

Java Grundlagen

Einführung in die Java-Programmierung

Gesamtinhaltsverzeichnis

1	Geschichte und Charakteristik	1-3
1.1	Geschichte von Java	1-3
1.1.1	Historie.....	1-5
1.2	Die Java-Entwicklungsumgebung.....	1-6
1.3	Charakteristika von Java	1-9
1.4	Die Java Virtual Machine	1-11
1.4.1	Bytecode.....	1-13
1.4.2	Weitere Bestandteile der Virtuellen Maschine	1-14
1.4.3	Security Manager.....	1-15
1.4.4	Sicherheitsmechanismen.....	1-15
1.4.5	ClassLoader.....	1-15
1.4.6	Dynamisches Laden der Klassen und Klassenpfad	1-16
1.4.7	Optionen beim Start der Virtuellen Maschine.....	1-17
1.5	Der Java-Compiler.....	1-18
1.6	Programme der Java Laufzeitumgebung.....	1-20
1.7	Komponenten der Java Standard Edition	1-20
2	Grundlagen der Java-Programmierung	2-3
2.1	Der Java-Quellcode.....	2-3
2.1.1	Namenssyntax	2-3
2.1.2	Kommentare	2-4
2.2	Ausgaben	2-5
2.2.1	System.out.print(Ausdruck).....	2-5
2.2.2	System.out.println(Ausdruck)	2-5
2.2.3	System.out.printf(Format, Ausdruck)	2-5
2.2.4	Ausgabebeispiele	2-6
2.3	Einfache Datentypen	2-7
2.3.1	Nichtnumerische Typen	2-7
2.3.2	Ganzzahltypen.....	2-7
2.3.3	Gleitpunkttypen.....	2-8
2.4	Deklaration und Definition von Variablen.....	2-9
2.4.1	Finale Variablen.....	2-10
2.5	Konstanten	2-11
2.5.1	Boolesche Konstanten	2-11
2.5.2	Numerische Konstanten.....	2-11

2.5.3	Zeichenkonstanten	2-13
2.5.4	Stringkonstanten	2-13
3	Operatoren und Anweisungen	3-3
3.1	Operatoren	3-3
3.1.1	Zuweisungsoperator	3-4
3.1.2	Arithmetische Operatoren	3-4
3.1.3	Vergleichsoperatoren	3-6
3.1.4	Logische Operatoren	3-7
3.1.5	Explizite Typumwandlung mit dem Cast-Operator	3-7
3.1.6	Bedingte Bewertung	3-8
3.1.7	new Operator	3-9
3.1.8	instanceof Operator	3-9
3.2	Liste und Hierarchie Operatoren	3-10
3.3	Anweisungen	3-11
3.3.1	if-Anweisung	3-11
3.3.2	Case-Struktur switch	3-13
3.4	Schleifen	3-15
3.4.1	while-Schleife	3-15
3.4.2	do...while-Schleife	3-16
3.4.3	for-Schleife	3-17
3.4.4	Sprunganweisungen break und continue	3-18
3.4.5	for-each-Schleife	3-20
3.4.6	return-Anweisung	3-21
3.5	Komplexer Datentyp Array	3-22
3.5.1	Arrays	3-22
3.5.2	Mehrdimensionale Arrays	3-24
4	Objektorientierte Programmentwicklung	4-3
4.1	Prozedurale versus objektorientierte Programmentwicklung	4-3
4.2	Objektorientierter Ansatz	4-4
4.2.1	Klassifizieren	4-5
4.2.2	Abstrahieren	4-6
4.2.3	Ordnen, Bilden von Hierarchien	4-6
4.3	Objekt	4-7
4.4	Klassen	4-9
4.4.1	Entwurf einer Klasse	4-9
4.4.2	UML-Notation	4-10
4.4.3	Zugriffsrechte und Deklarationen	4-12
4.4.4	Klassendefinition	4-13
4.5	Attribute	4-14

4.5.1	Attributdeklaration	4-14
4.6	Methoden (Funktionalitäten)	4-15
4.6.1	Methodendeklaration	4-15
4.6.2	Überladen von Methoden.....	4-16
4.6.3	Die main-Methode.....	4-17
4.6.4	Die Methode println()	4-18
4.7	Referenzen und Instanz Erzeugung	4-19
4.7.1	Zugriff auf Attribute und Methoden.....	4-20
4.7.2	Die this-Referenz	4-20
4.7.3	Formen von Objektreferenzen	4-22
4.7.4	Referenzen im Speicher der Virtuellen Maschine	4-23
4.8	Konstruktoren	4-24
4.8.1	Konstruktordeklaration	4-25
4.8.2	Überladen von Konstruktoren	4-27
4.8.3	Private Konstruktoren	4-29
4.9	Klassenattribute und Klassenmethoden	4-30
4.9.1	Klassenattribute	4-30
4.9.2	Klassenmethoden	4-31
4.9.3	Initialisierungen von Klassenattributen.....	4-32
4.9.4	Statische Elemente und die Virtuelle Maschine	4-33
4.11	Ausnahmen	4-34
4.11.1	Traditionelle Fehlerbehandlung	4-34
4.11.2	Die Klasse Exception	4-36
4.11.3	Beispiel Kehrwert	4-38
4.11.4	Ausnahmen mit fehlerbeschreibendem Text.....	4-39
5	Beziehungen.....	5-3
5.1	Assoziation	5-3
5.2	Aggregation / Komposition	5-4
5.3	Vererbung.....	5-6
5.3.1	Sichtbarkeit in der Vererbungshierarchie	5-9
5.3.2	Vererbung und Konstruktoren.....	5-9
5.3.3	Überschreiben von Attributen	5-10
5.3.4	Überschreiben von Methoden.....	5-11
5.3.5	Polymorphie.....	5-12
5.3.6	Vererbung und der Cast-Operator	5-13
5.3.7	Hierarchie der Exception-Klassen.....	5-15
5.3.8	Die RuntimeException	5-16
5.3.9	Eigene Exception-Klassen	5-16
5.4	Finale Elemente.....	5-17

6	Abstrakte Klassen, Interfaces und Pakete.....	6-3
6.1	Abstrakte Klassen.....	6-3
6.2	Mehrfachvererbung	6-5
6.3	Interfaces (Schnittstellen)	6-5
6.3.1	Allgemeines	6-5
6.3.2	Funktionales Interface.....	6-7
6.3.3	Interfaces und Mehrfachvererbung	6-8
6.4	Zusammenfassung der Deklarationen.....	6-9
6.4.1	Deklaration einer Klasse	6-9
6.4.2	Deklaration von Attributen.....	6-9
6.4.3	Deklaration von Methoden	6-10
6.4.4	Deklaration von Konstruktoren.....	6-10
6.5	Pakete	6-11
6.5.1	Eigene Pakete erstellen	6-12
6.5.2	Import von Klassen	6-13
6.5.3	Statische Importe	6-14
6.6	Zusammenfassung der Zugriffsrechte	6-16
7	Weiterführende Themen	7-3
7.1	Singleton	7-3
7.2	Assertions.....	7-5
7.3	Wrapper-Klassen.....	7-7
7.3.1	Autoboxing/Unboxing.....	7-8
7.4	Enumeration	7-9
7.5	Variable Argumentlisten	7-11
7.6	Metadaten	7-12
7.6.1	Vordefinierte Annotations und Meta-Annotations.....	7-13
7.6.2	Deklaration eigener Metadaten und Tool apt.....	7-14
8	Klassenbibliothek	8-3
8.1	Die Klassenbibliothek	8-3
8.2	Zeichenketten	8-4
8.2.1	Klasse String.....	8-4
8.2.2	Klasse StringBuilder und Klasse StringBuffer	8-7
8.3	Klasse Object	8-9
8.3.1	Die equals()-Methode	8-9
8.3.2	Die toString()-Methode.....	8-11
8.3.3	Die clone()-Methode	8-11
8.4	Einige Klassen des Pakets java.util	8-13
8.5	System Properties	8-14
8.6	Formatierte Ausgaben	8-15

8.7	Eine weitere Auswahl aus der Klassenbibliothek.....	8-16
8.8	Online-Dokumentation	8-17
9	Literatur	9-3
9.1	Literatur und Webseiten	9-3
9.1.1	Allgemeine Literatur zu Java Grundlagen	9-3
9.1.2	Java im Netz	9-3
9.2	UML – Unified Modeling Language	9-4
9.2.1	UML-Literatur	9-4
9.2.2	Anwendungsfalldiagramm.....	9-5
9.2.3	Klassendiagramm	9-5
9.2.4	Paketdiagramm.....	9-6
9.2.5	Sequenzdiagramm.....	9-6
10	Anhang.....	10-3
10.1	Java Schlüsselwörter.....	10-3
10.2	Namenskonventionen.....	10-3
10.3	Das Tool javadoc.....	10-4
10.4	Entwicklungstool Eclipse	10-6
10.4.1	Download Eclipse	10-6
10.4.2	Erzeugen eines ersten Java Projektes.....	10-6
10.5	Glossar	10-7
	Gesamtindex.....	IDX-1

1

Geschichte und Charakteristik

1.1	Geschichte von Java	1-3
1.1.1	Historie.....	1-5
1.2	Die Java-Entwicklungsumgebung.....	1-6
1.3	Charakteristika von Java	1-9
1.4	Die Java Virtual Machine	1-11
1.4.1	Bytecode.....	1-13
1.4.2	Weitere Bestandteile der Virtuellen Maschine	1-14
1.4.3	Security Manager.....	1-15
1.4.4	Sicherheitsmechanismen.....	1-15
1.4.5	ClassLoader.....	1-15
1.4.6	Dynamisches Laden der Klassen und Klassenpfad.....	1-16
1.4.7	Optionen beim Start der Virtuellen Maschine.....	1-17
1.5	Der Java-Compiler.....	1-18
1.6	Programme der Java Laufzeitumgebung.....	1-20
1.7	Komponenten der Java Standard Edition	1-20

1 Geschichte und Charakteristik

1.1 Geschichte von Java

Anfang der 1990er Jahre suchte Bill Joy, ein Mitbegründer der Firma Sun und Entwickler des vi-Editors und der C-Shell (UNIX), einen Programmierer, der in der Lage war, neue Programmierkonzepte für die Steuerung von Haushaltsgeräten zu entwerfen. Die laufenden Programme und Programmiersprachen mit ihren tausenden Zeilen messenden Code waren für diese Anwendungen einfach viel zu groß. In James Gosling, auch bei Sun tätig, fand er einen geeigneten Partner, dessen Gedanken sich in ähnlicher Richtung bewegten. Man erkannte bald, dass vorhandene Programmiersprachen wie C++ für Steuerungszwecke ungeeignet waren, und man begann, eine plattformneutrale Sprache zu entwickeln, die einige Konstrukte der Sprache C++ übernehmen sollte, aber ansonsten völlig neu war. In diesem Stadium wurde diese neue Sprache Oak genannt, da vor dem Zimmer von Gosling mehrere Eichen wuchsen. Im weiteren Verlauf der Entwicklung änderte sich der Name zu Java, einer in Amerika beliebten Bezeichnung für Kaffee.

Anfang 1994 begann der Boom des Internets infolge des WWW, und man erkannte bei Sun schnell, dass diese neuentwickelte Sprache sich hervorragend für das WEB eignet. Es wurde der erste Java-interpretierende Browser namens Webrunner entwickelt. Wegen Lizenzproblemen musste dieser Browser in HotJava umbenannt werden. Die ersten Versionen des Webrunners basierten noch auf C++, alle Nachfolger basierten auf dem neuentwickelten Java-Compiler. Der erste Java-Compiler wurde als Public Domain Software im Internet zur Verfügung gestellt. Auch heute ist das gesamte JDK (Java Development Kit) samt Quellcode und Dokumentation frei erhältlich.

Anfangs, als "C++ light" belächelt und bestenfalls als geeignet für einfache Eingabedialoge auf Internet-Seiten betrachtet, hat sich Java rasant weiterentwickelt. Mittlerweile kann kein professioneller Projektmanager bei der Auswahl der Programmiersprache Java ignorieren.

Java ist heute eine der bekanntesten Marken der Computerbranche und eine der am häufigsten bereitgestellten Technologien. Mit der Übernahme von Sun durch Oracle im Jahr 2009 wurde weiterhin mit Innovationen und Investitionen in die Java-Technologie fortgefahren.

Java hat sich aufgrund des wohldurchdachten Designs, der zukunftssicheren Spezifikation und der einfachen Syntax zu einer der populärsten Sprachen überhaupt gemausert, und das in beinahe allen Bereichen der Programmierung:




- Java ist aufgrund seines architekturneutralen Designs und seines überlegenen Sicherheitskonzeptes im Internet die Programmiersprache der Wahl.
- Moderne clientseitige Anwendungsprogramme werden mit Hilfe einer betriebssystemunabhängigen API flexibel und plattformunabhängig erzeugt. Die Installation des Client-Programms kann administrativ sehr einfach erfolgen, z. B. unter Verwendung der traditionellen Applets oder der neueren WebStart-Technologie.
- Der Zugriff auf Datenbanken erfolgt ebenfalls plattformunabhängig über eigene Treiber. Sogar als DBMS-interne Sprache wird Java mittlerweile verwendet.
- Verteilte Anwendungen werden mit Java-RMI und CORBA (der Java-IDL) realisiert.
- Servlets und JavaServer Pages (JSP) lassen sich einfach im Webserver nutzen, um dynamische Präsentationen im Internet zu ermöglichen.
- Java erobert den Server: Die Java Enterprise Edition (Java EE) bietet ein einheitliches architekturneutrales Framework zur Erstellung geschäftskritischer Prozesse. Darin sind mächtige APIs integriert, die neben den eben erwähnten Webkomponenten und den sogenannten Enterprise Java Beans weitere nützliche Elemente wie den Messaging Service, Mail, Benutzerauthentifizierung und -authorisierung, Transaktionsmanagement und ein API für den Zugriff auf Naming&Directory Services enthalten.
- Java steht auch für Kleingeräte zur Verfügung (Java Micro Edition, JINI).

Das alles ist natürlich nur eine Auswahl, die weder vollständig noch aktuell sein kann. Neueste Informationen sind jedoch im Internet zu finden: <http://www.oracle.com/technetwork/java>



1.1.1 Historie

Eine Aufstellung der Historie von Java ist der folgenden Liste zu entnehmen:

Geschichte und Charakteristik



Historie



- 1995 JDK 1.0-alpha: kostenlos über das Internet verfügbar
- 1996 JDK 1.0: in Netscape Navigator
- 1997 JDK 1.1: verbesserte APIs (JavaBeans, Events, JDBC, JNI, RMI)
- 1998 JDK 1.2: neue APIs (Collections, Swing, CORBA u. a.), Innere Klassen Performance-Verbesserungen
- 2000 JDK 1.3: Aufteilung in Standard, Enterprise und Micro Edition
- 2001 JDK 1.4: neue APIs (Logging, XML u. a.)
- 2004 JDK 5.0: neue Sprachelemente, generische Datentypen, Autoboxing, variable Parameterliste, neue APIs
- 2006 JDK 6: neue Sprachelemente und APIs, XML und Web-Services Performanceverbesserung
- 2011 JDK 7: neue APIs, Sprachverbesserungen und Spracherweiterungen
- 2014 JDK 8: Default-Implementierungen in Interfaces, Lambda Ausdrücke, neue Date&Time API ...
- 2017 JDK 9: Modul-Konzept, Erweiterungen der Stream-API

Abb. 1-1: Historie

1.2 Die Java-Entwicklungsumgebung

Java-Quellcode wird von einem Compiler in den Bytecode übersetzt. Ein Compiler ist in dem kostenlos erhältlichen

"Java Development Kit" (JDK) enthalten. Es gibt eine ganze Reihe von weiteren Java-Entwicklungsumgebungen. Diese reichen von einfachen Editoren mit Syntaxhervorhebung über professionelle Werkzeuge mit visuellen Werkzeugen bis hin zu komplexen Komplettlösungen mit integrierter Mehrbenutzerverwaltung, Versionsmanagement, Case-Tools etc.

The screenshot shows a presentation slide with a navigation bar at the top containing the text 'Geschichte und Charakteristik' and navigation icons. The main content area is titled 'Java SE Development Kit' in red. It contains a bulleted list of information about the Java SE Development Kit.

- JRE (Java Runtime Environment) = Laufzeitumgebung
- JDK (Java Development Kit) = Software Entwicklungs-Paket
- JDK = JRE + Compiler + Tools
- Seit Version 1.2 Aufteilung in drei Plattformen
 - Java Standard-Edition (**Java SE**, J2SE)
 - Java Enterprise Edition (**Java EE**, J2EE)
 - Serverseitige Komponenten, setzt auf Java SE auf
 - Java Micro Edition (**Java ME**, J2ME), Wireless Toolkit
 - für Kleingeräte
- In diesem Kurs nur Java SE
- Kostenloser Download von Oracle:
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Abb. 1-2: JDK

All diesen Entwicklungsumgebungen ist jedoch gemeinsam, dass die von Sun definierte und frei zur Verfügung gestellte Standard-Klassenbibliothek verwendet wird. Seit der Version 1.2 sind die folgenden sogenannten Editionen unter dem gemeinsamen Produktnamen der Java 2 Plattform erhältlich:

- Standard-Edition (Java SE oder J2SE)
- Enterprise Edition (Java EE oder J2EE)
- Micro Edition (Java ME oder J2ME)

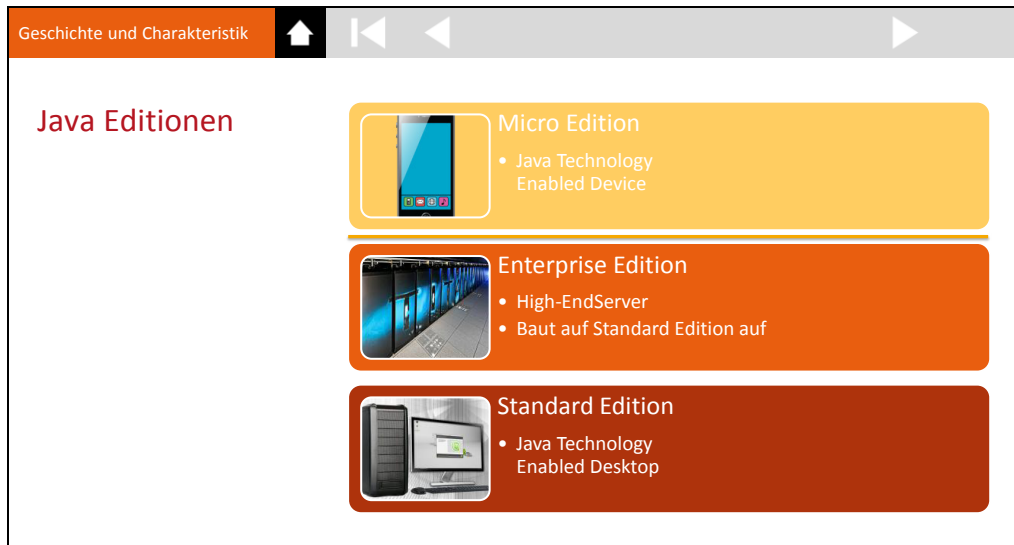


Abb. 1-3: Java Editionen

Die Standard-Edition enthält neben der kompletten Laufzeitumgebung den Compiler, einen Debugger sowie diverse Hilfsprogramme.

Die Enterprise-Edition bietet APIs für die serverseitige Verwendung von Java (insbesondere die Servlet-Technologie, EnterpriseJavaBeans, Java Message Service) und eine Referenz-Implementierung eines Application Server.


Die Micro-Edition definiert die Umgebung für den Einsatz von Java auf Systemen mit kleinem Speicher und beschränkter Prozessorkapazität.

Sowohl die Enterprise- als auch die Micro-Edition definieren keine neue Programmiersprache. An der Spezifikation des Bytecodes ändert sich nichts, so dass ohne Beschränkung der Allgemeinheit der gleiche Compiler verwendet werden kann.

Folgende Entwicklungsumgebungen unterstützen u. a. die Java SE:

Geschichte und Charakteristik

Entwicklungs- umgebungen



- Eclipse von der Eclipse Foundation
- NetBeans von Oracle Corporation
- BlueJ von der BlueJ Group (University of Kent)
- IntelliJ IDEA von JetBrains

u. v. a. m.


Abb. 1-4: Entwicklungsumgebungen

1.3 Charakteristika von Java

Java ist eine objektorientierte Programmiersprache, entwickelt vom Informationstechnologieunternehmen Sun (heute Oracle-Gruppe).

Geschichte und Charakteristik
⏮ ⏪ ⏩ ⏭

Charakteristika von



- Java ist eine einfache Programmiersprache
- Java hat einen überschaubaren Sprachumfang
- Java ist eine mit C und C++ verwandte Sprache
- Java ist eine Interpretersprache
- Java ist eine architekturneutrale Sprache
- Java ist eine rein objektorientierte Sprache
- Java erlaubt keine direkten Zugriffe auf Betriebssystemressourcen
- Java lässt robuste, stabile Programme entstehen
- Java ist eine streng typisierte Programmiersprache
- Java hat die Behandlung von Ausnahmen in Form von Exception-Klassen implementiert
- Java unterstützt nebenläufige Teilprozesse und verteilte Anwendungen

Abb. 1-5: Charakteristika

- Java hat einen überschaubaren Sprachumfang und ist deshalb eine recht einfache Programmiersprache.
- Java ist eine mit C und C++ verwandte Sprache:

Java lehnt sich syntaktisch an populäre Sprachen wie C und C++ an und bietet erfahrenen ProgrammiererInnen vertraute Konstrukte an. Im Vergleich zu C++ sind einige Sprachkonstrukte hinzugefügt worden, aber auch manches entfernt, bzw. mit neuen Bedeutungen versehen. Es besteht also eine Verwandtschaft, es handelt sich nicht um eine Weiterentwicklung der Sprache C++.

- Java ist eine Interpretersprache:

Der Java-Compiler `javac` erzeugt keinen Maschinencode, sondern einen Bytecode. Dieser Bytecode wird dann mit der Virtuellen Maschine (VM) `java` und nicht direkt auf Betriebssystemebene ausgeführt. Dies bedingt, dass Java-Programme meistens minimal langsamer als C oder C++ Programme sind (bei den ältesten Java-Versionen war das noch etwa ein Faktor 20, heute ist dies wesentlich besser). Zeitkritische Methoden können zudem als native C/C++ Funktionen ausgelagert und aus dem Java-Programm heraus aufgerufen werden.

- Java ist eine architekturneutrale Sprache:
- Durch die Interpretation des Bytecodes durch die VM wird die Portabilität und Hardwareunabhängigkeit der Sprache Java erreicht. Die VM ist jeweils so entworfen, dass jedes Java-Programm dieselbe Umgebung vorfindet. Compiler und Linker der gängigen Programmiersprachen werden zwar auch für verschiedene Hardwareplattformen entwickelt, der ausführbare Code ist dann aber nur auf einem bestimmten Betriebssystem ausführbar.
- Java ist eine rein objektorientierte Sprache:

Jede Funktionalität muss innerhalb einer Klasse definiert werden, somit gibt es keine globalen Funktionen. In Java wird der Programmierer in hohem Maße dazu motiviert, objektorientiert zu programmieren. Ebenso gibt es keine Möglichkeit, globale Variablen (außerhalb einer Klasse) zu definieren. Das hat allerdings relativ hohen Programmieraufwand, selbst bei einfachen Funktionalitäten, zur Folge.
- Java erlaubt keine direkten Zugriffe auf Betriebssystemressourcen:

Es gibt keine Pointer und somit keine Möglichkeit, direkt den Hauptspeicher des Rechners anzusprechen.

Direkte Aufrufe von Betriebssystemfunktionalitäten sind ebenfalls nicht möglich, können aber durch Konfiguration oder indirekt durch die Verwendung spezifizierter Bibliotheken des Java-Interpreters erfolgen.
- Java lässt robuste, stabile Programme entstehen:

Java enthält eine automatische Speicherbereinigung (Garbage Collection), die das Entstehen von Speicherlecks vermeiden hilft, allerdings die Performance verschlechtert.

Wie oben schon erwähnt, gibt es in Java keine Möglichkeit, direkt den Speicher des Rechners anzusprechen und darin Manipulationen vorzunehmen. Dadurch wird einerseits eine ganze Gruppe von Fehlermöglichkeiten von vornherein eliminiert. Das steigert die Stabilität der Anwendungen. Andererseits werden dadurch performacesteigernde Möglichkeiten wie Pointeranwendungen verhindert.
- Java ist eine streng typisierte Programmiersprache:

Jeder Ausdruck, jede Variable und jedes Objekt ist eindeutig einem Datentyp zugeordnet. Dadurch kann bereits der Compiler frühzeitig Prüfungen vornehmen und Fehler erkennen. Es ergibt sich dadurch auch eine Steuerungsmöglichkeit von Funktionsaufrufen und Effekten nur über unterschiedliche Datentypen.

- Java hat die Behandlung von Ausnahmen in Form von Exception-
klassen implementiert:

Funktionen können Ausnahmen erzeugen, die erzwungenermaßen behandelt werden müssen. Findet diese Ausnahmebehandlung nicht statt, so erzeugt bereits der Compiler eine Fehlermeldung. Auch für die Fehlerbehandlung ist somit eine saubere Programmstruktur vorgegeben.

- Java unterstützt nebenläufige Teilprozesse und verteilte Anwendungen:

Java bietet einfache Möglichkeiten, nebenläufige Teilprozesse, sogenannte Threads, innerhalb des Hauptprozesses zu erzeugen. Ebenso unterstützt Java verteilte Anwendungen.

