, 253
- type-cast, implicit- (example), 544, 580
- Type-Casting, 88
- type-casting, 257, 450
- type-casting identifiers to an abstract class-type, 552
- type-casting identifiers to interface-type, 194, 404
- type-casting of primitive-types, 100
- type-casting of reference-type, 169
- type-casting, explicit-, 100
- type-casting, implicit-, 86, 87, 100, 106, 169, 194, 253
- typed, dynamically-typed language, 75
- typed, run-time typed language, 75
- typed, statically-typed language, 75
- UCS (Unified Coding Standard), 265
- UDP (User Datagram Protocol), 354, 363, 378, 384
- UDP (User Datagram Protocol) and RMI, 415
- un-overriding from OUTSIDE a class with super-keyword, 175, 192, 286
- un-overriding from WITHIN a class with super-keyword, 174
- underflow/overflow warning with floating-point operations: NONE, 86
- underflow/overflow warning with integers: NONE, 86
- unicast as one-way-flow, 343
- unicast event-messaging, 287, 289, 633
- Unicode Transformation Format (UTF), 20, 265, 308
- Unicode-format, 34, 265
- Unified Coding Standard (UCS), 265
- Uniform Resource IDENTIFIER (URI), 365
- Uniform Resource Locator (URL), 365
- Uniform Resource Name (URN), 365
- UnknownError (java.lang), 281
- unnamed package, 241
- unreachable statement, 106
- UnresolvedPermission (java.security) in footnote, 467
- unshadowing a static variable, 173
- unshadowing an instance-variable, 172
- update()-method (java.awt.Component), 552, 595
- update()-method (java.security.Signature), 493
- update()-method (java.util.Observer), 285, 286, 330
- update()-method (java.util.Observer) abstract, 285
- Uploading files (client TO server), 371
- uploading files (client TO server), 372, 374
- upper-case letter, 34, 102
- upper-case letter, in integer-literal of type long, 78
- upper-case letters in jar-file manifest, 491
- upper-case letters, keyboard signals for-, 572
- URI (Uniform Resource IDENTIFIER), 365
- URL (java.net), 388, 390–392, 405, 439
- URL (Uniform Resource Locator), 365

- URL-encoding, 375
- URL-Rewriting, 413
- URLClassLoader (java.net), 438, 469, 629
- URLConnection (java.net), 389, 392
- URLStreamHandler (java.net), 391, 392
- URLStreamHandlerFactory (java.net), 391
- URN (Uniform Resource Name), 365
- user (key-management and signing files), 488
- user (using a security-manager), 465
- user and applet-configuration, 475
- user and jar-archive, 326
- user and means of interaction, 56
- user and operating-system, 43
- user and security, 448, 453, 457
- user and tools, 54
- user calling Java's main()-method, 127
- User Datagram Protocol (UDP), 354, 363, 378, 384
- User Datagram Protocol (UDP) and RMI, 415
- user interface, 507
- user interface, terminal-based-, 507
- user interface, text-based-, 507
- user interfaces, 507
- user, cannot change a security-manager, 463
- user, selecting policies, 460
- users and applications, 38
- UTF (Unicode Transformation Format), 20, 265, 308
- utility-programmer, 27

- validate()-method, triggerable-, 598
- value, passing arguments by-value, 120, 225, 256
- valueOf()-method (java.lang.Boolean, Byte, ...), 256
- valueOf()-method (java.lang.Integer), 642
- valueOf()-method (java.lang.String), 257
- Variable, 40, 74
- variable and member, 148
- variable and object and instance, 149
- variable and state (in object-orientation), 133
- variable of primitive-type, casting of-, 88
- variable of primitive-type, copying/cloning, 83
- variable of primitive-type, type-casting of-, 100
- variable of primitive-type, wrapped-, 256
- variable of reference-type, 143
- variable of reference-type, casting of-, 169
- variable of reference-type, copying/cloning, 144
- variable of String-class, 145
- variable or field, 148
- variable, array-, 234
- variable, assignment by-value, 83
- variable, class- (static variable), 138, 148, 154, 182
- variable, CLASSPATH- (operating-system), 61, 66, 243, 254, 436, 451, 629
- variable, data-equality with primitive-type, 162
- variable, dynamically typed-, 75
- variable, field-, 155
- variable, final local-, 122
- variable, final- (quasi-constant), 182
- variable, instance-, 139, 148, 182
- Variable, Local-, 121
- variable, local- (and field), 148
- variable, member-, 148
- variable, model- (MVC-concept), 516
- variable, non-local-, 123
- variable, non-local- (definition), 117
- variable, overridden?, 139
- variable, passed by-reference to a method, 120
- variable, passed by-value to a method, 120
- variable, PATH- (operating-system), 60, 409
- variable, pointer-equality with reference-type, 162
- variable, shadowing a static-, 168
- variable, static- (class-variable), 138, 148, 154, 182
- variable, static-, in an abstract class, 184
- variable, statically typed-, 75
- variable-declaration, 147
- variable-declaration, local-, 122
- Variable-Name, 40, 74
- variable-name, 73, 83, 119
- variable-name, ambiguous-, 187, 239
- variable-name, representing a reference, 145
- variables and properties, 312
- variables and streams, 292
- variables, implicitly final in interfaces, 186
- variables, shadowing-, 123, 126, 167, 168
- Vector (java.util), 238, 445
- Vector (java.util), as buffer, 586
- Vector (java.util), as package-member, 250
- Vector (java.util), as type of container-object, 138, 236
- Vector (java.util), example, 219, 634
- Vector (java.util), example, as buffer, 228
- verifier, byte-code-, 281, 441, 450, 451
- VetoableChangeListener (java.beans), 316, 331
- VetoableChangeSupport (java.beans), 316, 331
- View and MVC, 515
- virtual channel (in network), 346
- virtual class (non-instantiable class), 138
- Virtual Machine, 40
- virtual machine, 39
- virtual network, 343, 357
- virtual platform, 17, 36, 51, 55, 151, 333
- virtual-network-protocol, 348
- VirtualMachineError (java.lang), 281, 444, 445
- Visible (state of screen-component), 598
- visible (state of screen-component), 541, 598
- visual programming (with software-components), 311, 313
- Void (java.lang), 117, 257
- volatile field (uncached), 148, 183
- volatile modifier (for fields), 183