1.4 Die Java Virtual Machine

Java-Programme müssen durch den Anspruch der Plattformunabhängigkeit ("Write once, run anywhere") in einer eigenen betriebssystemunabhängigen Umgebung ausgeführt werden. Diese Umgebung wird von Sun spezifiziert unter der "Java Laufzeitumgebung" bzw. dem JRE (Java Runtime Environment). Zentraler Bestandteil des JRE ist die "Java Virtual Machine" (JVM).

Geschichte und Charakteristik

Die Virtuelle Maschine

- Funktionsweise spezifiziert ehemals von Sun
 - Implementiert von verschiedenen Herstellern
 - Virtuelle Maschinen sind für viele Plattformen bereits vorhanden
- Virtuelle Maschinen
 - Interpretieren Bytecode-Anweisungen
 - Bytecode ist ein prozessorunabhängiger Satz von maschinennahen Befehlen
 - Kontrollieren den Programmlauf auf Fehler
 - Haben eine automatische Speicherbereinigung
 - Zahlen und die Gleitkomma-Arithmetik sind plattformunabhängig gemäß dem IEEE Standard definiert
 - Einbindung externer Bibliotheken durch "Java Native Interface" möglich
 - Integrierte Ausnahmebehandlung vorhanden
 - Das Laden benötigter Programmteile ist dynamisch zur Laufzeit möglich

Abb. 1-6: VM

Diese Virtuelle Maschine muss für die Plattformen zur Verfügung gestellt werden, die Java unterstützen wollen. Momentan sind dies die gängigen PC-Plattformen (Macintosh, Windows etc.), und Serverplattformen (Solaris, Unix, Linux). Java ist aber auch auf dem Großrechner und innerhalb von Datenbanken präsent und wird konsequent auch auf dem Kleingerätemarkt (Organizer, Mobiltelefone) angeboten.

Prinzip der plattformunabhängigen Programmierung mit Bytecode und Virtueller Maschine:

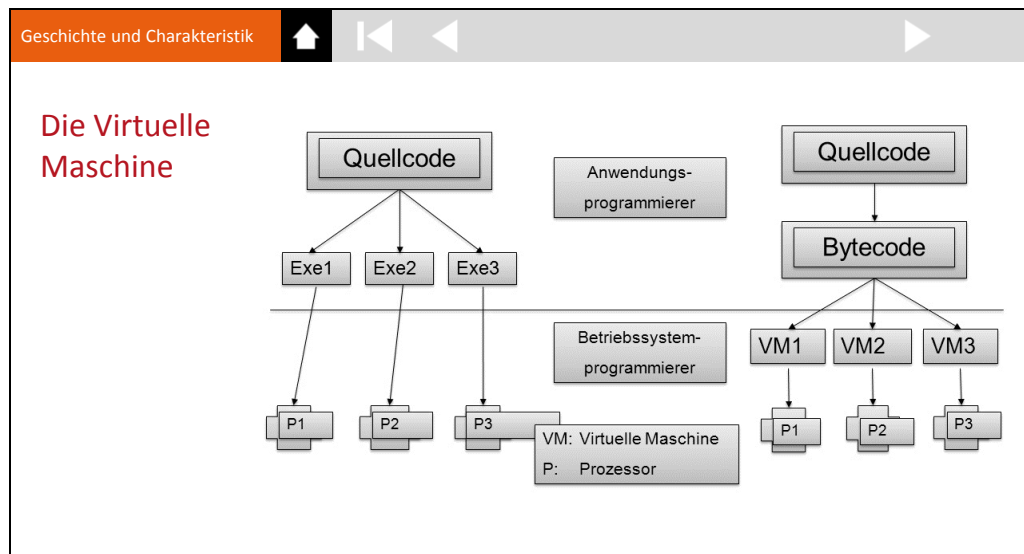


Abb. 1-7: VM

Links: Traditionelle Art der Programmierung: Durch Übersetzen mit einem entsprechenden Compiler und anschließendes Linken muss für jeden Prozessor eine eigene ausführbare Binärdatei erzeugt werden. Der Quellcode kann durch Verzicht auf plattformabhängige Befehle plattformunabhängig gehalten werden.

Rechts: Mit Hilfe des Compilers wird der von der Anwendungsprogrammiererin entwickelte Quellcode in Bytecode übersetzt. Dieser wird von der Virtuellen Maschine interpretiert. Dadurch wird die Plattform Unabhängigkeit erreicht.

Für jede Plattformen, die Java unterstützen will, wird eine eigene Virtuelle Maschine zur Verfügung gestellt. Die konkrete Implementierung muss natürlich durch plattformabhängige Programme erfolgen.

Die Produktpalette Virtueller Maschinen ist bereits sehr umfassend. Es existieren Ausprägungen für Server, PC-Systeme und Kleingeräte, aber auch Maschinen, die zur Laufzeit die Ausführungsgeschwindigkeit der Programme optimieren können. Suns Virtuelle Maschine beinhaltet die "HotSpot"-Technologie zur Performancesteigerung. Werkzeuge zum Testen von Java-Programmen benötigen spezielle Routinen zum automatischen Sammeln der Profiling-Information zur Ausführungszeit. Anbieter von Entwicklungsumgebungen verwenden häufig ebenfalls eigene Implementierungen mit dem Fokus auf komfortables Debugging und dynamisches Neu Laden geänderter Programmteile.

1.4.1 Bytecode

Der Befehlssatz einer Virtuellen Maschine ist von Sun exakt spezifiziert worden, dadurch kann Java als "plattformunabhängig" bezeichnet werden. Die Syntax des Befehlssatzes lehnt sich an die Assembler-Sprache an, ist aber im Gegensatz dazu allgemein gültig definiert worden und kennt neben elementaren maschinennahen Befehlen auch eine Reihe komplexerer Anweisungen wie z. B. das Reservieren eines strukturierten Speicherbereichs in Form von Arrays oder Datenstrukturen. Die Anweisungen werden in Form von Bytecodes von einer Virtuellen Maschine geladen. Bytecode ist nichts anderes als ein wohlstrukturierter Block von Zeichen, der neben den Anweisungen der Programmlogik noch weitere Bereiche enthält. So werden darin auch Konstanten, Verweise auf andere Bytecode-Blöcke sowie Debug- und Versionsinformationen abgelegt. Die Ablage der Bytecode-Informationen ist beliebig, ein traditionell dateibasiertes System ist ebenso möglich wie ein Laden über Netzwerkprotokolle, eine Datenbankabfrage oder andere Wege.

The screenshot shows a presentation slide with a navigation bar at the top. The navigation bar has an orange tab labeled 'Geschichte und Charakteristik', a home icon, and navigation arrows. The slide content is titled 'Der Bytecode' in red. It features a bulleted list of characteristics and two sub-sections: 'ClassLoader:' and 'SecurityManager:'. The list includes points about platform independence, safety mechanisms, format correctness, type consistency, no access to uninitialized memory, and no direct memory access. The 'ClassLoader' section mentions distinguishing system and application classes and the location of bytecode files. The 'SecurityManager' section mentions setting a security environment and date-based configuration.

Der Bytecode

- Plattformunabhängige Interpreter-Sprache für die Virtuelle Maschine
- Ähnlichkeiten zu Assembler
- Spezifikation enthält Sicherheits-Mechanismen
 - Korrektheit des Bytecode-Formats
 - Übereinstimmung der Typen bei Zuweisungsoperationen und Aufrufen von Routinen
 - Kein Zugriff auf nicht-initialisierte Speicherbereiche oder Variablen
 - Keinerlei direkter Speicherzugriff

ClassLoader:

- Unterscheidung Systemklassen und Anwenderklassen
- Lokation der Bytecode-Dateien

SecurityManager:

- Aufsetzen einer Sicherheitsumgebung für den Anwender
- Dateibasierte Konfiguration

Abb. 1-8: Bytecode

Wie ein Hardware-Prozessor im Laufe der Programmausführung einen herstellerabhängigen Satz von elementaren Assembler-Befehlen abarbeitet, interpretiert die Virtuelle Maschine ein Java-Programm als Folge von Bytecode-Anweisungen.

Die Virtuelle Maschine ist aber wesentlich robuster als ein einfacher Prozessor, der die übergebenen Anweisungen in der Regel ohne Kontrolle ausführt. Bevor ein geladener Bytecode-Block zur Ausführung gelangt, werden vom sogenannten Bytecode-Verifier eine Reihe von Prüfroutinen ausgeführt. Diese Prüfungen beinhalten z. B.:

- Korrektheit des Bytecode-Formats
- Übereinstimmung der Typen bei Zuweisungsoperationen und Aufrufen von Unterprogrammen
- kein Zugriff auf nicht-initialisierte Speicherbereiche oder Variablen
- keinerlei direkter Speicherzugriff.

Dies stellt sicher, dass die Virtuelle Maschine weder durch Fehler beim Laden des Bytecodes noch durch willkürliche Manipulationen korrupten oder unsicheren Code ausführen kann.

1.4.2 Weitere Bestandteile der Virtuellen Maschine

Neben der Prüfung und Interpretation von Bytecode enthält eine Virtuelle Maschine noch weitere Komponenten.

- Datentypen für Zahlen und die Gleitkomma-Arithmetik sind plattformunabhängig gemäß dem IEEE Standard for Binary Floating-Point Arithmetic definiert.
- Die Einbindung externer Bibliotheken oder der Zugriff aus anderen Programmen heraus erfolgt über eine wohldefinierte Schnittstelle, das JNI (Java Native Interface).
- Bei Laufzeitfehlern oder über Programmcode signalisierte Ausnahmesituationen in einem aufgerufenen Unterprogramm wird die Hierarchie der aufrufenden Programme solange zurückverfolgt, bis eine Stelle gefunden wird, an der der Fehler behandelt werden kann. Wird keine solche Stelle gefunden, wird die Virtuelle Maschine kontrolliert beendet.
- Das Laden benötigter Programmteile ist dynamisch zur Laufzeit möglich.
- Ebenso praktisch ist der automatische Speicherbereinigungs-Mechanismus, der innerhalb der VM ständig als Hintergrundprozess abläuft, die sogenannte "Garbage Collection". Vom Programm allozierter Speicher muss nicht durch eine gesonderte Programmanweisung wieder freigegeben werden.

1.4.3 Security Manager

Jedes Java-Programm besitzt einen sogenannten "Security Manager". Dieser wird bei allen Versuchen, auf geschützte Ressourcen zuzugreifen befragt, und kann vom Programmierer so eingestellt werden, dass der Zugriff erlaubt oder verboten ist. Die Verwendung eines Security Managers ist in den Systemklassen fest implementiert, eine dateibasierte Konfigurierungsmöglichkeit ist im Rahmen der Laufzeitumgebung möglich.

1.4.4 Sicherheitsmechanismen

Bezüglich der Sicherheit der Sprache Java gibt es Sicherheitsmechanismen auf mehreren Ebenen. Eine detaillierte Besprechung kann im Rahmen dieser Einführung nicht gegeben werden, eine Übersicht darf aber nicht fehlen.

1.4.4.1 Überprüfung des Bytecodes

Direkt nach dem Laden und noch vor Ausführung eines Java-Codes wird dieser einer Reihe von Überprüfungen unterworfen und zwar unter anderem, ob

- der Code keine Zugriffseinschränkungen (z. B. Zugriff auf geschützten Speicherbereich) verletzt,
- der Code Funktionen mit den richtigen Argumenten aufruft,
- der Code keinen Stacküberlauf verursacht,
- der Code keine unzulässigen Konvertierungen vornimmt,
- die vorgenommenen Objektzugriffe legal sind.

Einige dieser Überprüfungen werden vom sogenannten "Bytecode-Verifier" vorgenommen, einem Teil des ClassLoaders.

1.4.5 ClassLoader

Die Aufgabe des ClassLoaders ist neben dem bereits vorgestellten Einladen von Klassen auch deren Zuordnung zu Namensräumen. Jede geladene Klasse ist eindeutig die Quelle, von der aus sie geladen wurde, zugeordnet. Somit sind z. B. die geladenen Systemklassen unterscheidbar von den zusätzlichen Anwenderklassen. Greift eine Klasse auf eine Variable oder Funktion einer anderen Klasse zu, wird zuerst der Namensraum der Systemklassen durchsucht und dann erst der Namensraum der eingebundenen Klassen. Damit wird beispielsweise verhindert, dass eine "feindliche" Anwendung als Systemklasse angesehen werden kann und ein fehlerhaftes Verhalten produziert.

Das Laden von Klassen über das Netzwerk ist ebenfalls im ClassLoader festgelegt. Über diese Klasse werden Regeln definiert, wie Klassen über verschiedene Protokolle über das Netzwerk geladen werden. Es wird sichergestellt, dass die geladenen Klassen in einem anderen Namensraum liegen als die lokalen Klassen. Damit wird beispielsweise verhindert, dass eine Anwendung eine Systemklasse durch eine eigene Version ersetzt. Eine weitere Möglichkeit der Codeüberprüfung von Java ist die Fähigkeit festzustellen, ob Code von außerhalb oder innerhalb einer Firewall stammt. Ferner ist es möglich, verschlüsselte Nachrichten oder öffentliche Schlüssel in Code einzubinden, welcher dann den Sender identifiziert und auch seine Integrität garantiert.

1.4.6 Dynamisches Laden der Klassen und Klassenpfad

Die Virtuelle Maschine lädt nur bestimmte Klassen direkt in den Arbeitsspeicher. Diese sogenannten "Bootstrap-Klassen" sind die Systemklassen, die in der Datei `rt.jar` im `lib`-Verzeichnis gefunden werden.

Geschichte und Charakteristik
↑
◀
▶

Der Klassenpfad

- Die Lokation der Java-Archive und der Klassen wird von drei Quellen bestimmt
 - Systemklassen: Datei `rt.jar` in `%JAVA_HOME%\jre\lib`
 - Erweiterungsklassen: Java-Archive in `%JAVA_HOME%\jre\lib\ext`
 - Anwendungsklassen
 - Pfadangaben in der Umgebungsvariablen `CLASSPATH`
 - Übergabeparameter `-cp` beim Start der Virtuellen Maschine
- Standard (wenn `CLASSPATH` nicht gesetzt):
 - `.` (Punkt) = aktuelles Working-Directory, Pakete in Unterdirectories
 - Datei `jre/lib/rt.jar` = Klassenbibliothek der verwendeten Java-Software, liegt relativ zum `bin`-Directory (`../jre/lib`)
 - alle jar-Dateien in `jre/lib/ext`
- `CLASSPATH` setzen:
 - `CLASSPATH` enthält: Directories (deren Unterdirectories die Pakete sind), jar-Dateien (die eine Hierarchie von Paketen enthalten), durch Semikolon (Windows) oder Doppelpunkt (Unix) getrennt

Abb. 1-9: Klassenpfad

Eine besondere Systemklasse ist der sogenannte *Klassenlader*. Der Klassenlader oder `ClassLoader` ist nichts anderes als eine Java-Klasse, die in der Lage ist, eine frei wählbare Anzahl von Lokationen nach Bytecode zu durchsuchen und der Virtuellen Maschine zur Verfügung zu stellen. Um die Anzahl und Lokation der zu durchsuchenden Quellen frei einstellen zu können, verwendet der Standard-Klassenlader die Umgebungsvariable `CLASSPATH`. Dieser Pfad enthält eine über Semikolon (Windows) oder Doppelpunkt (Solaris, Linux) getrennte Liste von Java-Archivdateien und Verzeichnissen.

Die Virtuelle Maschine versucht nun, eine in einem Programm verwendete Klasse dadurch zu laden, indem

- als erstes die Systemklassen durchsucht werden und, falls die angesprochene Klasse darin nicht gefunden wird,
- danach über den `ClassLoader` der Klassenpfad in der angegebenen Reihenfolge.

Wird die benötigte Klasse gefunden, wird sie einmalig in den Speicher geladen. Sonst tritt ein Laufzeitfehler auf. Der Klassenpfad kann sowohl als Umgebungsvariable gesetzt werden als auch der Virtuellen Maschine beim Programmstart als Option mitgegeben werden.

Weiterhin wird das sogenannte *Java Extensions Framework* verwendet. Optionale Klassenbibliotheken oder Anwendungen können als Java-Archive in ein bestimmtes Verzeichnis kopiert werden. Die in diesem Verzeichnis (`lib/ext`) enthaltenen Archive werden automatisch verwendet.

1.4.7 Optionen beim Start der Virtuellen Maschine

Wichtige Optionen beim Programmstart der Virtuellen Maschine sind der folgenden Tabelle zu entnehmen:

Option	Beschreibung
-verbose	Ausgabe der aktuell durchgeführten Aktion mit den drei Möglichkeiten
-classpath oder -cp<path>	Erweiterung des Klassenpfades für Anwender-Klassen
-D<name>=<value>	Setzen einer System-Property beim Programmstart

1.5 Der Java-Compiler

Der Compiler kann über die Konsole gestartet werden. Als Übergabeparameter sind ein oder mehrere Java-Quellcode-Dateien zulässig. Der generierte Bytecode hat den gleichen Namen wie die definierte Klasse, die Endung lautet `.class`.

Im Folgenden sind einige Beispiele für den Aufruf gegeben:

Kommando	Wirkung
<code>javac HelloWorld.java</code>	Sucht die Datei <code>HelloWorld.java</code> im aktuellen Verzeichnis, erzeugt die Datei <code>HelloWorld.class</code> in einem dem Paket entsprechenden Unterverzeichnis unter dem aktuellen Verzeichnis.
<code>javac *.java</code>	Alle Quellcode-Dateien des aktuellen Verzeichnisses werden übersetzt. Pro definierter Klasse wird die entsprechende <code>.class</code> -Datei erstellt.
<code>javac -d directory *.java</code>	Alle Quellcode-Dateien des aktuellen Verzeichnisses werden übersetzt. Pro definierter Klasse wird die entsprechende <code>.class</code> -Datei im gewählten Verzeichnis erstellt.

Der Java-Compiler erzeugt aus dem Java-Quellcode Java-Programme. Genauer formuliert: Aus einer im Quellcode definierten Klasse erzeugt der Compiler die Bytecode-Datei. Der Java-Compiler des JDK hat den Namen `javac` und besitzt folgende wichtige Optionen:

Option	Beschreibung
<code>-verbose</code>	Ausgabe der aktuell durchgeführten Aktion
<code>-classpath <path></code>	Erweiterung des Klassenpfades für Anwender-Klassen
<code>-d <path></code>	Umleitung der Bytecode-Ausgabe in das angegebene Verzeichnis
<code>-deprecation</code>	Ausgabe einer Warnung, falls Sprachelemente verwendet werden, die in der aktuellen Sprachversion veraltet sind, aber noch unterstützt werden
<code>-g</code>	In den Bytecode wird jede Debugging-Information mit aufgenommen
<code>-g:none</code>	In den Bytecode wird keine Debugging-Information mit aufgenommen
<code>-O</code>	Optimierung des generierten Quellcodes bezüglich Ausführungsgeschwindigkeit
<i>Auswahl wichtiger Optionen des Java-Compilers, Java SE. Vollständige Liste durch den Aufruf <code>javac</code></i>	

Durch die Einführung des Bytecodes kann die Programmierung in Java plattformunabhängig gehalten werden. Die Laufzeitumgebung enthält alle notwendigen betriebssystemabhängigen Programme, die Klassenbibliothek und alle Applikationsklassen können unabhängig gehalten werden.

1.6 Programme der Java Laufzeitumgebung

Das `bin`-Verzeichnis der Laufzeitumgebung enthält noch eine Reihe weiterer Programme, die in der folgenden tabellarischen Aufstellung beschrieben werden. Diese Auflistung dient hier als Beispiel und erhebt keinerlei Anspruch auf eine fundierte Erläuterung und Vollständigkeit!

Programm	Beschreibung
<code>javaw</code>	Identisch mit <code>java</code> , jedoch ohne assoziiertes Konsolenfenster
<code>keytool</code>	Das JRE beinhaltet eine Benutzerdatenbank. Keytool verwaltet Schlüssel und Zertifikate
<code>policytool</code>	Zuordnung von Berechtigungen: Welche Programme haben welchen Zugriff auf geschützte Ressourcen
<code>rmid</code>	Der rmi-Activation-Dämon. Ein Hilfsprogramm, mit dem Serverprogramme erst bei einer Client-Anfrage generiert werden
<code>rmiregistry</code>	Der Namensservice für entfernte Java-RMI-Objekte
<code>tnameserv</code>	Der Namensservice für entfernte CORBA-Objekte

1.7 Komponenten der Java Standard Edition

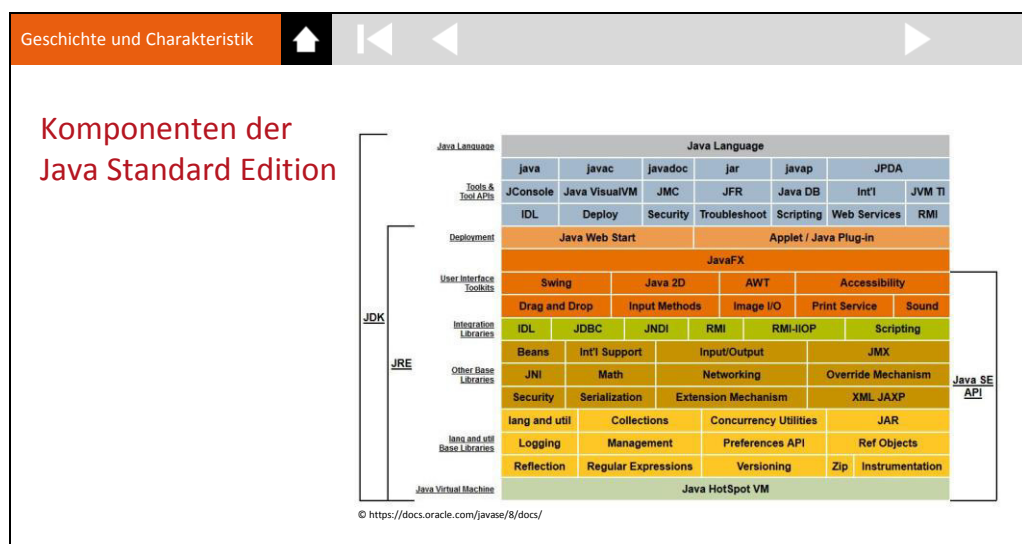


Abb. 1-10: Komponenten der JSE

2

Grundlagen der Java-Programmierung

2.1	Der Java-Quellcode	2-3
2.1.1	Namenssyntax	2-3
2.1.2	Kommentare	2-4
2.2	Ausgaben	2-5
2.2.1	System.out.print(Ausdruck)	2-5
2.2.2	System.out.println(Ausdruck)	2-5
2.2.3	System.out.printf(Format, Ausdruck)	2-5
2.2.4	Ausgabebeispiele	2-6
2.3	Einfache Datentypen	2-7
2.3.1	Nichtnumerische Typen	2-7
2.3.2	Ganzzahltypen	2-7
2.3.3	Gleitpunkttypen	2-8
2.4	Deklaration und Definition von Variablen	2-9
2.4.1	Finale Variablen	2-10
2.5	Konstanten	2-11
2.5.1	Boolesche Konstanten	2-11
2.5.2	Numerische Konstanten	2-11
2.5.3	Zeichenkonstanten	2-13
2.5.4	Stringkonstanten	2-13

2 Grundlagen der Java-Programmierung

2.1 Der Java-Quellcode

Für die Entwicklung eines Java-Programmes sind keine Entwicklungsumgebungen notwendig. Der Java-Quellcode besteht ausschließlich aus lesbaren Textdateien, die mit einem beliebigen Editor erzeugt werden können.

Ein Java-Quellcode hat die Dateiendung `.java` und enthält eine oder mehrerer Definitionen von sogenannten Klassen, die in einem objekt-orientierten Ansatz wiederum Funktionen (Methoden) und Attribute enthalten. Häufig ist aus Gründen der Übersichtlichkeit nur jeweils eine einzige Klasse in einer Quellcode-Datei enthalten. Der Name der Klasse entspricht dann dem Dateinamen ohne Endung.

2.1.1 Namenssyntax





Java unterscheidet zwischen Groß- und Kleinbuchstaben. Dies ist bei der Vergabe von Namen (Bezeichner von Klassen, Variablen, Funktionen etc.) zu berücksichtigen.

Bezeichner werden gebildet aus:

- Buchstaben
- Ziffern
- Unterstrich

Einzige Einschränkung: Die Bezeichner dürfen nicht mit einer Ziffer beginnen.

Grundlagen der Java-Programmierung



Java Quellcode



- Namenssyntax - Bezeichner werden gebildet aus:
 - Buchstaben
 - Ziffern
 - Unterstrich
- Kommentare
 - Zeilenkommentar: `//`
 - Blockkommentar: `/*`
`Block-Kommentar`
`*/`
 - Dokumentationskommentare: `/**`
`Javadoc-Kommentar`
`*/`

Abb. 2-1: Java Quellcode

2.1.2 Kommentare

Grundsätzlich ist es von Vorteil, Programme gut zu kommentieren und zu dokumentieren.

Java bietet dafür drei Arten von Kommentaren:

1. Zeilenkommentare werden eingeleitet von zwei Schrägstrichen
`// Zeilenkommentar`
Alle folgenden Zeichen dieser Zeile werden als Kommentar interpretiert.
2. Kommentare können mehrere Zeilen umfassen. Der Beginn wird durch Schrägstrich und Stern
`/*
 Blockkommentar
*/`
und das Ende durch Stern und Schrägstrich gekennzeichnet.
3. Dokumentationskommentare entsprechen der 2. Möglichkeit, werden aber am Beginn durch Schrägstrich und zwei Sterne eingeleitet
`/**
 Dokumentationskommentar
*/`
Alle Zeichen innerhalb dieser Konstruktion sind ebenfalls Kommentare, werden aber zusätzlich vom `javadoc`-Tool des JDK in die generierte Dokumentation übernommen.



Abb. 2-2: Einfaches Beispiel

2.2 Ausgaben

In Java kann man Text auf der Konsole ausgeben. Dazu stehen drei verschiedene Methoden zur Verfügung.

The screenshot shows a presentation slide with the title 'Ausgaben' in red. It lists three methods for outputting text in Java:

- ohne Zeilenumbruch
 - `System.out.print(Ausdruck)`
- mit Zeilenumbruch
 - `System.out.println(Ausdruck)`
- ohne Zeilenumbruch mit Formatierung
 - `System.out.printf(Format, Ausdruck)`

Below these methods, there are two sections: 'Formatangabe:' and 'Flags:'. The 'Formatangabe:' section lists format specifiers: %d for whole decimal numbers, %e for floating-point numbers in scientific notation, %f for floating-point numbers in fixed-point notation, and %s for strings. The 'Flags:' section lists flags: 0 for leading zeros, -: for left alignment, +: for always showing signs, and blank for leading spaces.

Abb. 2-3: Ausgaben

2.2.1 System.out.print(Ausdruck)

Ausgabe der nötigen Stellen des Ausdrucks ohne einen Zeilenvorschub auszulösen.

2.2.2 System.out.println(Ausdruck)

Ausgabe der nötigen Stellen mit einem anschließenden Zeilenvorschub. Eine besondere Rolle spielt dabei der +-Operator, er kann für verschiedenen Datentypen angewendet werden und verknüpft die Ausgaben.

2.2.3 System.out.printf(Format, Ausdruck)

Formatierte Ausgabe des Ausdrucks ohne impliziten Zeilenvorschub.. Der Formatstring enthält Formatangaben der Form %d, %f, %e, %s, usw.

Bedeutung der Formatangaben:

%d ganze Dezimalzahl
 %e Gleitkommazahl im Gleitkommaformat (mit Exponent)
 %f Gleitkommazahl im Festkommaformat (ohne Exponent)
 %s String

Beispiel: `System.out.printf("%5.2f", 12.3456);`
`System.out.printf("%6.1f\n", 12.3456);`
`System.out.printf("%10e\n", 12.3456);`


```
System.out.printf("%4d\n", 123);  
System.out.printf("%+3d\n", 123);  
System.out.printf("%+3d\n", -123);
```

```
12,35    12,3  
1,234560e+01  
123  
+123  
-123
```

2.2.4 Ausgabebeispiele

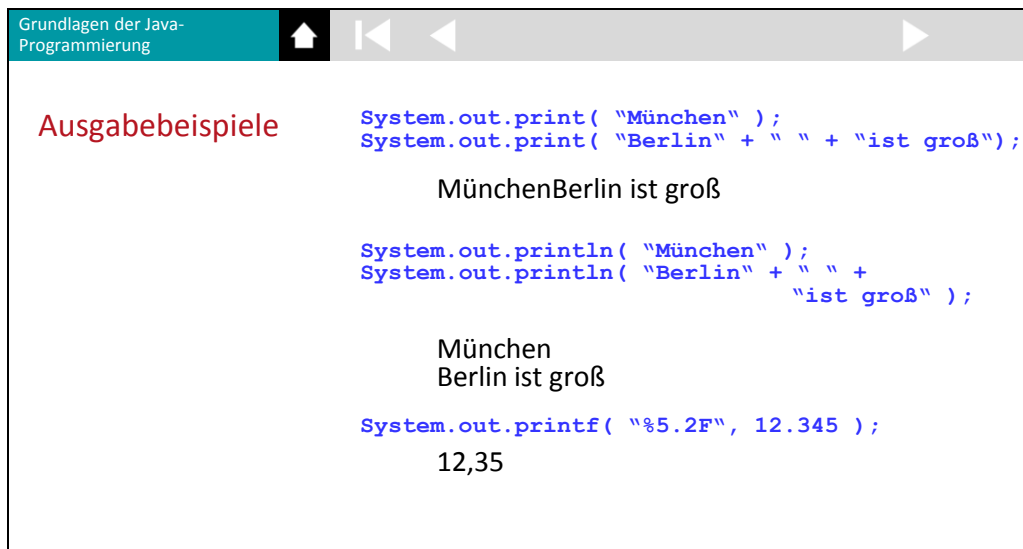


Abb. 2-4: Ausgabebeispiele

2.3 Einfache Datentypen

Java ist eine "typisierte" Programmiersprache. Das bedeutet, dass jede Variable, jeder Parameter und jede Funktion vor ihrer Verwendung mit einem Typ deklariert werden muss. Jede Zuweisung an eine Variable und jeder Rückgabewert einer Funktion wird vom Compiler auf den kompatiblen Typ geprüft. Fehlerhafte Zuweisungen oder Rückgabe Werte führen zu einem Fehler beim Übersetzen des Programms.

Grundlagen der Java-Programmierung			
Einfache Datentypen			
Typ	Größe	von	bis
■ Ganzzahltypen			
byte	1 Byte	-128	127
short	2 Byte	-32768	32767
int	4 Byte	-2.147.483.648	2.147.483.647
long	5 Byte	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
■ Gleitpunkttypen			
float	4 Byte	ungef. -3.4E+38	3.4E+38
double	8 Byte	ungef. -1.7E+308	+1.7E+308
■ Nichtnumerische Typen			
char	2 Byte		
boolean	1 Byte	true	false

Abb. 2-5: Einfache Datentypen

2.3.1 Nichtnumerische Typen

Name	Größe in Byte	Wertebereich
char	2	beliebiges Zeichen (Unicode)
boolean	1	Ein boolescher Wert (true oder false)

2.3.2 Ganzzahltypen

Name	Größe in Byte	Wertebereich
byte	1	-128 bis +127
short	2	-32768 bis +32767
int	4	-2.147.483.648 bis +2.147.483.647
long	8	-9.223.372.036.854.775.808 bis +9.223.372.036.854.775.807

2.3.3 Gleitpunkttypen

Name	Größe in Byte	Wertebereich
float	4	ungef. -3.4E+38 bis +3.4E+38
double	8	ungef. -1.7E+308 bis +1.7E+308

Daneben existiert noch der Datentyp `void`, der nur bei der Definition von Methoden(Funktionen) als Rückgabetyt "kein Rückgabewert" verwendet werden kann.

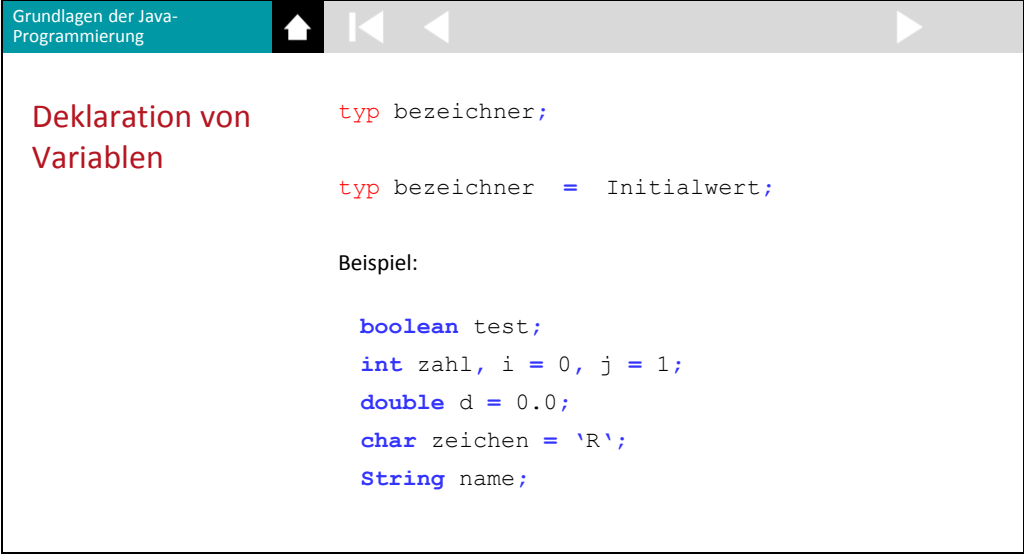
2.4 Deklaration und Definition von Variablen

Alle Variablen in Java müssen von einem bestimmten Datentyp sein. Der Typ bestimmt das interne Speicherformat, den Wert, den die Variable annehmen kann und die Operationen, welche mit dieser Variablen zulässig sind.

Die Deklaration einer Variablen hat folgendes Format, wobei die fett gedruckten Angaben obligatorisch sind:

```
typ bezeichner = Initialwert;
```

Mehrere Variable gleichen Typs können, durch Kommata getrennt, hinter einem Typ deklariert werden. Die Wertzuweisung an dieser Stelle ist optional, muss aber vor dem ersten Auslesen der Variablen erfolgen. Es ist nicht zulässig Variablen zu verwenden, bevor ihnen ein Wert zugewiesen wurde.



Grundlagen der Java-Programmierung

Deklaration von Variablen

```
typ bezeichner;  
  
typ bezeichner = Initialwert;
```

Beispiel:

```
boolean test;  
int zahl, i = 0, j = 1;  
double d = 0.0;  
char zeichen = 'R';  
String name;
```

Abb. 2-6: Deklaration

In Java gibt es nur lokale, keine globalen Variablen. Lokale Variablen dürfen innerhalb einer Methode an beliebiger Stelle und als Parameter in der Methodenklammer definiert werden.

Nach Konvention beginnen Variablennamen mit einem Kleinbuchstaben und sollen möglichst sprechende Namen bekommen.

2.4.1 Finale Variablen

Variablen mit festem Wert werden durch das Schlüsselwort `final` als definiert.

Nach geltender Konvention werden für finale Variablen ausschließlich Großbuchstaben verwendet. Finale Variablen können nach ihrer Initialisierung nicht mehr geändert werden.



The screenshot shows a presentation slide with a title bar 'Grundlagen der Java-Programmierung' and navigation icons. The main content is titled 'Finale Variablen' in red. It includes the general syntax `final typ bezeichner = Wert;`. An example shows `final double PI = 3.14159;` followed by 'oder' and a declaration `final double PI;` with a comment `//...` and an assignment `PI = 3.14159;`. A large red 'X' icon is placed next to the text 'Finale Variablen können nach ihrer Initialisierung nicht mehr geändert werden:'. Below this, it shows `final double PI = 3.14159;` followed by `PI = 2.176;` with a comment `// falsch: Neuzuweisung eines Wertes`.

```
final typ bezeichner = Wert;
```

Beispiel:

```
final double PI = 3.14159;
```

oder

```
final double PI;
```

```
//...
```

```
PI = 3.14159;
```

Finale Variablen können nach ihrer Initialisierung nicht mehr geändert werden:

```
final double PI = 3.14159;
```

```
PI = 2.176;    // falsch: Neuzuweisung eines Wertes
```

Abb. 2-7: Finale Variablen

2.5 Konstanten

Wie am Anfang dieses Abschnittes schon erwähnt, ist Java eine typgebundene Programmiersprache. Jedem Ausdruck, der im Programm verwendet wird, muss ein eindeutiger Datentyp zugeordnet sein. Deshalb haben auch Konstanten in Java einen Datentyp.

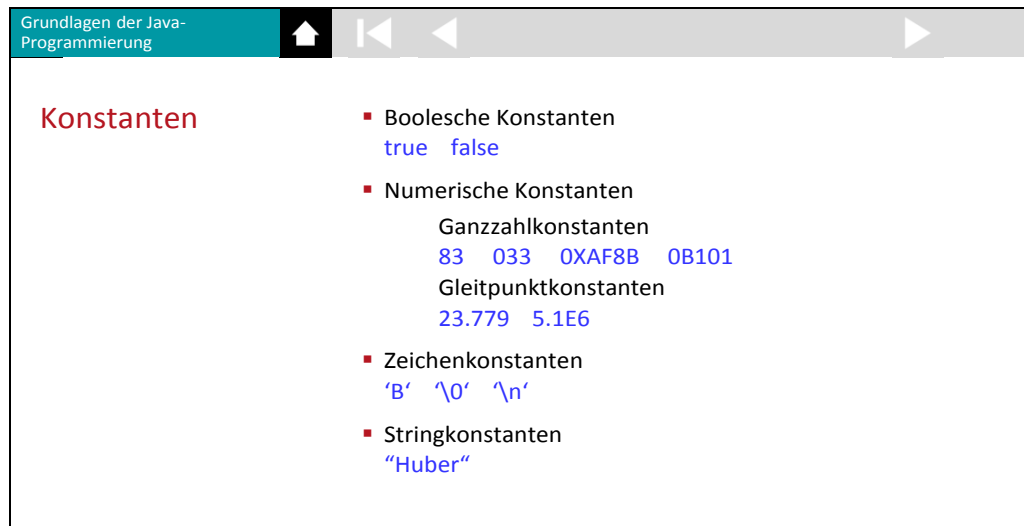


Abb. 2-8: Konstanten

2.5.1 Boolesche Konstanten

Es gibt die beiden booleschen Konstanten `true` und `false`.

Beispiel:

```
boolean ja = true;
boolean nein = false;
```

2.5.2 Numerische Konstanten

Die numerischen Konstanten werden in zwei Gruppen unterteilt, die Ganzzahlkonstanten und die Gleitpunktkonstanten.

2.5.2.1 Ganzzahlkonstanten

Ganzzahlkonstanten können im Dezimal-, Oktal- oder Hexadezimalformat angegeben werden:

- **Dezimalform**

Die erste Ziffer ungleich 0 (Null), gefolgt von den Dezimalziffern 0 bis 9.

`83`

`95_000_000` (seit JDK 7)

- **Oktalform**

Die führende 0 (Null) kennzeichnet eine Oktalzahl. Oktalziffern: 0 bis 7

`0145`

`033`

- **Hexadezimaler Form**

Eine führende 0 mit folgendem x oder X bedeutet, dass eine Hexadezimalzahl folgt. Die Hexadezimalziffern a bis f können auch mit A bis F notiert werden.

`0x4e`

`0XF5AF`

- **Binärform (seit JDK 7)**

Die führende 0 mit folgendem b kennzeichnet eine Binärzahl.

`0b101`

`0b11`

Eine ganzzahlige Konstante ist standardmäßig von Datentyp `int`.

Durch die Angabe eines Suffixes `L` oder `l` wird die Konstante zum `long`.

`12L`

`12l`

2.5.2.2 Gleitpunktkonstanten

Gleitpunktzahlen werden in Dezimal- oder Exponentialform angegeben:

`23.779`

`.99`

`88.`

`1.2e-3` `/* e oder E bezeichnet die Basis 10, die folgende ganze Zahl ist der Exponent zur Basis 10 */`

`5.1E6`

Die vor dem e/E stehende Ziffernkette heißt Mantisse, die auf e/E folgende Ziffernkette Exponent der Gleitpunktzahl. Beide Ziffernketten dürfen ein Vorzeichen haben, beide werden dezimal interpretiert. Der Exponent muss ganzzahlig sein.

Eine Gleitpunkt-Konstante ohne Suffix-Angabe hat den Typ `double`.

Eine Gleitpunkt-Konstante mit Suffix `f` oder `F` hat den Typ `float`.

`12.234F`

`12.234f`

2.5.3 Zeichenkonstanten

Zeichenkonstanten werden in einfache Hochkommata geschrieben und haben den Datentyp `char`.

Beispiel:

```
char z = 'A';  
z = '0'; // Das Zeichen 0 wird zugewiesen  
z = '\0'; // binär Null wird zugewiesen
```

Soll die Zeichenkonstante als Steuerzeichen verwendet werden, so sind aus Portabilitätsgründen folgende Werte zu verwenden:

<code>'\f'</code>	Formfeed
<code>'\n'</code>	Newline
<code>'\r'</code>	Carriage return
<code>'\t'</code>	Tabulator

Sonderzeichen müssen mit einem Backslash maskiert werden:

<code>'\\'</code>	Backslash
<code>'\"'</code>	Doppeltes Hochkomma
<code>'\''</code>	einfaches Hochkomma

Zeichenkonstanten können auch numerisch hinter einem Backslash angegeben werden. Der Zahlenwert muss oktall sein.

```
char zeichen = '\0377'; // (Oktale Angabe)
```

Unicodezeichen können in folgender Form angegeben werden:

```
char zeichenUnicode = '\uff1a';
```

Hinter `\u` müssen immer 4 Hexadezimalziffern folgen.

2.5.4 Stringkonstanten

Die Zeichen einer Stringkonstanten werden in doppelte Hochkommata geschrieben und haben den komplexen Datentyp `String`.

Bei der Verwendung einer Stringkonstanten handelt es sich immer um eine implizite Speicherplatzanforderung.

```
"Huber"  
"beliebiger Text mit \n\tSteuerzeichen\n"
```


3

Operatoren und Anweisungen

3.1	Operatoren	3-3
3.1.1	Zuweisungsoperator	3-4
3.1.2	Arithmetische Operatoren	3-4
3.1.3	Vergleichsoperatoren	3-6
3.1.4	Logische Operatoren	3-7
3.1.5	Explizite Typumwandlung mit dem Cast-Operator	3-7
3.1.6	Bedingte Bewertung	3-8
3.1.7	new Operator	3-9
3.1.8	instanceof Operator	3-9
3.2	Liste und Hierarchie Operatoren	3-10
3.3	Anweisungen	3-11
3.3.1	if-Anweisung	3-11
3.3.2	Case-Struktur switch	3-13
3.4	Schleifen	3-15
3.4.1	while-Schleife	3-15
3.4.2	do...while-Schleife	3-16
3.4.3	for-Schleife	3-17
3.4.4	Sprunganweisungen break und continue	3-18
3.4.5	for-each-Schleife	3-20
3.4.6	return-Anweisung	3-21
3.5	Komplexer Datentyp Array	3-22

3.5.1	Arrays	3-22
3.5.2	Mehrdimensionale Arrays	3-24

3 Operatoren und Anweisungen

3.1 Operatoren

Durch Verknüpfung von Operanden (z. B. Variablen und Konstanten) mit Operatoren können Ausdrücke gebildet und diese Operanden geändert werden. Es existieren Operatoren für:

- Zuweisung
- Arithmetik
- Vergleich
- Logik
- Typkonvertierung

Die Operatoren benötigen eine unterschiedliche Anzahl von Operanden. Es gibt die Gruppen der

1. unären Operatoren (1 Operand)
2. binären Operatoren (2 Operanden)
3. ternären Operatoren (3 Operanden)

Im Anschluss befindet sich eine Übersicht aller Operatoren mit Angaben zur Rangfolge und Assoziativität der Operatoren.

Operatoren und Anweisungen	
Operatoren	
▪ Zuweisungsoperator	=
▪ Arithmetische Operatoren	
a + b	Addition
a - b	Subtraktion
a * b	Multiplikation
a / b	Division
a % b	Modulo-Division (nur für Ganzzahlobjekte erlaubt)
▪ Kurzschreibweise	
a += b	
a -= b	
a *= b	
a /= b	
a %= b	
▪ Inkrement und Dekrement	
a++ oder ++a	
a-- oder --a	

Abb. 3-1: Operatoren

3.1.1 Zuweisungsoperator

Der Zuweisungsoperator ist für alle Typen definiert und wird von rechts nach links ausgewertet. Er kann auch direkt bei Definitionen von Variablen verwendet werden:

```
double d = 12.56;  
int i = 0, j = 0, k;  
k = 3;  
d = 1.5;
```

Bei der Verkettung des Zuweisungsoperators wird der Ausdruck von rechts nach links ausgewertet:

```
i = j = k;
```

3.1.2 Arithmetische Operatoren

Arithmetische Operationen werden in Java mit den üblichen Zeichen angegeben:

```
a + b          /* Addition */  
a - b          /* Subtraktion */  
a * b          /* Multiplikation */  
a / b          /* Division */
```

Zusätzlich gibt es die Modulo-Division:

```
a % b /* nur für Ganzzahlobjekte erlaubt */
```

Berechnet wird der Rest der Division von a durch b (ganzzahliger Rest).

Diese fünf Operatoren sind binär, da sie zwei Operanden benötigen.

Der Ausdruck hat als Wert das Ergebnis der arithmetischen Operation und als Datentyp den der Operanden nach eventuellen Konvertierungen (siehe Datentypumwandlungen).

Es gibt noch die beiden unären Vorzeichen-Operatoren, die nur einen Operanden haben.

Unärer Minus-Operator:

```
a * -b;          /* Negatives Vorzeichen */
```

Unärer Plus-Operator:

```
a * +b;          /* Positives Vorzeichen */
```

Kurzformschreibweise

Die arithmetischen Operatoren lassen sich mit dem Zuweisungsoperator kombinieren, wobei im Allgemeinen auch kürzerer Code generiert wird.

<code>a Operator= b;</code>	entspricht	<code>a = a Operator b;</code>
<code>a += b;</code>	ist dasselbe wie	<code>a = a + b;</code>
<code>a -= b;</code>	ist dasselbe wie	<code>a = a - b;</code>
<code>a *= b;</code>	ist dasselbe wie	<code>a = a * b;</code>
<code>a /= b;</code>	ist dasselbe wie	<code>a = a / b;</code>
<code>a %= b;</code>	ist dasselbe wie	<code>a = a % b;</code>

Inkrement und Dekrement:

Als weitere Möglichkeit der Ausdrucksverkürzung existieren die unären Inkrement- und Dekrementoperatoren, die sowohl als Präfix (Ausführung vorher) als auch als Postfix (Ausführung nachher) verwendet werden können:

`a = a + 1` lässt sich kürzer schreiben: `a++` oder `++a`

Entsprechend für

`a = a - 1` kann stehen: `a--` oder `--a`

Ob die Inkrement- bzw. Dekrementoperatoren vor das Objekt oder dahinter geschrieben werden, ist so lange egal, wie der Inkrement- oder Dekrementausdruck der einzige Operator in dem Ausdruck ist. In Kombination mit weiteren Operatoren allerdings es nicht gleichgültig, denn es gilt:

Steht der Inkrement/Dekrementoperator **hinter** dem Objekt, so wird der bisherige Wert des Objektes als Wert des ganzen Ausdrucks genommen.

Beispiel:

```
n = 3;  
c = n++;
```

Es wird erst der aktuelle Wert von n auf c zugewiesen und dann n inkrementiert. Nach der Anweisung hat n den Wert 4 und c den Wert 3.

Steht der Inkrement/Dekrementoperator dagegen **vor** dem Objekt, so ist der veränderte Wert des Objektes der Wert des ganzen Ausdrucks.

```
n = 3;  
c = ++n;
```

Es wird erst n inkrementiert, dann der neue Wert von n auf c zugewiesen, also haben beide den Wert 4.

Für die Dekrementierung gilt die Veränderung analog.

3.1.3 Vergleichsoperatoren

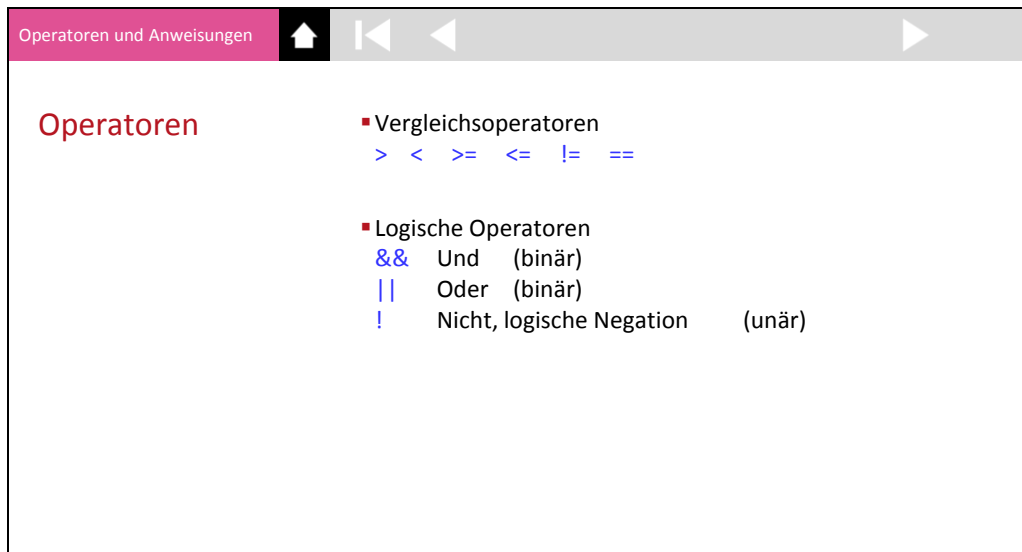


Abb. 3-2: Operatoren

Alle sechs Vergleichsoperatoren (Relationaloperatoren) gehören zur Gruppe der binären Operatoren. Das Ergebnis eines Vergleiches ist vom Datentyp boolean (true oder false).

Operator	Benutzung	Genau dann wahr, wenn
>	<code>a > b</code>	a größer b
<	<code>a < b</code>	a kleiner b
>=	<code>a >= b</code>	a größer oder gleich b
<=	<code>a <= b</code>	a kleiner oder gleich b
!=	<code>a != b</code>	a ungleich b
==	<code>a == b</code>	a gleich b

```
int a = 10;
double d = 120.1;
boolean b1, b2;
```

```
b1 = a <= 10; // b1 erhält den Wert true
b2 = d > 100; // b2 erhält den Wert true
```

3.1.4 Logische Operatoren

Vergleiche können mit **&&** (AND) und **||** (OR) verknüpft werden, sowie mit dem unären Operator **!** negiert werden.

```
int a = 10;
double d = 120.1;
boolean b1, b2, b3;

b1 = ( a <= 10 ) || ( d < 100.0 ); // true
b2 = ( a <= 10 ) && ( d < 100.0 ); // false
b3 = !( d < 100.0 );              // true
```

3.1.5 Explizite Typumwandlung mit dem Cast-Operator

Java ist eine streng typisierte Programmiersprache, der Compiler prüft auf korrekten Typ bei Zuweisungen und Übergabeparametern. Eine implizite Typumwandlung, ein Cast, ist aus einem kleineren in einen größeren Datentyp möglich. Eine Umwandlung in einen kleineren oder nur kompatiblen Datentyp ist zwingend explizit notwendig durch die Verwendung des Cast-Operators.