- wait()-method (java.lang.Object), 211, 213, 216, 221, 223, 232, 252
- waitFor()-method (java.lang.Process), 255
- weak references, concept of-, 445
- Web-Application, 406
- web-applications and servlets, 414
- web-server (see server), 407
- while-statement (a pre-post-condition loop), 110
- while-statement (usually a post-condition loop), 108

- while-statement, labeled- (a pre-post-condition loop), 111
- WHOIS, 362, 364, 382
- widening cast, 169
- widening cast and polymorphism, 195
- widening of the identifier's type, 170, 638
- Window (java.awt), 562, 598
- Window (java.awt) heavyweight, 565
- Window (java.awt) top-level screen-component, 564, 598
- Window as source of event-objects, 578
- window of an applet, 400
- window of an applet (security-mark), 475
- window or frame (as modal screen-component), 540
- window or frame (GUI container), 512, 573
- window or frame (GUI-context), 44, 263, 509
- window or frame (size-change), 573
- window, activated-, 577
- window, concept of activated-, 509
- window, concept of top-level-, 509
- window, dialog-, 540
- window, terminal-, 44, 54, 55, 64
- window, terminal- (and appletviewer-command), 396, 431, 613, 641
- window, terminal- (and CTRL-C-command), 432
- window, terminal- (and jar-command), 326, 645
- window, terminal- (and java-command with jar-option), 327
- window, terminal- (and java-command with security-option), 462
- window, terminal- (and java-command with String-arguments), 236
- window, terminal- (and java-command with verbose-option), 250, 434
- window, terminal- (and java-command with version-option), 59, 65, 250
- window, terminal- (and java-command), 51, 65, 431, 505, 619
- window, terminal- (and java-command, triggers main()-method), 116, 127, 142
- window, terminal- (and javac-command), 65, 72, 431
- window, terminal- (and javadoc-command), 324
- window, terminal- (and javap-command), 642
- window, terminal- (and keytool-command), 489
- window, terminal- (and policytool-command), 458
- window, terminal- (and RMI-related commands), 418
- window, terminal- (and servlet-related commands), 407
- window, terminal- (and Uniform Resource Locator (URL)), 373
- window, terminal- (as single user interface), 508
- window, terminal- (its current directory), 48
- window, terminal- (Linux operating-system), 62
- window, terminal- (Mac OS X), 61
- window, terminal- (multiple- for multiple JREs), 52, 231
- window, terminal- (typical java-commands), 37
- window, terminal- (Windows operating-system), 63
- window- or frame-states (concept), 509
- window-closing-box, 581
- window-object, 263
- window-object, anonymous-, 159
- WindowEvent (java.awt.event), 573, 575, 578
- WindowListener (java.awt.event), 578, 581
- World Wide Web (WWW), 372
- wrapped variable of primitive-type, 256
- Wrapper, 169
- wrapper-class, 169, 171
- wrapper-class (Float, Double), 86, 92
- wrapper-class (GuardedObject), 474
- wrapper-class (Integer), 85
- wrapper-class (Number), 256
- wrapper-class (Void), 257
- wrapper-stream, 292
- wrapping and unwrapping, 256
- write()-method (java.io.FilterOutputStream, java.io.FilterWriter), 302
- write()-method (java.io.OutputStream, java.io.Writer), 127, 295
- write()-method (java.io.OutputStream, java.io.Writer) abstract, 296
- writeObject()-method (custom-defined), 318
- writeObject()-method (java.io.ObjectOutputStream), 308, 309
- Writer (java.io), 304, 310
- Writer (java.io) abstract, 296
- www (as subdomain), 360, 374
- WWW (World Wide Web), 372
- XML (eXtensible Markup Language), 410, 414
- XOR-operator (^, eXclusive OR), 84
- yield()-method (java.lang.Thread), 204, 211–213, 232
- ZIP, 326
- ZIP-compression, 326, 522
- ZIP-file, 240
- ZIP-file (and applet), 403, 614
- ZIP-file (and directory-path), 247
- ZIP-file (in Java API), 250
- ZipFile (java.util.zip), 250