The screenshot shows a presentation slide with a pink header bar containing the text 'Operatoren und Anweisungen' and navigation icons. The slide content is as follows:

- Operatoren: Cast-Operator und ternärer Operator**
- Explizite Typumwandlung (Cast-Operator)
Syntax:
`(typ) Ausdruck`
Beispiel:
`char a = (char) 65;`
- Bedingte Bewertung (ternär)
Syntax:
`boolean ? Wert1 : Wert2`
Beispiel:
`min = (a < b) ? a : b;`

Abb. 3-3: Operatoren

Der Cast-Operator besteht nur aus einem runden Klammerpaar.

Syntax:

```
( NeuerTyp ) Ausdruck
```

Beispiel:

```
int i = 1;
double d = 2.6;

d = i;           // Konvertierung implizit
i = d;           // error
i = (int)d;      // explizite Typumwandlung
d = 1 / 2;       // Integer-Division: Ergebnis: 0
d = 1.0 / 2;     // Double-Division: Ergebnis: 0.5
d = (double)i / 2; // Double-Division
```

3.1.6 Bedingte Bewertung

Die Bedingte Bewertung wird in Java durch den ternären Operator **? :** dargestellt und kann als verkürzte Form der if...else-Konstruktionen eingesetzt werden.

Syntax:

```
boolean ? Wert1 : Wert2
```

Ist der boolesche Ausdruck true, entspricht der Gesamtwert Wert1, ansonsten Wert2.

Beispiel:

```
int a = 3, b = 34, min;
if ( a < b )
    min = a;
else
    min = b;

min = (a < b) ? a : b ;
```

3.1.7 new Operator

In Java werden Arrays und Objekte mit Hilfe des unären new-Operators erzeugt. Sowohl das Erzeugen eines Arrays als auch das Erzeugen eines Objekts sind Ausdrücke, deren Rückgabewert die Referenz auf das gerade erzeugte Array bzw. Objekt ist.

Beispiel:

```
int [] array;  
array = new int[10];  
Person willi;  
willi = new Person();
```

3.1.8 instanceof Operator

Der binäre instanceof-Operator wird verwendet, um herauszufinden, zu welcher Klasse ein bestimmtes Objekt gehört.

Syntax:

```
referenz instanceof Klasse
```

Der Ausdruck `a instanceof b` liefert genau dann `true`, wenn `a` eine Instanz der Klasse `b` oder einer ihrer Unterklassen ist. Falls das Ergebnis des instanceof-Operators nicht bereits zur Compile-Zeit ermittelt werden kann, generiert der Java-Compiler Code, um den entsprechenden Check zur Laufzeit durchführen zu können.

Beispiele:

```
String name = "Heinz";  
if ( name instanceof String )  
    System.out.println( "name ist vom Typ String" );
```

```
public boolean einmieten(Person pPerson)  
{  
    if( pPerson instanceof Rentner ){  
        System.out.println("Hotelgast ist Rentner");  
    }  
}
```

3.2 Liste und Hierarchie Operatoren

Alle Operatoren (außer den bitlogischen) sind hier nach Priorität aufgelistet. Die waagerechten Linien trennen die Prioritätsgruppen.

Symbol	Bedeutung	Assoziativität
() [] .	Klammeroperator, Funktionsaufruf Array Komponenten- oder Methodenauswahl	links nach rechts
! - + (unär) ++ -- (datentyp) new ~	logische Negierung negatives / positives Vorzeichen Inkrement Dekrement Typumwandlung, cast-Operator dynamische Speicheranforderung Einernkomplement	rechts nach links
* / %	Multiplikation Division Modulodivision	links nach rechts
+ -	Addition Subtraktion	links nach rechts
< <= > >= instanceof	Vergleiche: kleiner/größer Klassenzugehörigkeit	links nach rechts
== !=	Vergleiche: gleich ungleich	links nach rechts
&	bitweise UND-Verknüpfung	links nach rechts
^	bitweise exklusives ODER	links nach rechts
	bitweise inklusives ODER	links nach rechts
&&	logisches UND	links nach rechts
	logisches ODER	links nach rechts
? : (ternär)	Bedingte Bewertung	rechts nach links
= *= /= += - = %= <<= >>= >>= &= ^= =	Zuweisung zusammengesetzte Zuweisung zusammengesetzte bitweise Zuweisung	rechts nach links
,	Komma	links nach rechts

3.3 Anweisungen

Zunächst muss die Bedeutung der Begriffe Ausdruck und Anweisung klar unterschieden werden. Einfache Ausdrücke sind Konstanten und Variablen, komplexe Ausdrücke sind die Verknüpfungen derselben mit einem oder mehreren Operatoren.

Beispiel:

```
zahl * 10 / nenner
```

Jeder Ausdruck hat einen eindeutigen Typ und Wert.

Eine Anweisung in Java kann erstens aus einem Ausdruck mit anschließendem Semikolon generiert werden.

Zweitens wird eine Anweisung durch ein geschweiftes Klammerpaar dargestellt.

Die dritte Möglichkeit, eine Anweisung zu formulieren, ist die Verwendung eines Schlüsselwortes wie: if, while, do while, for, switch, return, break, continue.

3.3.1 if-Anweisung

Anweisungen:
Fallunterscheidung mit if

- Syntax:
`if (boolescher Ausdruck)`
 Anweisung
- Syntax mit else:
`if (boolescher Ausdruck)`
 Anweisung
`else`
 Anweisung
- Syntax alternativ mit Block:
`if (boolescher Ausdruck) {`
 Anweisungen
 Anweisungen
`}`
`else {`
 Anweisungen
 Anweisungen
`}`

Abb. 3-4: if

Die if-Anweisung wird benutzt, um in Abhängigkeit von einem Bedingungsausdruck bestimmte Programmteile auszuführen bzw. nicht auszuführen.

Die Anweisung besteht aus dem Schlüsselwort if, einem geklammerten Bedingungsausdruck sowie einer ausführbaren Anweisung (auch Verbundanweisung, Anweisungsblock).

Syntax:

```
    if (boolescher Ausdruck)
        Anweisung
```

Alternativ mit else-Zweig:

```
    if (boolescher Ausdruck)
        Anweisung
    else
        Anweisung
```

Ist der boolesche Ausdruck wahr, wird der if-Zweig durchlaufen, sonst der else-Zweig, falls vorhanden.

Beispiel:

```
int a = 3, b = 34, min;
```

```
    if ( a < b )
        min = a;
    else
        min = b;
```

```
if( x < 0 )
    if( y < 0 )
        System.out.println("x und y sind negativ\n");
    else
        System.out.println("x < 0, y >= 0\n");
else
    System.out.println("x ist >= 0\n");
```

3.3.2 Case-Struktur switch

Die Case-Struktur (Fallunterscheidung) in Java wird durch die switch-Anweisung abgebildet.

Der **Ausdruck** kann ganzzahlig oder ein Typ char sein. Seit JDK 7 ist auch der Typ String erlaubt.

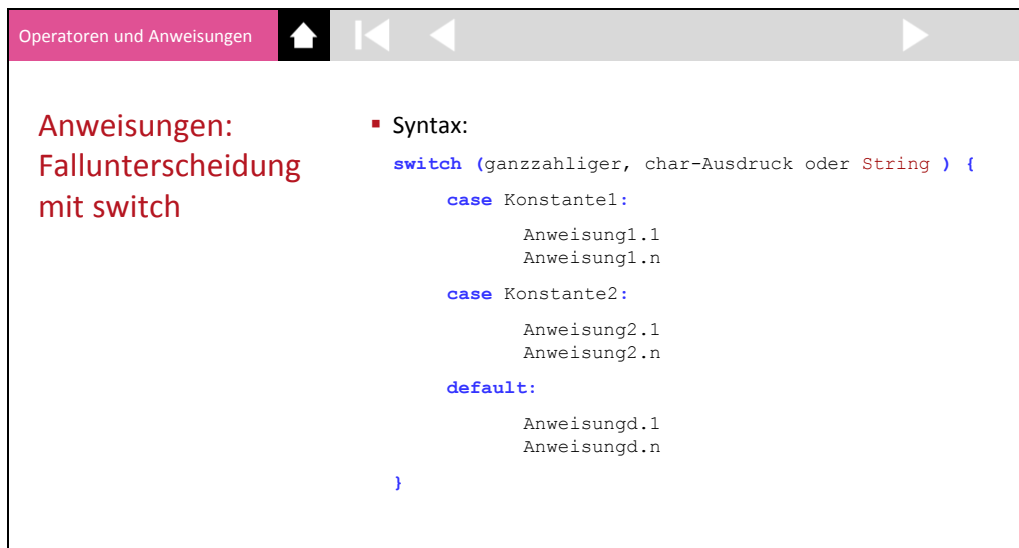


Abb. 3-5: switch

Wirkungsweise:

Der Ausdruck in der runden Klammer von switch muss ein String, ganzzahlig oder von Typ char sein. Sein Wert wird mit den Konstanten hinter dem Schlüsselwort case auf Gleichheit geprüft. Wenn Gleichheit besteht, setzt die Programmsteuerung bei der ersten dort stehenden Anweisung auf. Alle folgenden Anweisungen bis zum Ende des switch werden ausgeführt. Deshalb ist es sinnvoll, die einzelnen Zweige mit break abzuschließen.

Bei jedem case dürfen nur Konstanten stehen, keine allgemeinen Ausdrücke, insbesondere keine Oder-Verknüpfungen von Werten. Jedoch ist es erlaubt, gar keine Anweisung hinter ein case zu schreiben. Das ersetzt die Wirkung einer Oder-Verknüpfung.

Beispiel:

```
int monat = 3, tage = 0;
switch(monat)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        tage = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        tage = 30;
        break;
    case 2:
        tage = 28;
        break;
    default:
        tage = 0;
}
```

3.4 Schleifen

Operatoren und Anweisungen

⬆
⏮
⏪
⏩
⏭

**Anweisungen:
while-Schleife und
do-while-Schleife**

- Syntax while-Schleife:
(kopfgesteuert)

```
while (boolescher Ausdruck)
    Anweisung
```

- Syntax alternativ mit Block:

```
while (boolescher Ausdruck){
    Anweisungen
    Anweisungen
}
```

- Syntax do-while-Schleife:
(fußgesteuert)

```
do
    Anweisung
while (boolescher Ausdruck);
```

- Syntax alternativ mit Block:

```
do {
    Anweisungen
    Anweisungen
}while (boolescher Ausdruck);
```

Abb. 3-6: Schleifen

3.4.1 while-Schleife

Die kopfgesteuerte while-Schleife:

Syntax:

```
while( boolescher Ausdruck )
    Anweisung
```

Zuerst wird geprüft, ob die Bedingung wahr ist. Wenn ja, wird der Schleifenkörper durchlaufen und zwar solange die Bedingung des Schleifenkopfes wahr ist. Man nennt diese Schleife kopfgesteuert, da die Bedingung am Kopf der Schleife überprüft wird. Ist die Bedingung falsch, wird die Schleife nicht durchlaufen.

Beispiel:

```
int i = 0;
/* Ausgabe der Zahlen von 0 bis 9 */
while(i < 10)
{
    System.out.println (" " + i);
    i++;
}
System.out.println (" " + i);
```


Die nächste Schleife läuft endlos (warum?):

```
int i = 0;
while ( i < 10 );
{
    i++;
}
```

Das Semikolon hinter dem while-Kopf ist eine Leeranweisung. Folglich wird die Zählvariable i niemals erhöht, da der Block nicht zur while-Schleife gehört.

3.4.2 do...while-Schleife

Die fußgesteuerte Schleife:

Syntax:

```
do
    Anweisung;
while( boolescher Ausdruck );
```

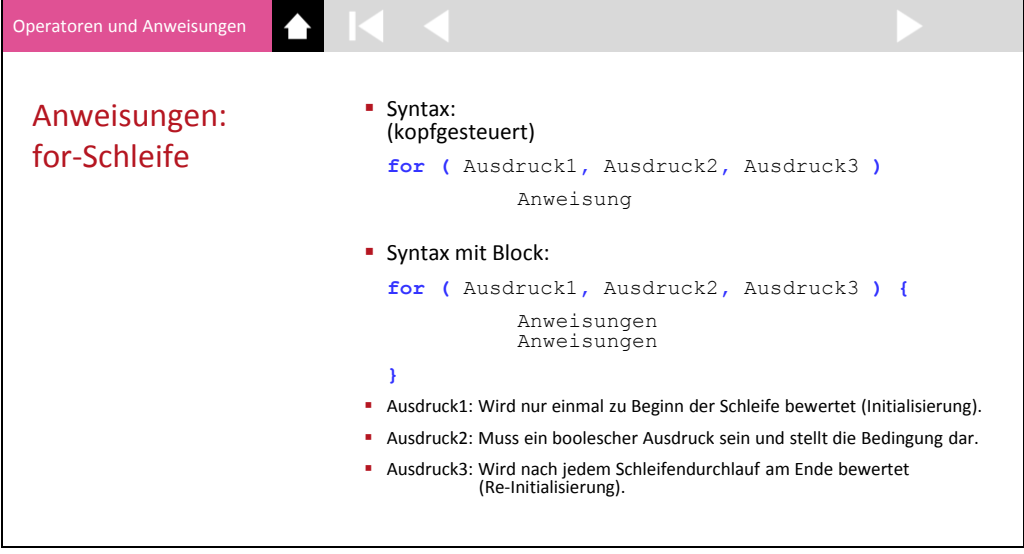
Man beachte, dass das Semikolon am Ende des Schleifenfußes zur Syntax gehört.

Diese Schleife nennt man fußgesteuert, da die Bedingung erst am Ende überprüft wird und daher die Schleife mindestens einmal durchlaufen wird.

Beispiel:

```
int i = 0;
do
{    /* Ausgabe der Zahlen von 0 - 9 */
    System.out.println ( " " + i++ );
}while( i < 10 );
```

3.4.3 for-Schleife



The screenshot shows a presentation slide with a pink header bar containing the text 'Operatoren und Anweisungen' and navigation icons. The main content area has a light gray background. On the left, the text 'Anweisungen: for-Schleife' is displayed in red. To the right, there are two bullet points with code examples. The first bullet point shows the basic syntax for a for loop with three expressions. The second bullet point shows the syntax for a for loop with a block of statements. Below these, three more bullet points provide detailed explanations of the expressions: Ausdruck1 is for initialization, Ausdruck2 is the loop condition, and Ausdruck3 is for re-initialization.

**Anweisungen:
for-Schleife**

- Syntax:
(kopfgesteuert)
`for (Ausdruck1, Ausdruck2, Ausdruck3)`
 Anweisung
- Syntax mit Block:
`for (Ausdruck1, Ausdruck2, Ausdruck3) {`
 Anweisungen
 Anweisungen
`}`
- Ausdruck1: Wird nur einmal zu Beginn der Schleife bewertet (Initialisierung).
- Ausdruck2: Muss ein boolescher Ausdruck sein und stellt die Bedingung dar.
- Ausdruck3: Wird nach jedem Schleifendurchlauf am Ende bewertet (Re-Initialisierung).

Abb. 3-7: Schleife

Beispiel:

```
int i;  
for( i = 0; i < 10; i++ )  
{  
    // Anweisungen  
}
```

Obige for-Schleife zeigt gleiche Wirkung wie die folgende while-Schleife:

```
i = 0;  
while( i < 10 )  
{  
    // Anweisungen  
    i++;  
}
```

Im Initialisierungs- und Reinitialisierungsteil der for-Schleife können mehrere Ausdrücke angegeben werden, diese müssen dann durch Kommata getrennt werden.

```
int i, j;  
for( i = 1, j = 10 ; i <= 5 ; i++, j -= 2 )  
{  
    // Anweisungen  
}
```

Im Kopf der for-Schleife kann eine Variable neu angelegt und im Körper der Schleife benutzt werden. Nach Verlassen der Schleife existiert diese Variable nicht mehr. Diese for-Schleifen werden häufig für die Bearbeitung von Arrays verwendet.

```
for(int i = 0; i < 10; i++)  
{  
    System.out.println( i );  
}
```

3.4.4 Sprunganweisungen break und continue

The screenshot shows a presentation slide with a pink header bar containing the text 'Operatoren und Anweisungen' and navigation icons. The main content area has a light blue background and lists three types of jump instructions:

- Anweisungen:**
return und instanceof
- Sprunganweisung
Syntax:
`break;`
`continue;`
- return
Syntax:
`return` Ausdruck;
`return;`
- instanceof Operator
Syntax:
referenz `instanceof` Klasse

Abb. 3-8: Sprunganweisungen

Schleifen lassen sich vorzeitig mit `break` verlassen oder mit `continue` neu aufsetzen.

In diesem Beispiel werden nur die geraden Zahlen angezeigt.

```
int i = 0;
while( i++ < 10 ) {
    if( ( i % 2 ) == 1 )
        continue;
    System.out.println( "i: " + i );
}
```

Die folgende Schleife wird bei i = 20 verlassen:

```
int i = 0;
while( true ) {
    if( i == 20 )
        break;
    ++i;
}
```

Die break-Anweisung kann mit einem Label versehen und für das vorzeitige Verlassen geschachtelter Schleifen verwendet werden. Dieses Format entspricht in etwa den goto-Anweisungen anderer Sprachen.

```
int i ,j = 100;
marke: for( i=1 ; i <= 100; i++ ) {
    for( ; j > 0 ; j-- ) {
        if( j == 50 )
            break marke ;
    }
}
```

Hat j den Wert 50 wird das Label marke angesprungen und die mit diesem Label markierte äußere for-Schleife verlassen. Danach hat i den Wert 1 und j den Wert 50.

Ersetzt man im obigen Beispiel

```
break marke;
```

durch

```
continue marke;
```

so wird die äußere for-Schleife weiter durchlaufen. Hinterher hat i den Wert 101 und j den Wert 50.

3.4.5 for-each-Schleife

The screenshot shows a presentation slide with a pink header bar containing the text 'Operatoren und Anweisungen' and navigation icons. The main content area has a title 'Anweisungen: forEach-Schleife' in red. Below the title, there are two bullet points describing the syntax of the for-each loop. The first bullet point describes the 'kopfgesteuert' (header-controlled) syntax, and the second bullet point describes the 'Syntax mit Block' (syntax with block). Below these, there are three definitions for the variables 'typ', 'Element', and 'Sammlung'.

```

Anweisungen:
forEach-Schleife

▪ Syntax:
  (kopfgesteuert)
  for ( typ Element : Sammlung )
      Anweisung

▪ Syntax mit Block:
  for ( typ Element : Sammlung ) {
      Anweisungen
      Anweisungen
  }

▪ typ:      Kann ein einfacher oder komplexer Datentyp sein.
▪ Element: Variablenname für das einzelne Element der Sammlung.
▪ Sammlung: Kann ein Array oder eine andere Collection sein.
  
```

Abb. 3-9: Schleife

Früher musste innerhalb der for-Schleife immer ein Zähler definiert werden. Seit JDK 5 gibt es zusätzlich folgende Syntax:

Syntax:

```
for( typ element : sammlung )
    Anweisung;
```

typ: Der Datentyp (einfach oder komplex) des einzelnen Elements der Sammlung.

element: Variablenname für das einzelne Element der Sammlung

sammlung: Jede Art von Collection, z.B. ein Array oder eine ArrayList.

Diese Form der Schleife wird gelesen als „Für alle Elemente aus dem Array/der Liste...“ und wird auch häufig als „**foreach**“-Schleife bezeichnet.

3.4.6 return-Anweisung

Die Anweisung `return` wird am Ende von Methoden benutzt. Sie beendet eine Methode und bewirkt einen Rücksprung zum Programmteil, welcher die Methoden aufgerufen hat.

Syntax:

```
return Ausdruck;
```

Der Wert des Ausdruckes wird an die aufrufende Stelle zurückgegeben und muss vom gleichen Typ wie die Methode sein, in der diese `return`-Anweisung steht.

Wird eine Methode mit dem Datentyp `void` definiert, so entspricht dies einer Prozedur in anderen Programmiersprachen. Diese `void`-Methoden können keinen Wert zurückliefern und entsprechend steht bei der `return`-Anweisung kein Ausdruck.

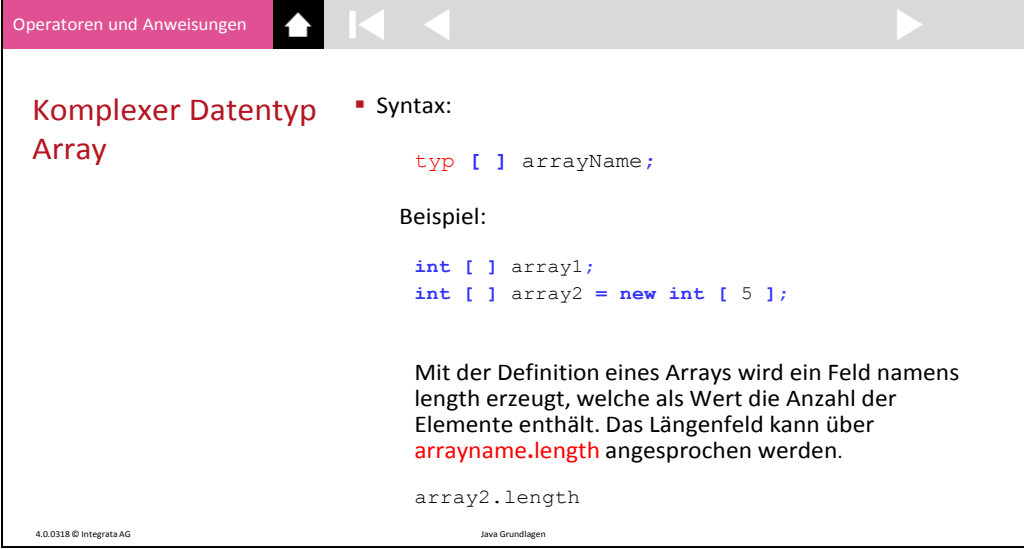
Syntax:

```
return ;
```

3.5 Komplexer Datentyp Array

Zu den komplexen Datentypen gehören einerseits die Arrays und andererseits die Klassen.

Alle Objekte eines komplexen Datentyps werden über Referenzen angesprochen. Es muss der Name des Objektes, die „Variable“ deklariert werden und außerdem der Speicherplatz für das Objekt selbst allokiert werden.



The screenshot shows a presentation slide with a pink header bar containing the text 'Operatoren und Anweisungen' and navigation icons. The main content area has a title 'Komplexer Datentyp Array' in red. To the right of the title is a small red square followed by the word 'Syntax:'. Below this, the syntax for declaring an array is shown: `typ [] arrayName;`. Further down, under the heading 'Beispiel:', two examples of array declarations are provided: `int [] array1;` and `int [] array2 = new int [5];`. A paragraph of text explains that when defining an array, a field named `length` is created, which holds the number of elements. It notes that this field can be accessed via `arrayname.length`. The example `array2.length` is shown below this text. At the bottom left, the text '4.0.0318 © Integrata AG' is visible, and at the bottom right, 'Java Grundlagen' is visible.

Komplexer Datentyp Array ■ Syntax:

```
typ [ ] arrayName;
```

Beispiel:

```
int [ ] array1;  
int [ ] array2 = new int [ 5 ];
```

Mit der Definition eines Arrays wird ein Feld namens `length` erzeugt, welche als Wert die Anzahl der Elemente enthält. Das Längelfeld kann über `arrayname.length` angesprochen werden.

```
array2.length
```

4.0.0318 © Integrata AG Java Grundlagen

Abb. 3-10: Array

3.5.1 Arrays

Bei einem Array handelt es sich um einen homogenen Datentyp, also die Aneinanderreihung von Daten desselben Typs. Arrays dienen als Container für einfache Datentypen und Referenzen. Jedes Array hat eine bestimmte Größe und einen bestimmten Typ.

Bei der Deklaration eines Arrays wird ein leeres eckiges Klammernpaar `[]` angegeben.

Syntax:

```
elementType [ ] arrayName;
```

Unter Java gibt es zwei Möglichkeiten, ein Array anzulegen:

1. Mit `new` und Nullinitialisierung
2. Mit Initialisierung durch Aufzählung in einem geschweiften Klammernpaar

Beispiel:

```
int[] array1;  
int[] array2 = new int[ 5 ] ;
```

In zweiten Fall werden alle Arrayelemente mit dem Wert 0 initialisiert. Später können die Elemente wie folgt angesprochen werden:

```
array2[0] = 2;  
array2[1] = 3;  
array2[2] = 4;  
array2[3] = 5;  
array2[4] = 6;
```

Ein Array kann bei der Definition auch mit Werten initialisiert werden:

```
int[] array3 = {1, 3, 5, 7};
```

Bei der Überschreitung der Arraygrenzen mit einem unzulässigen Index wird zur Laufzeit ein Ausnahmefehler hervorgerufen.

Beispielsweise bei folgenden Zugriffen:

```
array2[-1] = 13;  
array2[25] = 87;
```

Mit der Definition eines Arrays wird ein Feld namens `length` erzeugt, welche als Wert die Anzahl der Elemente enthält. Das Längelfeld kann über `arrayname.length` angesprochen werden.

```
array2.length
```

Zu beachten ist, dass bei der Deklaration von Arrays die eckigen Klammern Paare grundsätzlich leer sein müssen.

```
int dim = 5;  
  
int [5] array1 = { 2, 3, 4, 5, 6 }; // falsch  
int [dim] array2 = new int [ 5 ] ; // falsch
```

Möglich ist aber:

```
int[] array = new int[ dim ] ;
```

Über einen Index kann auf das gewünschte Element eines Arrays zugegriffen werden. Der Zugriff auf das erste Element erfolgt immer über den Index 0.

Bei der Bearbeitung von Arrays sind for-Schleifen prädestiniert:

```
for(int i = 0; i < array2.length; i++)  
{  
    System.out.println( "[i] = " + array2[i] );  
}
```


3.5.2 Mehrdimensionale Arrays

Sogenannte mehrdimensionale Arrays (Array im Array) werden mit mehreren eckigen Klammerpaaren deklariert:

```
int[][] multiInt = new int[2][3];
```

Hier wird ein Array mit 2 Dimensionen angelegt.

Jedes Element der 1. Dimension ist seinerseits eine Referenz auf ein Array von drei `int`-Werte. Die Initialisierung kann dann komponentenweise erfolgen:

```
multiInt[0][0] = 1;
multiInt[0][1] = 2;
multiInt[0][2] = 3;
multiInt[1][0] = 4;
multiInt[1][1] = 5;
multiInt[1][2] = 6;
```

Definition und Initialisierung können auch in einem Schritt erfolgen:

```
int[][] multiInt = { {1, 2, 3 },
                    {4, 5, 6 }
                  };
```

Zweidimensionale Arrays müssen in Java nicht "rechteckig" zu sein, daher sind auch folgende Gebilde möglich:

```
double[][] multiDouble = { {1.0}
                           , {2.0, 3.0}
                           , {4.0, 5.0, 6.0}
                           };
```

4

Objektorientierte Programmentwicklung

4.1	Prozedurale versus objektorientierte Programmentwicklung	4-3
4.2	Objektorientierter Ansatz	4-4
4.2.1	Klassifizieren.....	4-5
4.2.2	Abstrahieren	4-6
4.2.3	Ordnen, Bilden von Hierarchien	4-6
4.3	Objekt	4-7
4.4	Klassen.....	4-9
4.4.1	Entwurf einer Klasse	4-9
4.4.2	UML-Notation.....	4-10
4.4.3	Zugriffsrechte und Deklarationen	4-12
4.4.4	Klassendefinition	4-13
4.5	Attribute	4-14
4.5.1	Attributdeklaration	4-14
4.6	Methoden (Funktionalitäten)	4-15
4.6.1	Methodendeklaration	4-15
4.6.2	Überladen von Methoden.....	4-16
4.6.3	Die main-Methode.....	4-17
4.6.4	Die Methode println()	4-18
4.7	Referenzen und Instanz Erzeugung	4-19
4.7.1	Zugriff auf Attribute und Methoden.....	4-20
4.7.2	Die this-Referenz	4-20

4.7.3	Formen von Objektreferenzen	4-22
4.7.4	Referenzen im Speicher der Virtuellen Maschine	4-23
4.8	Konstruktoren	4-24
4.8.1	Konstruktordeklaration	4-25
4.8.2	Überladen von Konstruktoren	4-27
4.8.3	Private Konstruktoren	4-29
4.9	Klassenattribute und Klassenmethoden	4-30
4.9.1	Klassenattribute	4-30
4.9.2	Klassenmethoden	4-31
4.9.3	Initialisierungen von Klassenattributen	4-32
4.9.4	Statische Elemente und die Virtuelle Maschine	4-33
4.11	Ausnahmen	4-34
4.11.1	Traditionelle Fehlerbehandlung	4-34
4.11.2	Die Klasse Exception	4-36
4.11.3	Beispiel Kehrwert.....	4-38
4.11.4	Ausnahmen mit fehlerbeschreibendem Text.....	4-39

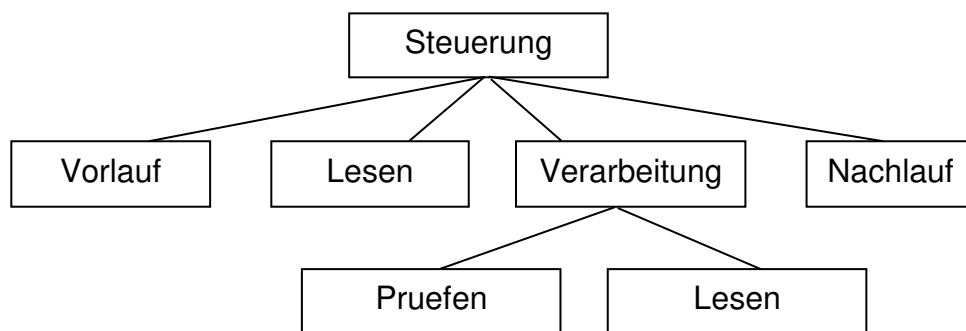
4 Objektorientierte Programmentwicklung

4.1 Prozedurale versus objektorientierte Programmentwicklung

Programmierung bedeutet, die Realität (oder Teile davon) in einem dem Computer verständliches Modell abzubilden. Der dazu notwendige Abstraktionsprozess prägt sich in prozeduralen Programmen wesentlich stärker aus als bei objektorientierten Entwicklungen.

Die in den prozeduralen Programmiersprachen (Fortran, COBOL, Pascal, C) verwendete Abstraktionsmethode ist die Zerlegung in Prozeduren (Funktionen), die sich gegenseitig aufrufen und bei den Aufrufen Daten austauschen.

Ein Strukturdiagramm eines prozeduralen Programms könnte folgendes Aussehen haben:



Dabei wird bei Betrachtung dieses Diagramms und der einzelnen Prozeduren nicht sofort deutlich, was z.B. „Lesen“ oder „Pruefen“ bedeutet. Welche Daten werden gelesen? Was wird geprüft?

Im Gegensatz dazu bedeutet Objektorientierung, die Realität abzubilden, die Komplexität wird reduziert. Aus den Substantiven des realen Problems werden Klassen gebildet.

Klassen haben Eigenschaften, die Attribute. Sie sind die Daten der Klasse. Dabei ist eine Klasse in einem ersten Ansatz nichts anderes als ein record in Pascal oder ein struct in C.

Die Erweiterung besteht darin, dass Klassen auch Fähigkeiten besitzen, das sind die Methoden der Klasse.

4.2 Objektorientierter Ansatz

Meistens schreibt man Programme, um damit Vorgänge in der realen Welt zu modellieren, zu unterstützen oder zu automatisieren. In dieser Welt sind wir nicht von Datenstrukturen umgeben, sondern von Objekten, also von Tieren, Ampeln, Autos oder PCs. Wenn unsere Programme also einen Bezug zur physischen Welt haben sollen, liegt es nahe, auch in den Programmen mit Objekten umzugehen.

Objektorientierte Programmentwicklung

Das Problem

Komplexes, ungeordnetes System, Zusammenhänge?
Was ist wesentlich, was unwesentlich?
Existieren Abhängigkeiten?

Abb. 4-1: Das Problem

Wir haben also ein komplexes, ungeordnetes Chaos. Was ist wesentlich, was ist unwesentlich?

Objektorientierte Programmentwicklung

Der objektorientierte Ansatz

- Klassifizieren
- Abstrahieren
- Ordnen, Bilden von Hierarchien
- Ein „menschlicher“ Lösungsansatz!

Abb. 4-2: Objektorientierter Ansatz

Der erste Schritt bei der Entwicklung eines objektorientiert aufgebauten Programms ist somit immer, den Ausschnitt der Realität zu betrachten, der für dieses Programm relevant ist, und die darin vorkommenden Objekte sowie deren Beziehungen untereinander zu identifizieren.

Der Lösungsansatz der objektorientierten Programmierung ist also Klassifizieren, Abstrahieren und Hierarchien bilden.

4.2.1 Klassifizieren

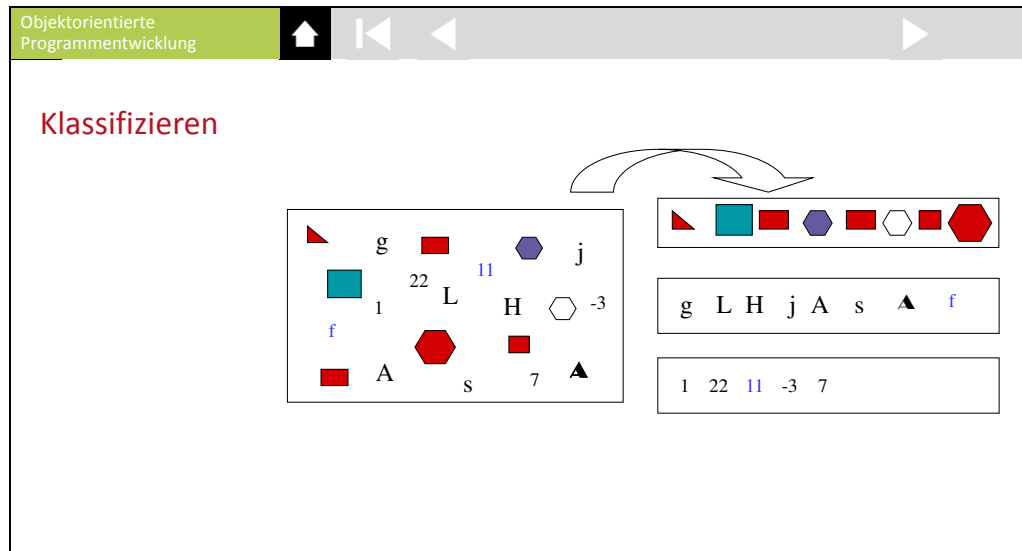


Abb. 4-3: Klassifizieren

Aus dem Beispielen wird deutlich, wodurch sich ein Objekt charakterisiert: nämlich durch seine *Eigenschaften* (auch *Attribute* genannt). Die Arten von Eigenschaften sind bei jeder Gruppe von Objekten dieselben, ihre Werte indessen verschieden. Auch im realen Leben halten wir ja Gegenstände anhand ihrer Eigenschaften auseinander.

Ein weiteres wichtiges Merkmal ist das Verhalten eines Objekts. Dieses hängt ab vom *Zustand*, in dem das Objekt sich augenblicklich befindet.

4.2.2 Abstrahieren

Ein Zeichnungsobjekt **hat eine** Farbe, eine Position und eine Größe als Eigenschaften. Das komplexe Zeichnungsobjekt ist eine **Komposition** einfacherer Elemente.

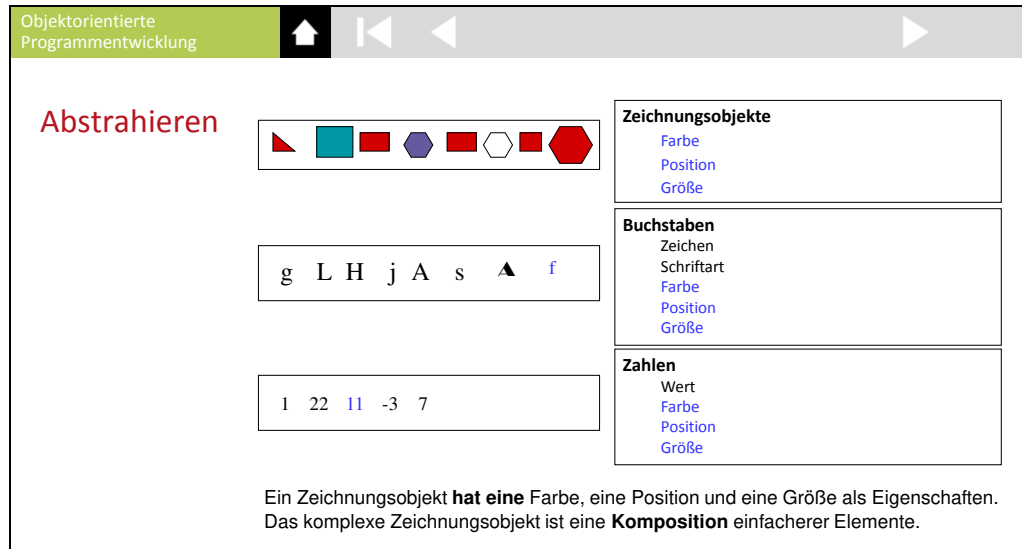


Abb. 4-4: Abstrahieren

4.2.3 Ordnen, Bilden von Hierarchien

Ein Buchstabe **ist ein** Zeichnungsobjekt, das ein Zeichen in einer Schriftart darstellt.

Eine Zahl **ist ein** Zeichnungsobjekt, das einen Zahlenwert darstellt.

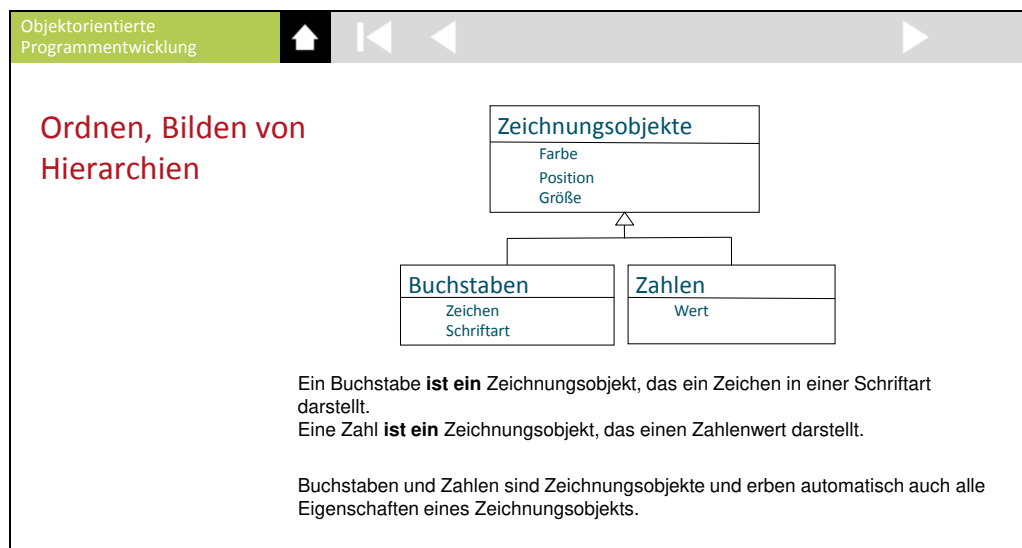


Abb. 4-5: Ordnen

Buchstaben und Zahlen sind Zeichnungsobjekte und erben automatisch auch alle Eigenschaften eines Zeichnungsobjekts.

4.3 Objekt


Objektorientierte Programmentwicklung

Was ist ein Objekt?

ganz konkrete

- ☐ Gegenstände,
- ☐ Geräte,
- ☐ Ereignisse,
- ☐ Strukturen,
- ☐ Rollen,
- ☐ Örtlichkeiten,

eben alles,
wovon man sich einen
Begriff machen kann



Ein Objekt

- ☐ ist die Abstraktion eines "Begriffs",
- ☐ *hat* eine eigene Identität,
- ☐ zeigt ein für seine Art typisches Verhalten,
- ☐ hat zu jedem Zeitpunkt einen bestimmten Zustand, der für das Verhalten in bestimmten Situationen ausschlaggebend sein kann.




Abb. 4-6: Objekt

Ein Objekt (eine Instanz) hat Attribute und Methoden. Ein Objekt ist ein Exemplar oder (wie der Fachausdruck dafür lautet) eine Instanz einer Klasse. Welche Attribute und Methoden ein Objekt besitzt, wird bei der Deklaration der Klasse festgelegt.

Objektorientierte Programmentwicklung

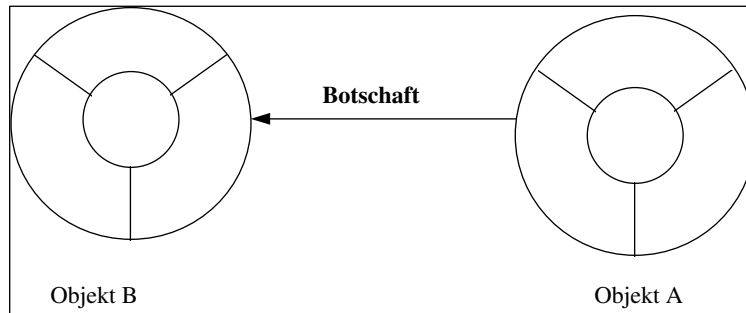
Objekt

Ein Objekt ist das Ergebnis eines reproduzierbaren Produktionsprozesses.
Ein Objekt ist eine Instanz (ein Exemplar) einer Klasse.

- Ein Objekt besitzt
 - Eigenschaften \Rightarrow **Attribute**
 - Fähigkeiten \Rightarrow **Methoden**
 - Interaktivität \Rightarrow **Botschaften**
- Analogie zu traditionellen Programmen
 - Attribute \Leftrightarrow Variable
 - Methoden \Leftrightarrow Funktionen, Prozeduren
 - Botschaften \Leftrightarrow Ablaufsteuerung, Parameter
- Anderer Denkansatz:
 - Ein Objekt ist stets das Ergebnis einer Produktion.
 - **Objekte werden aus der Klasse erzeugt = instanziiert**

Abb. 4-7: Objekt

Ein Objekt für sich allein ist in den meisten Fällen wenig sinnvoll. Erst wenn Objekte miteinander in Kontakt treten und Informationen austauschen, besteht die Möglichkeit, komplexe Sachverhalte abzubilden. Wenn das Objekt A eine der Methoden von Objekt B ansprechen möchte, dann sendet Objekt A an Objekt B eine Botschaft.



Eine Botschaft besteht aus drei Bestandteilen:

- a) Das Zielobjekt (Objekt B)
- b) Der Name der auszuführenden Methode
- c) Die Übergabeparameter zu dieser Methode

Anhand dieser Informationen kann dann das Zielobjekt die gewünschte Methode ausführen.

In Java ist die Übermittlung von Botschaften mit der Parameterübergabe an Methoden zu realisieren.

Auch der Begriff der Botschaft ist umgangssprachlich zu fassen: Einfache Sätze bestehen aus drei Bestandteilen, dem Subjekt, dem Prädikat und dem Objekt. Diese Begriffe sind analog zu den Bestandteilen der Botschaft.

4.4 Klassen

Der zentrale Begriff der objektorientierten Programmierung ist die Klasse.

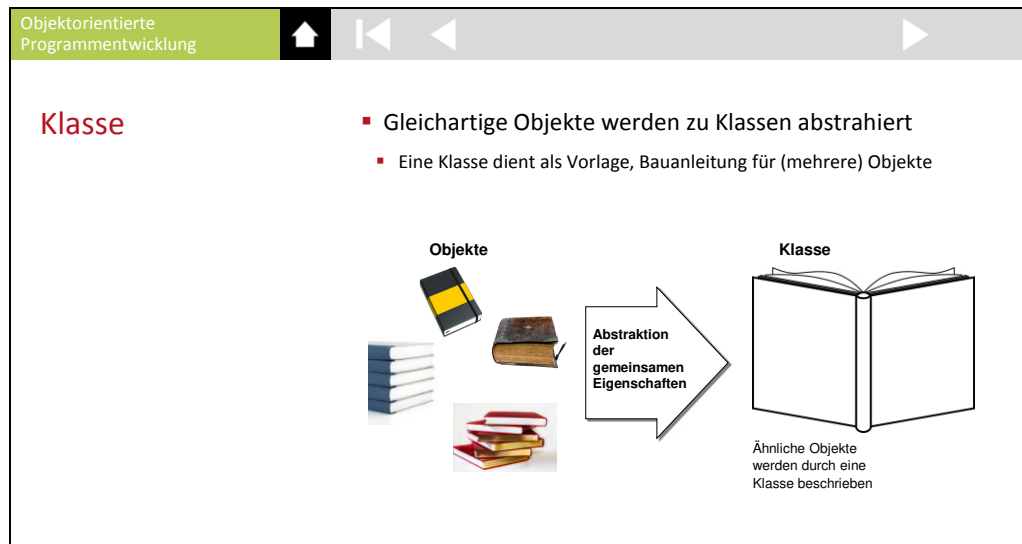


Abb. 4-8: Klasse

Eine Klasse ist zunächst nur die Beschreibung (Deklaration) oder Bauanleitung eines zukünftigen Objektes. Erst wenn nach dieser Bauanleitung eine Instanz dieser Klasse im Speicher angelegt wird, spricht man von einem Objekt.

Bei dieser Vorgehensweise werden nicht nur die Attribute eines Objekts beschrieben, sondern auch die Methoden, die die Kommunikationsschnittstellen nach außen darstellen.

Bei der Deklaration einer Klasse bleiben die Attribute üblicherweise verborgen (Kapselung) und nur die Methoden werden für den allgemeinen Gebrauch bekannt gegeben. Aus diesem Grund sind Methoden der verhaltensorientierte Teil einer Klasse und die einzige Möglichkeit, Daten der Klasse zu manipulieren.

Die Aktivierung von Methoden erfolgt über Botschaften bzw. Nachrichten oder Parameter, d. h. Objekte können untereinander über die wohldefinierten Schnittstellen miteinander kommunizieren.

4.4.1 Entwurf einer Klasse

Als konkretes Beispiel sei hier der Versuch der Modellierung einer Person durchgeführt. Im hier präsentierten Ansatz besteht die Klasse Person aus den beiden Attributen

nachname Folge von Einzelzeichen

vorname Folge von Einzelzeichen.

Eine Person besitze in diesem Beispiel die Fähigkeit, ihren Namen zu sagen. Eine Fähigkeit wird in einer Klasse als Methode abgebildet. Da diese Methode eine Information aus dem Objekt holt, wird daraus eine get-Methode, nennen wir sie `getName()`.

Sie wird sowohl den Vornamen als auch den Nachnamen liefern. Der Rückgabewert besteht aus einer Folge von Einzelzeichen, wird also vom Typ String sein.

4.4.2 UML-Notation

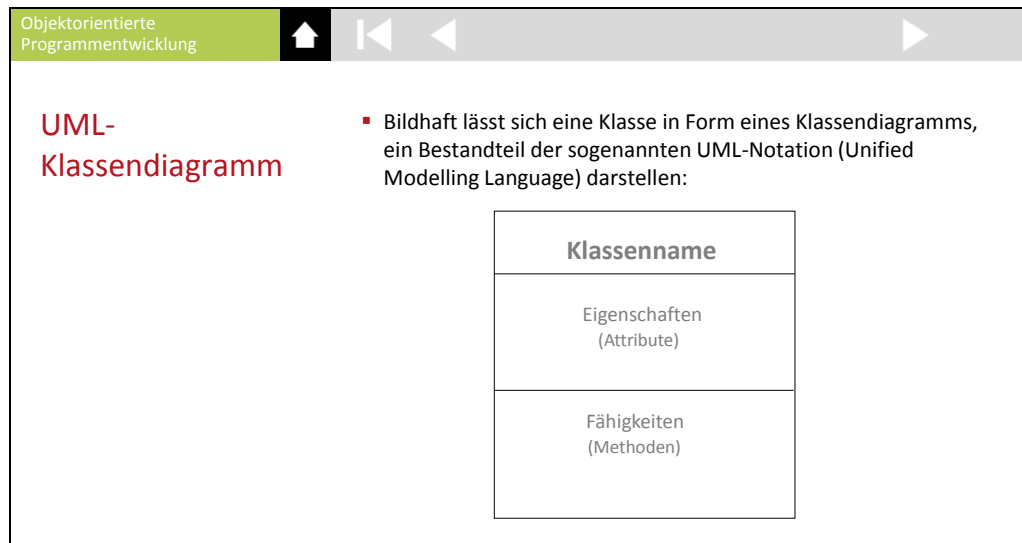
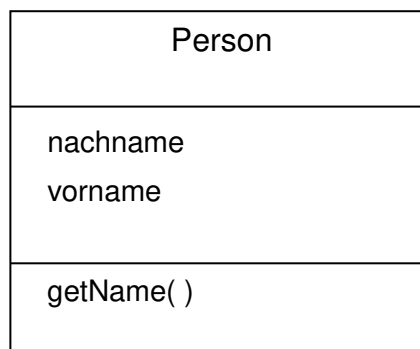


Abb. 4-9: UML

Bildhaft lässt sich die Klasse Person in Form eines Klassendiagramms, einem Bestandteil der UML-Notation (Unified Modelling Language) darstellen:



Nicht alle Attribute und Methoden einer Klasse können sinnvoll nach außen gegeben werden. Interna können innerhalb der Klasse gekapselt werden und sind dann außerhalb der Klasse nicht sichtbar. Für diese Kapselung von Attributen und Methoden gibt es mehrere Gründe:

- **Sicherheit** Eine direkte Manipulation von Attributen oder das Aufrufen von Methoden eines Objektes ohne detaillierte Kenntnis des internen Aufbaus der entsprechenden Klasse kann zu unvorhergesehenen Effekten führen.
- **Wartbarkeit** Gekapselte Attribute und Methoden können modifiziert oder verbessert werden, ohne dass bei Verwendung der Klasse die Änderungen berücksichtigt werden müssten.
- **Wiederverwendbarkeit** Eine Klasse, die die gewünschten Attribute und Methoden besitzt, kann verwendet werden, ohne Rücksicht auf und Kenntnis von internen Details.

Es ist somit notwendig, zwei Arten von Attributen und Methoden zu definieren: öffentliche und private. Öffentliche Attribute und Methoden sind auch außerhalb der Klasse, in der sie deklariert wurden, sichtbar. Private sind nur innerhalb der Klasse sichtbar, in der sie deklariert wurden, andere Klassen haben keinerlei Kenntnis dieser Elemente.

Private Attribute und Methoden sind "interne Details" der Klasse. Öffentliche Methoden sind die "Bedienungselemente" der Klasse.

Öffentliche Elemente werden im Klassendiagramm mit dem "+"-Zeichen versehen, private mit einem "-"-Zeichen.

In der Regel werden in einer Klassenmodellierung alle Attribute als „privat“ deklariert werden. Der direkte Zugriff sowohl lesend als auch schreibend ist zwar möglich, aber meist nicht begründbar (z. B. aufgrund des Laufzeitverhaltens) und bleibt besser die Ausnahme.

Private Attribute werden dadurch zugreifbar gemacht, dass ein paar sogenannter öffentlicher Zugriffsmethoden, die die Vor- und Nachnamen ändern bzw. auslesen können, definiert werden.

Es werden unterschieden die schreibenden Zugriffsmethoden ("setter"-Methoden) und die lesenden Methoden ("getter"). Die Namen dieser Methoden sind wiederum prinzipiell frei wählbar, es wird jedoch aus Konventionsgründen empfohlen, die setter-Methode wie hier gezeigt mit set...(), die getter-Methode mit get...() zu benennen.

Es mag sich nun der eine oder andere Leser die Frage stellen: was bringt das Ganze eigentlich, außer zusätzlichen Methoden und damit Schreibarbeit? Die Zugriffsmethoden bieten einen idealen Ansatzpunkt für Plausibilitätskontrollen: Die Änderung eines Attributs wird nur von der Klasse selbst kontrolliert und erlaubt.

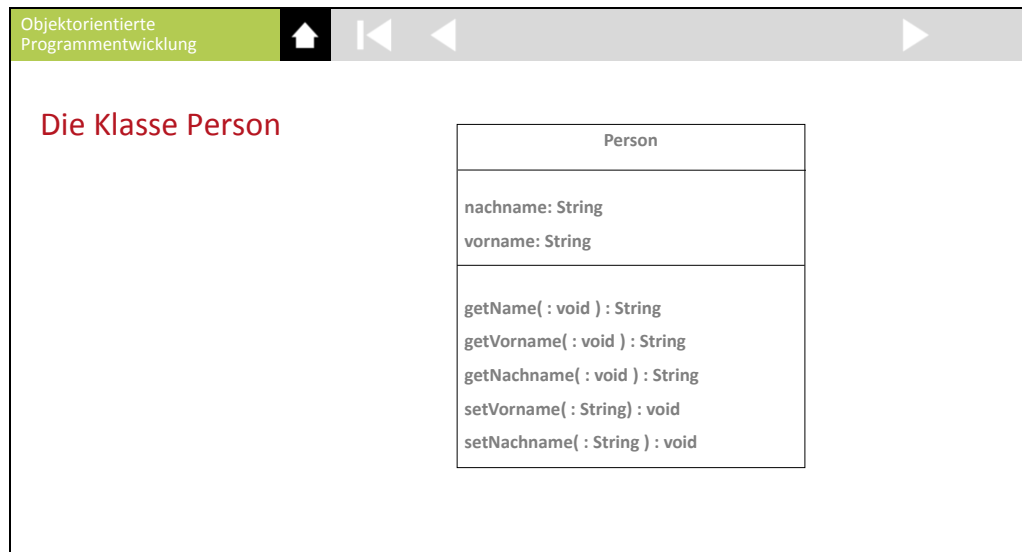


Abb. 4-10: UML: Person

Klassen sind für sich selbst verantwortlich.

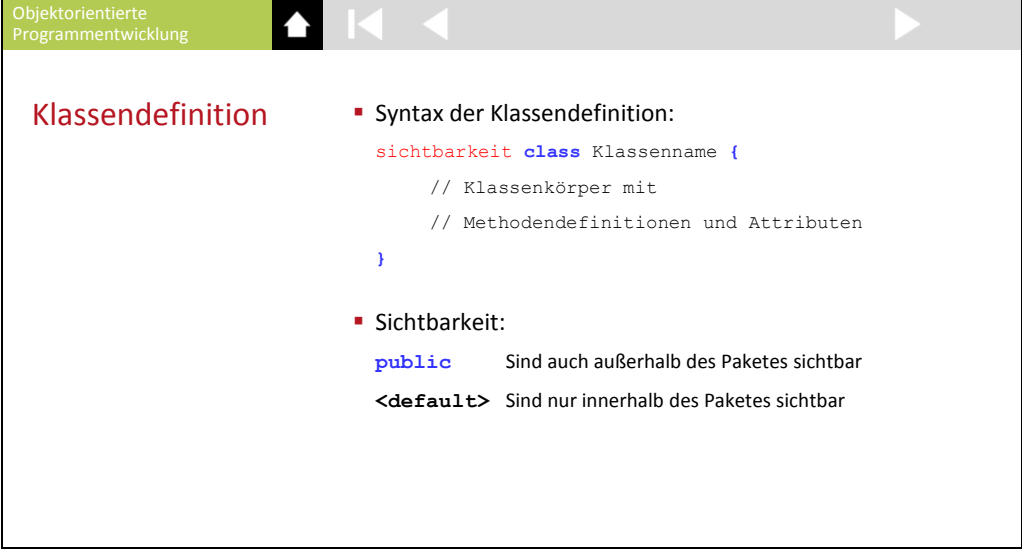
4.4.3 Zugriffsrechte und Deklarationen

Für Klassen, Attribute und Methoden wird die Sichtbarkeit (Zugriffsrecht) festgelegt. Jedem einzelnen Teil wird bei der Deklaration eine Sichtbarkeit zugewiesen. Dazu stehen die Schlüsselwörter `public` und `private` zur Verfügung.

Daneben ist noch die Paketsichtbarkeit gegeben: Erfolgt eine Deklaration ohne Verwendung des Schlüsselwortes `public` oder `private` ist das Attribut bzw. die Methode nur innerhalb des Paketes sichtbar.

4.4.4 Klassendefinition

Die Definition einer Klasse erfolgt durch das Schlüsselwort `class`. Auch Klassen können insgesamt gekapselt werden.



The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung'. The slide title is 'Klassendefinition'. It contains two bullet points: 'Syntax der Klassendefinition:' followed by a code snippet, and 'Sichtbarkeit:' followed by two lines of text. The code snippet shows the general syntax for a class definition with comments. The visibility section explains the difference between 'public' and '<default>' access modifiers.

```
sichtbarkeit class Klassenname {  
    // Klassenkörper mit  
    // Methodendefinitionen und Attributen  
}
```

▪ Sichtbarkeit:

`public` Sind auch außerhalb des Paketes sichtbar

`<default>` Sind nur innerhalb des Paketes sichtbar

Abb. 4-11: Klasse

Klassenname:

Namen beginnen per Konvention mit einem Großbuchstaben

Quellcode:

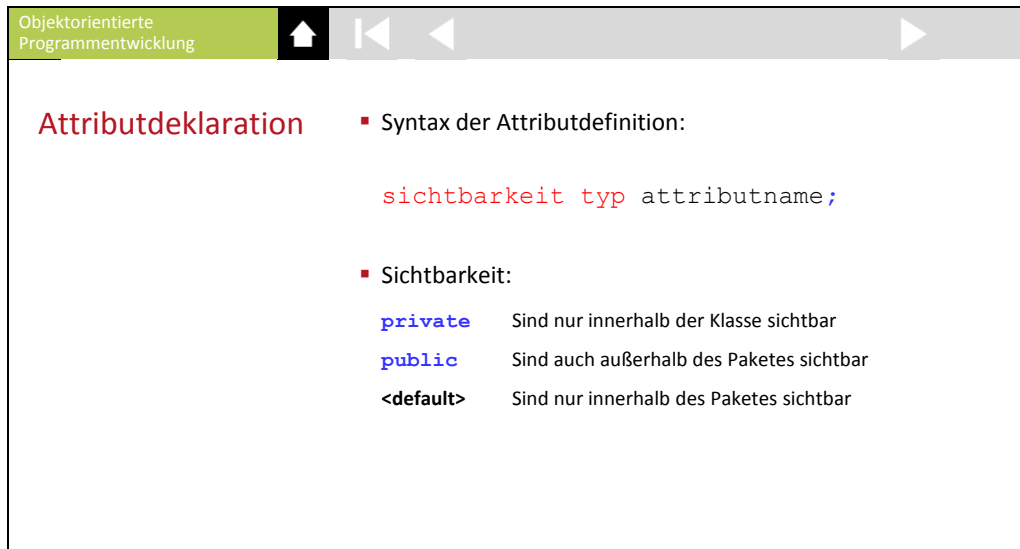
Eine Klasse muss in einer Datei exakt gleichen Namens gespeichert werden (Groß-/Kleinschreibung beachten). Die Klasse `Person` also in einer Datei namens `Person.java`.

4.5 Attribute

Mit Hilfe von Attributen können somit die Eigenschaften für die Klasse festgelegt werden.

4.5.1 Attributdeklaration

Attribute werden innerhalb des geschweiften Klammern Paares der Klassendefinition deklariert:



The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung'. The slide title is 'Attributdeklaration'. It contains two bullet points: 'Syntax der Attributdefinition:' followed by the code 'sichtbarkeit typ attributname;' and 'Sichtbarkeit:' followed by a list of visibility modifiers: 'private' (Sind nur innerhalb der Klasse sichtbar), 'public' (Sind auch außerhalb des Paketes sichtbar), and '<default>' (Sind nur innerhalb des Paketes sichtbar).

Attributdeklaration

- Syntax der Attributdefinition:
`sichtbarkeit typ attributname;`
- Sichtbarkeit:
 - `private` Sind nur innerhalb der Klasse sichtbar
 - `public` Sind auch außerhalb des Paketes sichtbar
 - `<default>` Sind nur innerhalb des Paketes sichtbar

Abb. 4-12: Attributdeklaration

Attributname:

Namen beginnen per Konvention mit Kleinbuchstaben.

4.6 Methoden (Funktionalitäten)

Java unterscheidet nicht zwischen Funktionen und Methoden. Da Funktionen nicht global (außerhalb von Klassen) definiert werden können, sind die Begriffe Funktion und Methode in Java synonym.

4.6.1 Methodendeklaration

Methoden müssen einen Typ haben, dies entspricht dem Typ des return-Wertes. Methoden, die keinen Rückgabewert haben (Prozeduren) werden als void deklariert.

Methoden erwarten in einem runden Klammerpaar durch Kommas getrennt die typisierte Liste der übergebenen Parameter. Die Klammern müssen auch bei leerer Parameterliste unbedingt mit angegeben werden.

Die Funktionalität der Methode wird innerhalb eines geschweiften Klammerpaares nach der Methodendeklaration direkt innerhalb der Klasse definiert. Typ, Name und Parameterliste einer Methode werden als Methodensignatur bezeichnet.

Der Methodenablauf endet durch die explizite Anweisung return oder an der schließenden geschweiften Klammer.

Objektorientierte Programmentwicklung

Methoden (Funktionalitäten)

- Syntax der Methodendefinition:

```
sichtbarkeit returntyp methodenname (typ1 par1, typ2 par2, ... )  
{ // methodenblock (-körper)  
    mit lokalen Variablen  
    und Anweisungen  
}
```
- Sichtbarkeit:
 - private** Sind nur innerhalb der Klasse sichtbar
 - public** Sind auch außerhalb des Paketes sichtbar
 - <default>** Sind nur innerhalb des Paketes sichtbar

Abb. 4-13: Deklaration Methode

Methodenname:

Namen beginnen per Konvention mit einem Kleinbuchstaben

Beispiel:

Da wir unsere Klasse Person allgemein zur Verfügung stellen wollen, deklarieren wir im Folgenden die Klasse Person als public.

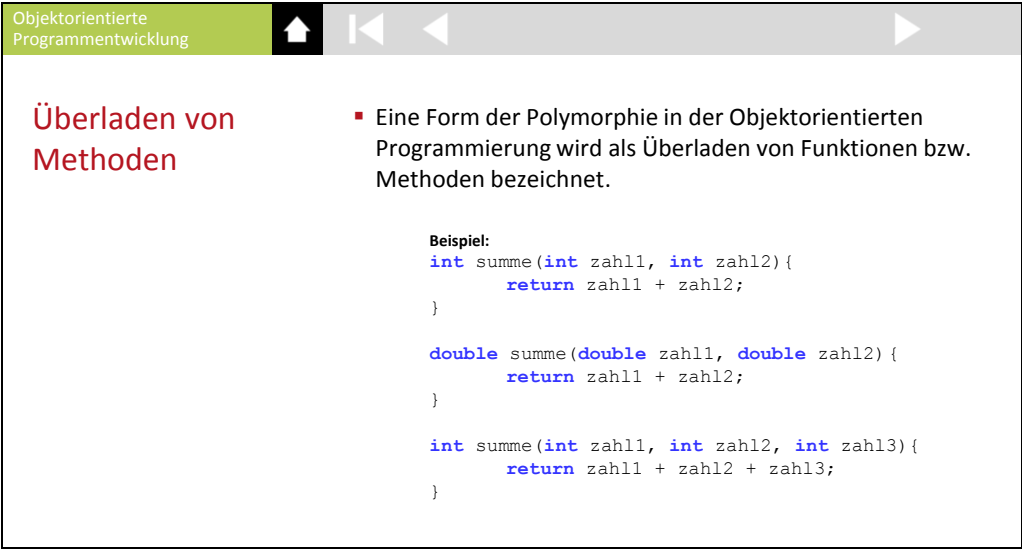
```
public class Person {  
    private String nachname;  
    private String vorname;  
    // Methoden:  
    public String getName() {  
        return nachname + ", " + vorname;  
    }  
    public void setNachname(String name) {  
        this.nachname = name;  
    }  
    ...  
}
```

Dieser Quellcode wird unter dem Namen Person.java abgespeichert.

4.6.2 Überladen von Methoden

Eine Form der Polymorphie in der Objektorientierten Programmierung wird als Überladen von Funktionen bzw. Methoden bezeichnet. Hierbei existieren mehrere Methoden mit gleichem Namen. Zur Unterscheidung wird die Parameterliste herangezogen.

Gleichnamige Methoden müssen sich in der Anzahl oder dem Datentyp der Parameter (oder in beiden Kriterien) unterscheiden.



Objektorientierte Programmentwicklung

Überladen von Methoden

- Eine Form der Polymorphie in der Objektorientierten Programmierung wird als Überladen von Funktionen bzw. Methoden bezeichnet.

Beispiel:

```
int summe(int zahl1, int zahl2) {  
    return zahl1 + zahl2;  
}  
  
double summe(double zahl1, double zahl2) {  
    return zahl1 + zahl2;  
}  
  
int summe(int zahl1, int zahl2, int zahl3) {  
    return zahl1 + zahl2 + zahl3;  
}
```

Abb. 4-14: Überladen

4.6.3 Die main-Methode

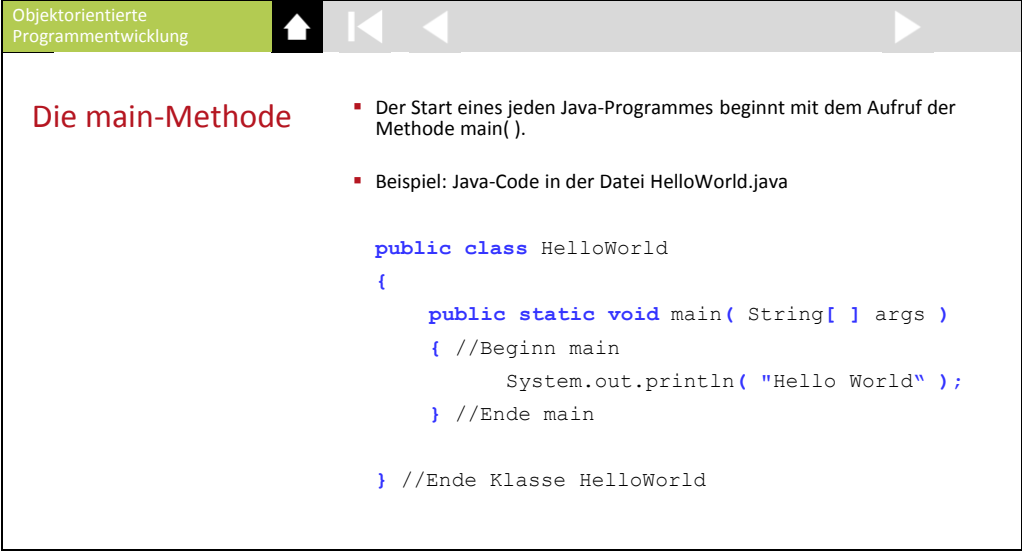
Der Start eines jeden Java-Programms beginnt mit dem Aufruf der Methode `main()`.

Die `main`-Methode muss ebenfalls in eine Klasse eingebettet sein und folgende Signatur haben:

```
public static void main(String[] args) {}
```

void	Typ der <code>main</code> -Methode. Sie hat keinen Rückgabewert.
static	Die <code>main</code> -Methode muss eine Klassenmethode sein, die schon vor dem Anlegen von Instanzen der Klasse verfügbar ist.
String[] args	Der Übergabeparameter an <code>main</code> ist ein Array aus Strings. Dieser wird durch die auf der Kommandozeile beim Interpreter Aufruf mit angegebenen Argumenten gefüllt. Der Name des Arrays (hier <code>args</code> genannt) ist frei wählbar.

Die Klasse, die die Methode `main` enthält, nennt man eine ausführbare Java-Klasse. Der Aufruf der `main`-Methode erfolgt implizit durch Starten der VM mit der ausführbaren Klasse.



The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung' and navigation icons. The slide title is 'Die main-Methode'. It contains two bullet points: 'Der Start eines jeden Java-Programmes beginnt mit dem Aufruf der Methode `main()`.' and 'Beispiel: Java-Code in der Datei `HelloWorld.java`'. Below the bullet points is a code block showing the Java code for the `HelloWorld` class.

```
public class HelloWorld
{
    public static void main( String[ ] args )
    { //Beginn main
        System.out.println( "Hello World" );
    } //Ende main

} //Ende Klasse HelloWorld
```

Abb. 4-15: `main`

Diese ausführbare Klasse wird von der Kommandozeile wie folgt gestartet:

```
java HelloWorld
```

4.6.4 Die Methode println()

Monitorausgaben werden mit der Methode println() realisiert.

Die Methode println() ist mehrfach überladen und somit können unterschiedliche Datentypen auf die Standardausgabe geschickt werden.

Dabei kann die Ausgabe über den Datentyp der Übergabeparameter gesteuert werden.

Eine besondere Rolle spielt dabei der +-Operator, er kann für verschiedenen Datentypen angewendet werden.

The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung' and navigation icons. The slide title is 'Die Methode println()' in red. A bullet point states: 'Monitorausgaben werden mit der Methode println() realisiert.' Below this, under the heading 'Beispiel:', four code snippets are shown with their corresponding output comments:

```
System.out.println(1 + 2);           // Ausgabe: 3

System.out.println( "Erg=" + 1 + '2' ); // Ausgabe: Erg=12

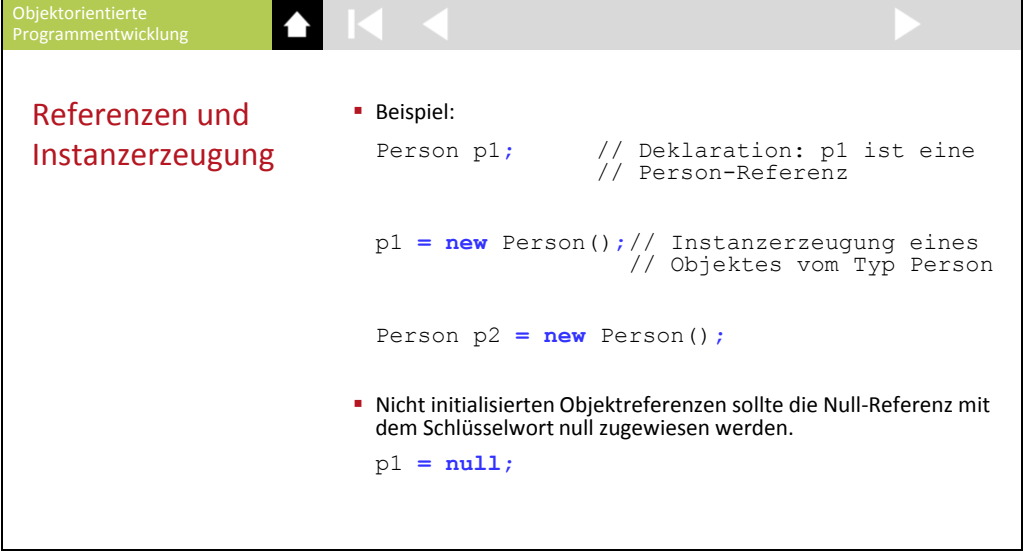
System.out.println( "1" + 2 );       // Ausgabe: 12

System.out.println( 1 + '2' );       // Ausgabe: 51
```

Abb. 4-16: println

4.7 Referenzen und Instanz Erzeugung

Soll vom Typ der Klasse Person ein Objekt instanziiert werden, so geschieht das mit dem Operator new (ein Schlüsselwort), gefolgt vom Klassennamen und einem runden Klammern Paar:



The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung' and navigation icons. The slide title is 'Referenzen und Instanzerzeugung'. It contains two bullet points with code examples. The first bullet point, 'Beispiel:', shows the declaration of a Person reference and the creation of a new Person object. The second bullet point states that uninitialized object references should be set to null and provides the code 'p1 = null;'.

```
Person p1;           // Deklaration: p1 ist eine
                     // Person-Referenz

p1 = new Person();   // Instanzerzeugung eines
                     // Objektes vom Typ Person

Person p2 = new Person();

Nicht initialisierten Objektreferenzen sollte die Null-Referenz mit
dem Schlüsselwort null zugewiesen werden.

p1 = null;
```

Abb. 4-17: Instanz Erzeugung

p1 und p2 sind Referenzen auf die eigentlichen Objekte, nicht die Objekte selbst. Der new-Operator macht somit zwei Dinge:

- new legt im Speicher das konkrete Objekt an und
- liefert die Referenz auf das eben erzeugte Objekt zurück.

4.7.1 Zugriff auf Attribute und Methoden

Java ist eine rein objektorientierte Programmiersprache, d. h. alle Attribute und Methoden sind nur im Zusammenhang mit einem konkreten Objekt sinnvoll. Beim Ansprechen von Attributen und Methoden wird der Punktoperator verwendet.

The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung'. Below the header, the title 'Zugriff auf Attribute und Methoden' is displayed in red. The slide content includes a bulleted list with two items: 'Punktoperator' and 'Die this-Referenz'. Under 'Punktoperator', the syntax is shown as 'referenzname.attributname;' and 'referenzname.methodenname();'. Under 'Die this-Referenz', a paragraph explains that the 'this' reference represents the current object within a method and can be used to access class attributes and methods.

Zugriff auf Attribute und Methoden

- Punktoperator

Syntax:

```
referenzname.attributname;  
referenzname.methodenname( );
```

- Die this-Referenz

Die Referenz `this` repräsentiert innerhalb einer Methode das aktuell gültige Objekt und kann für den Zugriff auf die Attribute und Methoden der Klasse verwendet werden.

Abb. 4-18: Attribute und Methoden

Um beispielsweise die Methode `getName()` aufrufen zu können, ist die Verwendung einer Objektreferenz notwendig. Java verwendet den Punkt als Operator zwischen Objekt und Methode.

Im ersten Beispiel wird mit der Referenz `p1` die Methode `getName()` aufgerufen:

```
p1.getName( );
```

Im zweiten Beispiel wird das Klassenattribut `nachname` der Person, ebenfalls eine Referenz, auf die Zeichenkette "Metzger" gesetzt.

```
p1.setNachname("Metzger");
```

4.7.2 Die this-Referenz

Methoden einer Klasse werden, wie eben beschrieben, mit einer Objektreferenz aufgerufen. Diese Information wird als „verborgener Parameter“ an die Methode übergeben und kann mit dem Schlüsselwort `this` angesprochen werden.

Die Referenz `this` repräsentiert stets das aktuell gültige Objekt und kann für den Zugriff auf die Attribute und Methoden der Klasse verwendet werden.

Die `this`-Referenz wird implizit verwendet. Deshalb kann `this` weggelassen werden, falls keine Namenskonflikte (z. B. mit gleichnamigen lokalen Variablen oder Übergabeparametern) vorhanden sind.

Aus Gründen der Klarheit wird `this` häufig auch angegeben ohne dass eine Notwendigkeit besteht.

Notwendig wird `this` bei Namensgleichheit mit lokalen Variablen und wenn das gesamte Objekt angesprochen werden soll.

Beispiel:

In der Methode `setZuname()` wird die `this`-Referenz notwendig, da eine Namensgleichheit zwischen Attribut und Übergabeparameter besteht.

```
public class Person {  
    // Attribute:  
    private String nachname;  
    private String vorname;  
  
    // Methoden:  
    public String getName()  
    {  
        return this.vorname + " " + this.nachname;  
        // hier kann this weggelassen werden  
    }  
  
    public void setZuname( String nachname )  
    {  
        this.nachname = nachname;  
        /* hier wird ohne this der  
        Übergabeparameter angesprochen */  
    }  
}
```

4.7.3 Formen von Objektreferenzen

Wir haben somit explizit bereits drei mögliche Objektreferenzen kennen gelernt: Benannt, null und this. Genauer sind es jedoch vier. Bei der Deklaration der Methode `getName()` wird der Rückgabotyp `String` deklariert, einfach durch die Kodierung `String getName()`. Der Rückgabewert ist folglich ebenfalls eine Referenz vom Typ `String`. Aber welchen internen Namen hat diese Referenz? Um die Problematik anschaulicher zu machen, erweitern wir die Methode zu:

```
public String getName( ){  
    String lTest = this.nachname + ", " + this.vorname;  
    return lTest;  
}
```

So geschrieben ist der interne Name klar, er lautet `lTest`. Die Lokale Variable `lTest` wird aber bereits vom Stack entfernt, wenn die Funktion terminiert. Der Rückgabewert kommt beim Aufrufer als anonyme Referenz an.

Die vier möglichen Formen von Referenzen:

Benannte Objektreferenzen:

```
Objekttyp objektReferenzName = new Objekttyp ();
```

Nichtgültige Objektreferenz:

Schlüsselwort `null`

Aktuell gültige Objektreferenz:

Schlüsselwort `this`

Anonyme Objektreferenz:

return-Werte und
Stringkonstanten `"Berlin"`

4.7.4 Referenzen im Speicher der Virtuellen Maschine

Im Folgenden noch ein Bild, das auf einen Blick den zentralen Begriff dieses Kapitels darstellen soll: Objektreferenzen und ihre Bedeutung in Java.

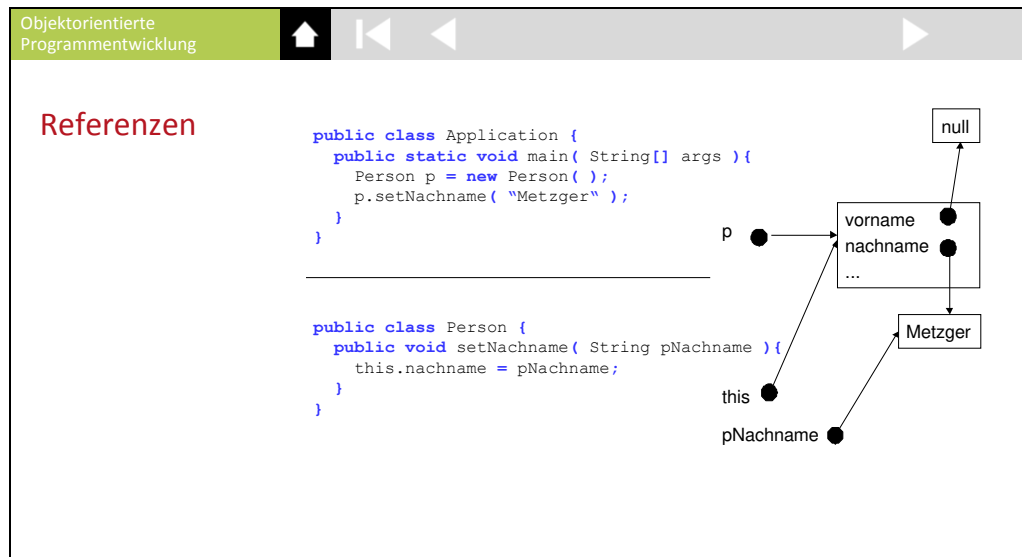


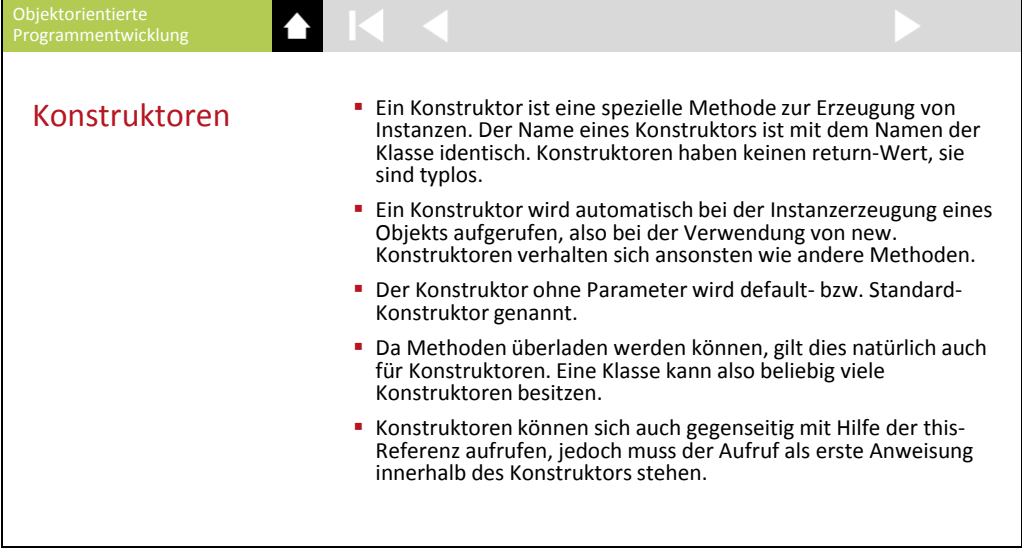
Abb. 4-19: Referenzen

Im linken Teil sind die beiden Klassen `Application` und `Person` dargestellt, im rechten Teil das entsprechende Bild im Hauptspeicher des Rechners. Die Referenzen `p`, `this` und `pNachname` befinden sich in unterschiedlichen Methoden-Frames, referenzieren aber gleiche Objekte im Hauptspeicher. Es existieren während des gesamten Programmlaufs exakt zwei Objekte, eines vom Typ `Person`, das andere vom Typ `String`. Innerhalb des Methoden-Frames der `main`-Methode der Klasse `Application` ist nur die Objektreferenz `p` bekannt, die auf das Objekt vom Typ `Person` zeigt. Die Klasse `Person` spricht dasselbe Objekt mit der `this`-Referenz an. Das Objekt vom Typ `String` wird in der Klasse `Application` anonym erzeugt und als Parameter der `setNachname(String)`-Methode übergeben. Innerhalb der Klasse `Person` ist der Zugriff möglich über die Parameterreferenz `pNachname` bzw. nach der Zuweisung auch noch über `this.nachname`.

Die Darstellung dieses eigentlich simplen Sachverhaltes ("Der Nachname einer Person wird mit "Metzger" belegt") mit Referenzpfeilen ist sicherlich gewöhnungsbedürftig. Komplexere Sachverhalte werden in dieser Darstellung jedoch sehr schnell anschaulich und verständlich.

4.8 Konstruktoren

Bei der Instanz Erzeugung von Objekten einer Klasse mit Hilfe des Operators `new` muss intern Speicherplatz allokiert werden und die Attribute auf den jeweiligen Initialwert gesetzt werden. Hier stellt sich die Frage, wer denn die Attribute bei der Instanziierung auf die default-Werte setzt. Ebenfalls deuten die runden Klammern hinter dem obligatorischen Datentyp bei `new` darauf hin, dass hier eine Methode aufgerufen wird.



The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung' and navigation icons. The slide title 'Konstruktoren' is in red. It contains a bulleted list of five points about constructors in Java.

- Ein Konstruktor ist eine spezielle Methode zur Erzeugung von Instanzen. Der Name eines Konstruktors ist mit dem Namen der Klasse identisch. Konstruktoren haben keinen return-Wert, sie sind typlos.
- Ein Konstruktor wird automatisch bei der Instanzerzeugung eines Objekts aufgerufen, also bei der Verwendung von `new`. Konstruktoren verhalten sich ansonsten wie andere Methoden.
- Der Konstruktor ohne Parameter wird default- bzw. Standard-Konstruktor genannt.
- Da Methoden überladen werden können, gilt dies natürlich auch für Konstruktoren. Eine Klasse kann also beliebig viele Konstruktoren besitzen.
- Konstruktoren können sich auch gegenseitig mit Hilfe der `this`-Referenz aufrufen, jedoch muss der Aufruf als erste Anweisung innerhalb des Konstruktors stehen.

Abb. 4-20: Konstruktor

Diesen aktiven Vorgang bezeichnen wir im Folgenden als den Aufruf des sogenannten default-Konstruktors einer Klasse. Jede Klasse hat einen Konstruktor, der ohne Argument aufgerufen werden kann. Dieser default-Konstruktor, auch Standard Konstruktor genannt, wird automatisch vom Compiler erzeugt und kann mit Hilfe des `new`-Operators implizit angesprochen werden.

```
Person p1 = new Person( );
```

`Person()` kann als der Konstruktor Aufruf aufgefasst werden. Wenn `Person()` ein Konstruktor ist, können wir diesen Begriff nun aber verallgemeinern.

Ein Konstruktor ist eine spezielle Methode zur Erzeugung von Instanzen. Der Name eines Konstruktors ist mit dem Namen der Klasse identisch. Der Konstruktor selbst ist typlos. Bei der Deklaration eines Konstruktors darf auch das Schlüsselwort `void` nicht als Typ verwendet werden.

Sobald die Entwicklerin mit dem vom Compiler erzeugten Standard Konstruktor nicht zufrieden ist, kann sie durch Implementierung eines Konstruktors den Systemteil erweitern.

4.8.1 Konstruktordeklaration

Syntax der Konstruktordefinition:

```
sichtbarkeit Klassenname (parameter)
{
    // Konstruktorkörper
}
```

Sichtbarkeit:

private Sind nur innerhalb der Klasse sichtbar
public Sind auch außerhalb der Klasse sichtbar
<default> Sind innerhalb des Paketes sichtbar

Methodenname:

Name = Klassenname

Beispiel:

```
public class Person {  
    private String nachname;  
    private String vorname;  
  
    // parameterloser Konstruktor  
    // default-Konstruktor  
    public Person() {  
        System.out.println("Geburt einer Person!");  
    }  
  
    public void setNachname(String nachname) {  
        this.nachname = nachname;  
    }  
  
    public String getNachname() {  
        return this.nachname;  
    }  
  
    public String getName() {  
        return this.getNachname( ) + ", "  
            + this.getVorname();  
    }  
}
```

Ein Konstruktor wird automatisch bei der Instanziierung eines Objekts aufgerufen, also bei der Verwendung von `new`. Constructoren verhalten sich ansonsten wie andere Methoden. Auch innerhalb von Constructoren kann auf die Attribute und anderen Methoden der Klasse zugegriffen werden.

In diesem Beispiel wird einfach eine Meldung auf die Konsole geschrieben.

4.8.2 Überladen von Konstruktoren

Bisher haben wir den Konstruktor ohne Parameter, den sogenannten default-Konstruktor verwendet.

Da Methoden überladen werden können, gilt diese Form des Polymorphismus auch für Konstruktoren. Eine Klasse kann also beliebig viele Konstruktoren besitzen, die sich in der Parameterliste unterscheiden müssen.

Beispiel:

```
public class Person {  
    private String nachname;  
    private String vorname;  
  
    // Parameterloser Konstruktor  
    //(default-Konstruktor)  
    public Person( ) {  
        System.out.println("Geburt einer Person!");  
    }  
  
    // Konstruktor mit zwei Parametern  
    public Person(String vn, String nn) {  
        this.vorname = vn;  
        this.setNachname(nn);  
    }  
  
    public void setNachname(String nachname) {  
        this.nachname = nachname;  
    }  
  
    public String getNachname() {  
        return this.nachname;  
    }  
    public String getName(){  
        return this.nachname + ", " + this.getVorname();  
    }  
}
```

Konstruktoren können sich auch gegenseitig mit Hilfe der `this`-Referenz aufrufen, jedoch muss der Aufruf als erste Anweisung innerhalb des Konstruktors stehen:

Beispiel:

```
public class Person {  
    private String nachname;  
    private String vorname;  
  
    // default-Konstruktor (ohne Parameter)  
    public Person() {  
        System.out.println ("Geburt einer Person!");  
    }  
  
    // Konstruktor mit zwei Parametern  
    public Person(String vn, String nn) {  
        this(); // Aufruf des default-Konstruktors  
        this.vorname = vn;  
        this.setNachname(nn);  
    }  
  
    public void setNachname(String nachname) {  
        this.nachname = nachname;  
    }  
  
    public String getNachname() {  
        return this.nachname;  
    }  
    public String getName(){  
        return this.nachname + ", " + this.getVorname();  
    }  
}
```

Der Aufruf des parameterlosen Konstruktors aus einem anderen Konstruktor erhöht die Sicherheit und Wartbarkeit der Klasse und vermeidet Redundanz. Falls während der Modellierung der Klasse neue Attribute initialisiert werden müssen oder die default-Werte geändert werden, muss dies nur an einer zentralen Stelle, nämlich im parameterlosen Konstruktor erfolgen.

An dieser Stelle sei nochmals eindringlich auf drei Punkte hingewiesen:

Ein Konstruktor hat keinen Rückgabewert, nicht einmal void. Wird ein Rückgabewert deklariert, wird diese Methode als normale Methode betrachtet und somit nicht automatisch bei der Instanziierung aufgerufen.

Wird kein Konstruktor für die Klasse programmiert, wird bei der Instanziierung durch die Virtuelle Java-Maschine der vom System generierte default-Konstruktor benutzt, der alle Attribute der Klasse auf binär Null setzt. Dieser Konstruktor wird jedoch für aufrufende Klassen nicht mehr automatisch zur Verfügung gestellt, sobald auch nur ein einziger Konstruktor selbst programmiert wurde. Wird neben einem Konstruktor mit Übergabeparametern auch ein parameterloser Konstruktor benötigt, muss dieser dann explizit geschrieben werden.

Der vom System generierte Konstruktor Teil wird immer aufgerufen und ausgeführt. Eigene Konstrukturen ergänzen den default-Konstruktor und werden zeitlich nach ihm ausgeführt. Im eigenen Konstruktor ist somit die this-Referenz sauber definiert.

4.8.3 Private Konstrukturen

Wir haben in den obigen Beispielen gesehen, dass eine Menge von Konstrukturen implementiert werden kann. In diesem Zusammenhang können wir auch private Konstrukturen einführen. Ein privater Konstruktor kann nur innerhalb der deklarierenden Klasse angesprochen werden.

Somit ist es möglich, z. B. im parameterlosen Konstruktor die default-Initialisierung durchzuführen und diesen zentral aus allen anderen Konstrukturen heraus aufzurufen. Ist nun diese parameterlose Instanziierung aufgrund der Klassenmodellierung nicht erwünscht, kann dieser Konstruktor auf private gesetzt werden und ist somit für eine andere Klasse nicht aufrufbar.

4.9 Klassenattribute und Klassenmethoden

4.9.1 Klassenattribute

In manchen Situationen ist es wünschenswert, Bestandteile einer Klasse zu besitzen, die nicht Instanz bezogen, sondern klassenbezogen sind, d. h. unabhängig von der Anzahl der Instanzen nur einmal pro Klasse existieren (Koordinatenursprung, Mehrwertsteuersatz und ähnliches). Mit Hilfe des Schlüsselwortes `static` wird definiert, dass ein Attribut nur einmal pro Klasse existiert.

The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung' and navigation icons. The slide title is 'Klassenattribute und Klassenmethoden'. It contains two bullet points: 1. 'Klassenattribute sind nicht instanzbezogen und existieren nur ein Mal pro Klasse.' followed by the syntax 'sichtbarkeit static typ name = Wert;'. 2. 'Klassenmethoden werden oft als Zugriffsmethoden für Klassenattribute verwendet. Statische Methoden können mit dem Namen der Klasse vor dem Punkoperator aufgerufen werden.' followed by the syntax 'sichtbarkeit static returntyp methodenname(param) { // Zugriff nur auf statische Attribute der Klasse }'.

Objektorientierte Programmentwicklung

Klassenattribute und Klassenmethoden

- Klassenattribute sind nicht instanzbezogen und existieren nur ein Mal pro Klasse.
Syntax:
`sichtbarkeit static typ name = Wert;`
- Klassenmethoden werden oft als Zugriffsmethoden für Klassenattribute verwendet. Statische Methoden können mit dem Namen der Klasse vor dem Punkoperator aufgerufen werden.
Syntax:
`sichtbarkeit static returntyp methodenname(param)
{
 // Zugriff nur auf statische Attribute der Klasse
}`

Abb. 4-21: Klassenattribute und Klassenmethoden

Innerhalb der Klassendefinition kann jedoch ohne Benutzung des Klassennamens auf die statischen Attribute zugegriffen werden.

Statische Elemente existieren unabhängig von der Anzahl der Instanzen und auf sie kann auch zugegriffen werden, wenn keine Instanz existiert.

Ein Beispiel für ein statisches Attribut der Klasse `Person` ist z. B. die Anzahl der Augen, die stets für alle Menschen 2 sein sollte.

```
private static int augenAnzahl = 2;
```

Auch ein Personenzähler (Instanzzähler) wird sinnvollerweise als statisches Attribut modelliert:

```
private static int personenZaehler = 0;
```

und bei jeder Instanz Erzeugung um eins erhöht.

Da Attribute, auch statische, wegen der Datenkapselung immer privat sein sollten, erscheint es sinnvoll Methoden anzubieten, die ebenfalls ohne Instanz der Klasse aufrufbar sind.

4.9.2 Klassenmethoden

Analog zu statischen Daten werden Klassenmethoden ebenfalls durch das Schlüsselwort `static` deklariert.

Klassenmethoden werden als Zugriffsmethoden für Klassenattribute verwendet.

```
public static int getAnzahl()  
{  
    return personenZaehler;  
}
```

Ein weiteres Beispiel für eine Klassenmethode für die Klasse `Person` wäre eine Methode `getSpecies()`:

```
public static String getSpecies()  
{  
    return "Homo sapiens";  
}
```

Von außen können statische Methoden mit dem Namen der Klasse vor dem Punkoperator aufgerufen werden.

```
Person.getSpecies();
```

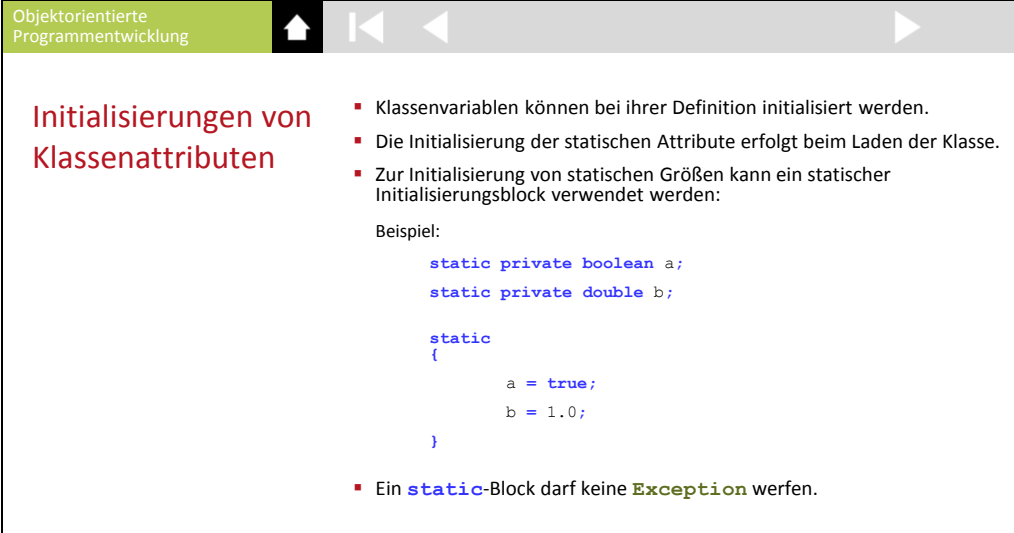
Ein wichtiger Unterschied existiert zu nichtstatischen Methoden: Klassenmethoden haben keine `this`-Referenz. Daraus folgt, dass statische Methoden nur auf statische Attribute zugreifen können.

4.9.3 Initialisierungen von Klassenattributen

Klassen- und Instanzvariablen können bei ihrer Definition initialisiert werden.

Die Initialisierung erfolgt bei statischen Variablen beim Laden der Klasse, bei den nichtstatischen beim Konstruktor Aufruf.

Zur Initialisierung von statischen Größen kann man statische Initialisierungsblöcke verwenden:



The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung' and navigation icons. The slide title is 'Initialisierungen von Klassenattributen'. It contains a bulleted list of three points, a code example, and a final bullet point.

- Klassenvariablen können bei ihrer Definition initialisiert werden.
- Die Initialisierung der statischen Attribute erfolgt beim Laden der Klasse.
- Zur Initialisierung von statischen Größen kann ein statischer Initialisierungsblock verwendet werden:

Beispiel:

```
static private boolean a;  
static private double b;  
  
static  
{  
    a = true;  
    b = 1.0;  
}
```

- Ein `static`-Block darf keine `Exception` werfen.

Abb. 4-22: Initialisierung von Klassenattributen

Es können mehrere statische Initialisierungsblöcke verwendet werden. Sie werden beim Laden der Klasse in der Reihenfolge ihres Auftretens abgearbeitet.

4.9.4 Statische Elemente und die Virtuelle Maschine

Durch die Einführung der statischen Attribute und Methoden lässt sich nun die Deklaration der main-Methode als static begründen.

Wie funktioniert die Virtuelle Maschine beim Laden einer beliebigen Klasse:

1. Der Bytecode wird durch den Bytecode-Verifier geprüft
2. Der Klassenlader lädt die Klasse in den Speicher der VM
3. Die VM legt alle statischen Attribute und Methoden exakt einmal an
4. Ein eventuell vorhandener static-Block wird abgearbeitet

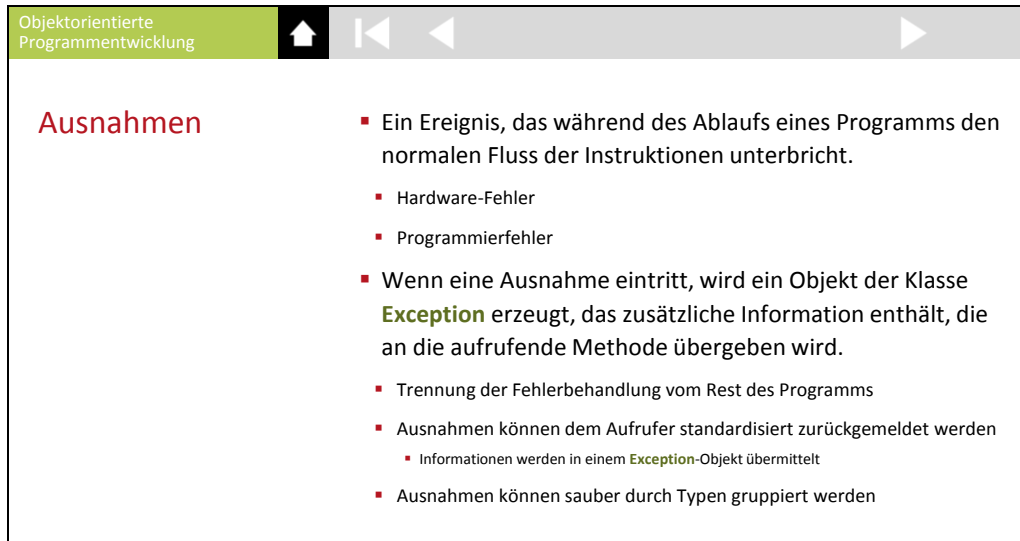
Was passiert beim Instanzieren eines Objektes?

1. Falls die Klasse noch nicht geladen wurde, wird sie geladen.
2. Die VM legt alle nicht-statischen Attribute pro Objekt exakt einmal an.
3. Es wird ein Konstruktor für dieses Objekt aufgerufen.

Was passiert beim Aufruf der VM mit dem übergebenen Klassennamen?

1. Der Klassenname wird interpretiert und die zugehörige Klasse wird geladen.
2. Die main-Methode wird innerhalb dieser Klasse gesucht und ausgeführt, ist keine Methode der Signatur `public static void main(String [] args)` vorhanden, wird eine "NoSuchMethodException" geworfen.

4.11 Ausnahmen



Objektorientierte Programmentwicklung

Ausnahmen

- Ein Ereignis, das während des Ablaufs eines Programms den normalen Fluss der Instruktionen unterbricht.
 - Hardware-Fehler
 - Programmierfehler
- Wenn eine Ausnahme eintritt, wird ein Objekt der Klasse **Exception** erzeugt, das zusätzliche Information enthält, die an die aufrufende Methode übergeben wird.
 - Trennung der Fehlerbehandlung vom Rest des Programms
- Ausnahmen können dem Aufrufer standardisiert zurückgemeldet werden
 - Informationen werden in einem **Exception**-Objekt übermittelt
- Ausnahmen können sauber durch Typen gruppiert werden

Abb. 4-23: Ausnahmen

4.11.1 Traditionelle Fehlerbehandlung

Fehlerbehandlung wird in prozeduralen Programmiersprachen häufig mit verschiedenen Strategien gelöst. Das Setzen globaler Fehlervariablen, der Aufruf zentraler Fehlerbehandlungsroutinen oder die Kodierung von Fehlermeldungen durch die Rückgabe von Fehlerstrukturen seien hier exemplarisch genannt. Alleine diese Vielfalt von Strategien kann zu schlecht wartbaren Programmen führen. Aber auch die syntaktischen Möglichkeiten sind häufig so beschränkt, dass bei der Programmierung die potentiellen Fehlerquellen alle gesondert berücksichtigt werden müssen. Ein, sicherlich extrem kodiertes, trotzdem aber exemplarisches Beispiel ist folgendes:

Der einfache Pseudocode für „Lesen einer Datei“:

**Ausnahme-
behandlung: Beispiel**

- Pseudocode zum Lesen einer Datei ohne Fehlerbehandlung

```
readFile {  
    Öffne die Datei;  
  
    Bestimme die Größe;  
  
    Belege Speicher;  
  
    Lies die Datei in den Speicher;  
  
    Schließe die Datei;  
}
```

Abb. 4-24: Beispiel

wächst zu folgendem Programm:

**Ausnahmen:
Traditionelle
Fehlerbehandlung**

```
FehlerTyp readFile {  
    //initialisiere Fehlerwert =0;  
    //Öffne die Datei...  
    if (Datei ist offen) {  
        //Bestimme die Größe;  
        if (Größe bestimmt) {  
            //Belege Speicher;  
            if(Speicher belegt) {  
                Lies die Datei in den Speicher;  
                if(Fehler aufgetreten)  
                    Fehlerwert = -1;  
            } else  
                Fehlerwert = -2;  
        } else  
            Fehlerwert = -3;  
        Schließe die Datei;  
        Fehlerwert = -5;  
    }  
    return Fehlerwert;  
}
```

Abb. 4-25: Beispiel

Eine verwirrende Mischung aus Programmlogik und Fehlerprüfung.

So unübersichtlich, dass die meisten Leser wohl nicht bemerkt haben, dass die Fehlerbehandlung für den Pseudocode „Schließen einer Datei“ doch vollkommen vergessen wurde und somit in manchen Programmsituationen Dateien nicht ordnungsgemäß geschlossen werden. Wünschenswert wäre nun, dass der Compiler auf diese unsaubere Situation hinweisen könnte.

4.11.2 Die Klasse Exception

Mit Hilfe von Ausnahmen wird in Java das Auftreten und Behandeln von Fehlern wesentlich vereinfacht. Jede Methode hat die Möglichkeit, beim Auftreten eines Fehlers ein Objekt vom Typ `java.lang.Exception` zu erzeugen und der aufrufenden Methode zu übermitteln, ohne die `return`-Anweisung zu verwenden: Die Ausnahme „wird geworfen“.

Erzeugen der Ausnahme:

```
Exception e = new Exception();
```

Werfen der eben erzeugten Ausnahme:

```
throw e;
```

Oder in der üblichen verkürzten Form als anonyme Exception:

```
throw new Exception();
```

Es ist zu beachten, dass dieses geworfene Objekt vom Typ `Exception` bisher vollkommen unspezifisch ist und nur signalisieren kann „Es ist eine Ausnahme aufgetreten“.

`Exception`-Objekte können jedoch bei der Instanziierung durch die Übergabe einer Zeichenkette näher spezifiziert werden.

Eine Methode, hier z. B. eine namens „`methode()`“, die eine Ausnahme werfen kann, muss bei der Deklaration durch `throws Exception` ergänzt werden.

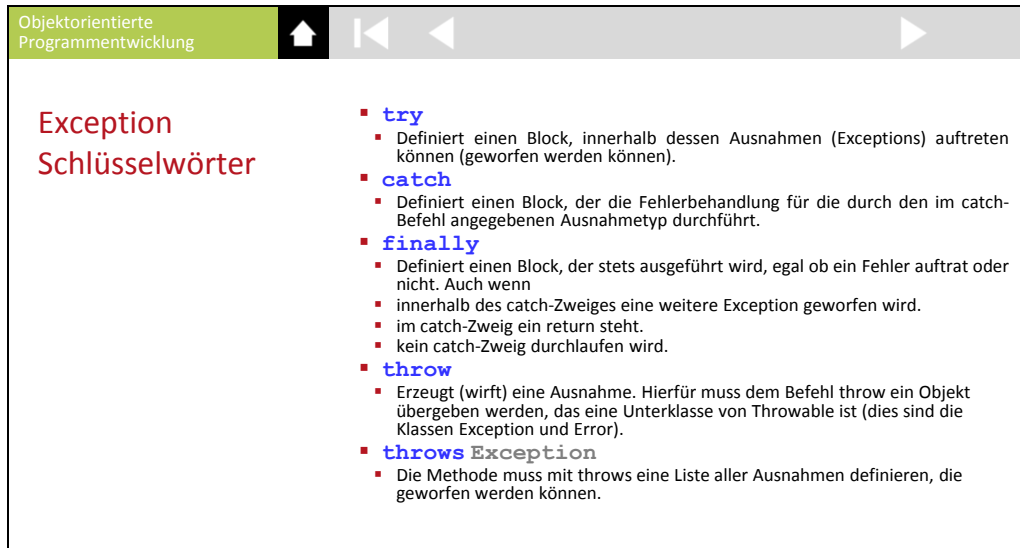
```
public void methode() throws Exception {  
    // Anweisungen  
    // darin enthalten  
    throw new Exception();  
}
```

Der große Vorteil des Werfens von Ausnahmen ist nun, dass die aufrufende Methode notwendig auf die Ausnahme reagieren muss, ein einfacher Aufruf via

```
referenz.methode();
```

führt nun zu einem Compilerfehler.

Um die Methode aufrufen zu können, muss der Aufruf in einem `try`-Block erfolgen. Ein `try`-Block wird stets von mindestens einem `catch`-Block begleitet, der in einer Fehlersituation die geworfene Ausnahme auffängt und die Fehlerbehandlungsroutine enthält. Nach dem `catch`-Zweig kann noch ein `finally`-Block stehen, der in jedem Falle durchlaufen wird.



The screenshot shows a presentation slide with a green header bar containing the text 'Objektorientierte Programmentwicklung' and navigation icons. The slide title is 'Exception Schlüsselwörter'. The content is a bulleted list of Java exception handling keywords and their uses.

Exception Schlüsselwörter

- **try**
 - Definiert einen Block, innerhalb dessen Ausnahmen (Exceptions) auftreten können (geworfen werden können).
- **catch**
 - Definiert einen Block, der die Fehlerbehandlung für die durch den im catch-Befehl angegebenen Ausnahmetyp durchführt.
- **finally**
 - Definiert einen Block, der stets ausgeführt wird, egal ob ein Fehler auftrat oder nicht. Auch wenn
 - innerhalb des catch-Zweiges eine weitere Exception geworfen wird.
 - im catch-Zweig ein return steht.
 - kein catch-Zweig durchlaufen wird.
- **throw**
 - Erzeugt (wirft) eine Ausnahme. Hierfür muss dem Befehl throw ein Objekt übergeben werden, das eine Unterklasse von Throwable ist (dies sind die Klassen Exception und Error).
- **throws Exception**
 - Die Methode muss mit throws eine Liste aller Ausnahmen definieren, die geworfen werden können.

Abb. 4-26: Ausnahmebehandlung

Allgemeiner Aufbau:

```
try {  
    referenz.methode();  
    // weitere Anweisungen  
}  
catch(ExceptionX e) {  
    System.out.println("Fehler X!");  
}  
catch(ExceptionY e) {  
    System.out.println("Fehler Y!");  
}  
catch(ExceptionZ e) {  
    System.out.println("Fehler Z!");  
}  
catch(Exception e) {  
    System.out.println("allgemeiner Fehler");  
}  
finally { // wird in jedem Fall durchlaufen  
    //Aufräumarbeiten  
}
```

In diesem Beispiel wird als Fehlerbehandlung einfach die Ausgabe „Fehler X Y Z“ simuliert.

Die hier verwendeten Klassen ExceptionX, ExceptionY und ExceptionZ existieren in der Klassenbibliothek nicht.

4.11.3 Beispiel Kehrwert

Objektorientierte Programmentwicklung

⬆ ⬅ ➡ ➦

Exception Beispiel: Kehrwert

```

public class Rechnen {
    public static void main( String [ ] args) {
        try {
            ...
            MathLib ml = new MathLib();
            double ergebnis = ml.kehrwert( a );
            System.out.println(ergebnis);
        }
        catch (Exception e) {
            System.out.println("Kehrwert nicht berechnet");
        }
    }
}

```

▪ Folgendes Beispiel (Berechnung des Kehrwerts) soll die Verwendung der Klasse Exception verdeutlichen:

```

public class MathLib {
    public double kehrwert(double d)
        throws Exception {
        if(d == 0.0)
            throw new Exception();
        else // else kann auch fehlen
            return 1.0/d;
    }
}

```

1. Aufruf: Kehrwert wird ohne Probleme berechnet (double a = 5.0;)

2. Aufruf: Kehrwert kann nicht berechnet werden (double a = 0.0;)

Abb. 4-27: Kehrwert

Besonders sei anhand dieses Beispiels auf die Vorteile der Verwendung von Exception hingewiesen:

1. Saubere Trennung zwischen Programmlogik und Fehlerbehandlungsteil im Quellcode.
2. Es werden keine willkürlichen Rückgabewerte oder globale Fehlervariablen benötigt.
3. Die aufrufende Ebene muss auf das mögliche Auftreten eines Fehlers reagieren können.

Für die flexiblere Fehlerbehandlung seien hier noch auf zwei Dinge angesprochen:

1. Beim Erzeugen einer Exception kann ein beschreibender Text mitgegeben werden, der ebenfalls eine feinere Fehlerbehandlung ermöglicht.
2. Es gibt nicht nur eine Klasse Exception, sondern eine ganze Reihe von Subklassen für spezielle Ausnahmesituationen wie z.B. `NumberFormatException`, `IOException` oder `SQLException`. Unterschiedliche catch-Blöcke können dann jeweils genau einen Exception-Typ fangen.

4.11.4 Ausnahmen mit fehlerbeschreibendem Text

Die bisherige Verwendung der Exception ist recht unkomfortabel, da außer der Aussagen "Fehler aufgetreten" keine weiteren Informationen in die aufrufende Ebene gegeben wurden. Analog der Klasse String kann auch eine Exception mit einem Text erzeugt werden, der dann über die Methode getMessage() ausgelesen werden kann.

Im vorigen Beispiel wird in der Klasse Rechnen der catch-Block durch folgenden ersetzt:

```
catch(Exception e) {  
    System.out.println( "Kehrwert nicht berechnet\n"  
        + e.getMessage() );  
}
```

Die Methode kehrwert() in der Klasse MathLib ruft jetzt einen anderen Konstruktor der Klasse Exception auf:

```
public double kehrwert(double d) throws Exception {  
    if(d == 0.0)  
        throw new Exception("Division durch Null");  
  
    return 1.0/d;  
}
```


5

Beziehungen

5.1	Assoziation	5-3
5.2	Aggregation / Komposition	5-4
5.3	Vererbung.....	5-6
5.3.1	Sichtbarkeit in der Vererbungshierarchie	5-9
5.3.2	Vererbung und Konstruktoren.....	5-9
5.3.3	Überschreiben von Attributen	5-10
5.3.4	Überschreiben von Methoden.....	5-11
5.3.5	Polymorphie.....	5-12
5.3.6	Vererbung und der Cast-Operator	5-13
5.3.7	Hierarchie der Exception-Klassen.....	5-15
5.3.8	Die RuntimeException	5-16
5.3.9	Eigene Exception-Klassen	5-16
5.4	Finale Elemente.....	5-17

5 Beziehungen

Bisher wurden Klassen und ihre Objekte als abgeschlossene, einzelne Systeme betrachtet. Klassen stehen aber nicht isoliert im Raum, sondern können Beziehungen zu anderen Klassen haben. Von diesen Beziehungen zwischen Klassen handelt dieses Kapitel.

Beziehungen zwischen Klassen können vielfältiger Natur und unterschiedlich stark sein. Einige wichtige Beziehungen sollen im Folgenden vorgestellt werden.

5.1 Assoziation

Da die Objektorientierte Programmierung nur eine Abbildung der Realität darstellt, ergibt sich die Assoziationsbeziehung aus einer Beziehung im wirklichen Leben.

Eine Formulierung wie „verwendet ein“ oder „nutzt ein“ deutet auf eine Assoziation hin.

Nehmen wir die bisherige Klasse Person und eine weitere Klasse Auto. Aus der Formulierung „Eine Person nutzt/fährt ein Auto“ ergibt sich eine Assoziationsbeziehung zwischen der Klasse Person und der Klasse Auto.

Im UML-Diagramm wird die Assoziation durch eine einfache Linie dargestellt.

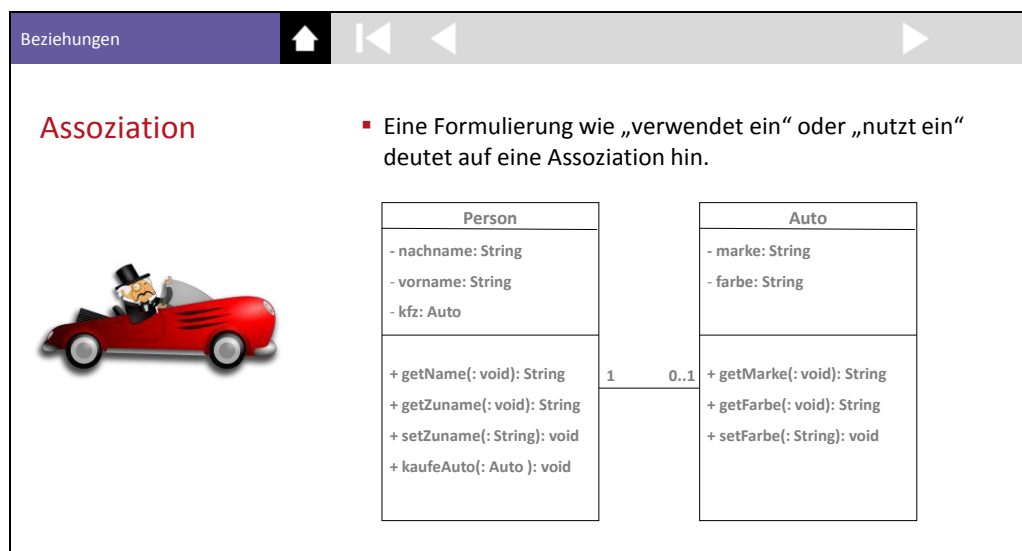


Abb. 5-1: Assoziation

Beim Instanzieren einer Person wird nicht gleich eine Instanz der Klasse Auto erzeugt.

Die abgebildete Beziehung muss nachträglich hergestellt werden. Wir benötigen also eine Methode in der Klasse Person, die einem Personenobjekt eine Instanz der Klasse Auto zuführt.

Nehmen wir eine Methode mit folgender Signatur:

```
void kaufeAuto( Auto );
```

Die übergebene Referenz eines Objektes von Typ Auto muss in der Person gespeichert werden können. Daraus folgt die Notwendigkeit eines Attributes vom Typ Auto. Dieses Attribut muss kein Array sein, da die Kardinalität 1 zu 0..1 gefordert wird.

5.2 Aggregation / Komposition

Eine Aggregationsbeziehung liegt meist dann vor, wenn von „hat ein“ die Rede ist. Das Auto hat einen Motor.

Die **Aggregation** wird im UML-Diagramm mit der offenen Raute dargestellt.

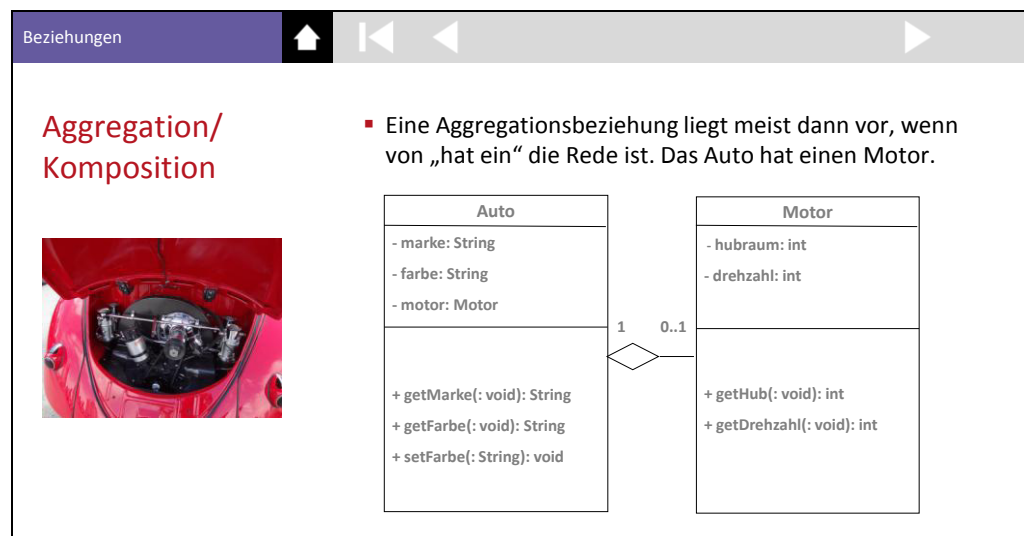


Abb. 5-2: Aggregation

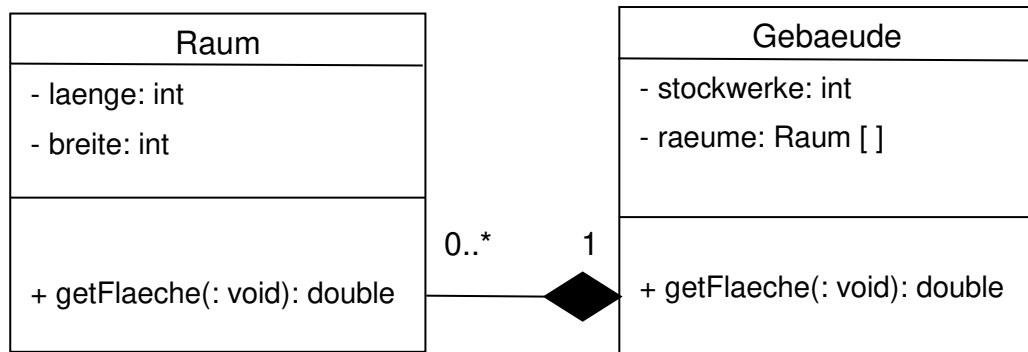
Beim Instanzieren eines Objektes Auto wird auch eine Instanz von Motor erzeugt. Dies wird durch den new Operator im Konstruktor des Autos erreicht.

Um über eine Referenz der Klasse Auto auf die privaten Daten des Motors Zugriff zu haben, werden in der Klasse Auto entsprechende Methoden angeboten.

Ein Objekt der Klasse Motor kann auch losgelöst von einer Instanz der Klasse Auto existieren.

Das ist ein Unterschied zur **Komposition**, einer der Aggregation ähnlichen Beziehung. Bei der Komposition handelt es sich ebenfalls um eine „hat ein“ Beziehung.

Die Darstellung im UML-Diagramm erfolgt mit einer geschlossenen Raute.



Ein Raum existiert nie allein, er muss Teil eines Gebäudes sein. Daraus ergibt sich die Komposition als Beziehung.

Auch kann ein Raum nicht mehreren Gebäuden zugeordnet werden.

5.3 Vererbung

Bei der Vererbung handelt es sich um die stärkste Beziehung, die die objektorientierten Programmierung zu bieten hat. Die abgeleiteten Klassen können die Daten und Funktionalitäten bereits vorhandener Klassen nutzen und darauf aufbauen.

Auf diese Weise wird die Wiederverwendbarkeit von Software unterstützt. Eigenschaften, welche schon in einer Klasse implementiert wurden, können an eine andere Klasse vererbt und müssen nicht noch einmal eingebaut werden. Daraus resultiert kleinerer Code und außerdem werden gemeinsame Eigenschaften nur an einer Stelle geprüft oder geändert.

Bei der Vererbung handelt es sich um eine sogenannte „ist ein“ Beziehung. Ein Mitarbeiter „ist eine“ Person.

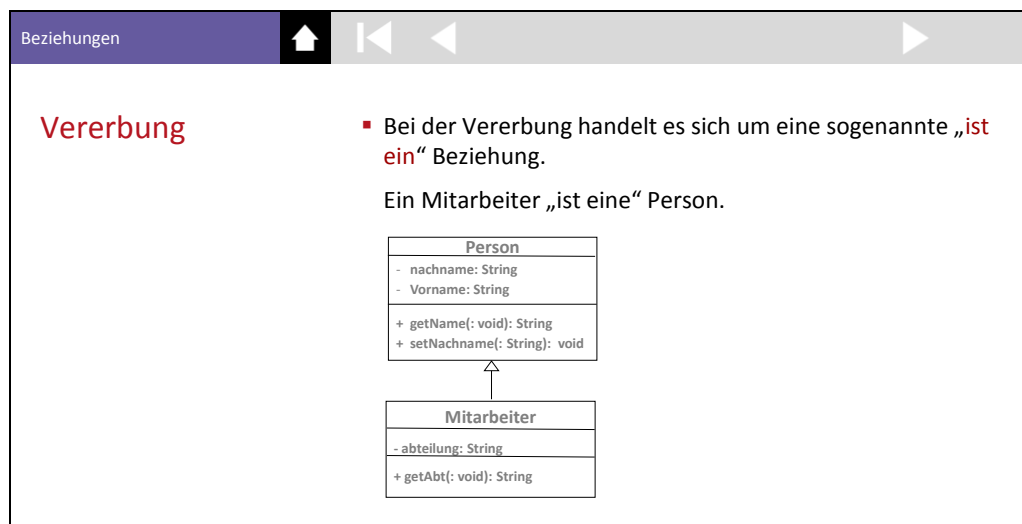
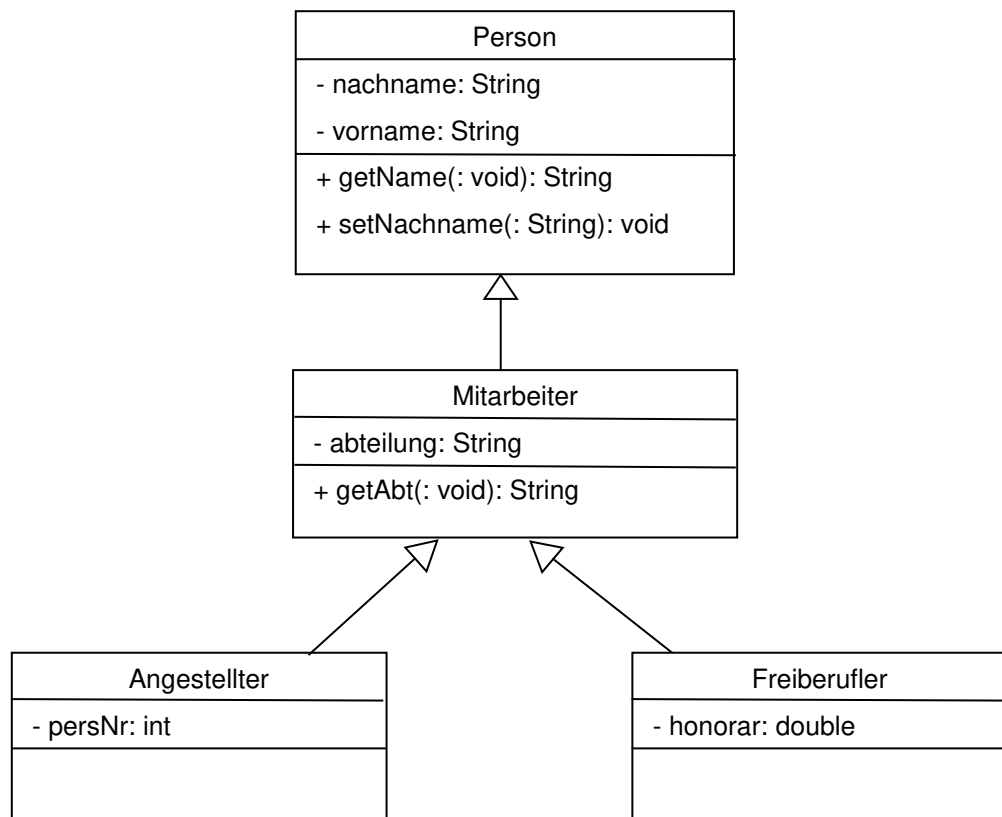


Abb. 5-3: Vererbung

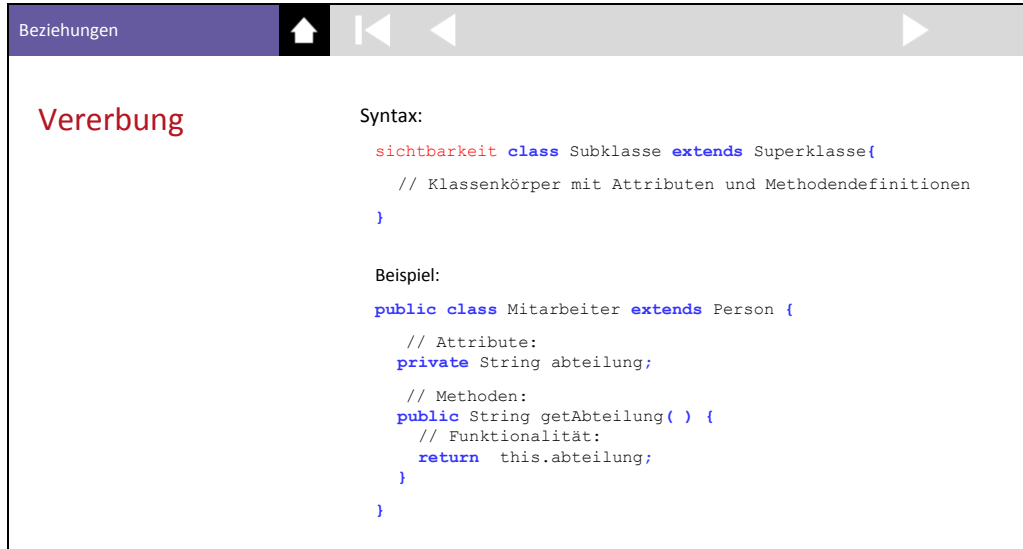
Die vererbende Klasse wird Basis-, Ober- oder Superklasse genannt. Die erbende Klasse bezeichnet man als abgeleitete Klasse oder Unter- bzw. Subklasse. Wenn von der Subklasse wieder abgeleitet wird, ergibt sich eine baumartige Struktur von miteinander in Beziehung stehender Klassen.

Im UML-Diagramm wird ein offener Pfeil in Richtung der Basisklasse als Symbol für die Vererbungsbeziehung verwendet.



Die Klasse **Person** dient als Basisklasse für **Mitarbeiter**. Oder anders formuliert: Ein **Mitarbeiter** ist eine **Person**.

Die Subklasse **Mitarbeiter** erbt alle Attribute und Methoden der Basis-klassse **Person**, mit Ausnahme der Konstruktoren.



Vererbung

Syntax:

```
sichtbarkeit class Subklasse extends Superklasse{  
    // Klassenkörper mit Attributen und Methodendefinitionen  
}
```

Beispiel:

```
public class Mitarbeiter extends Person {  
    // Attribute:  
    private String abteilung;  
  
    // Methoden:  
    public String getAbteilung() {  
        // Funktionalität:  
        return this.abteilung;  
    }  
}
```

Abb. 5-4: Vererbung

Zu beachten ist noch, dass der direkte Zugriff auf die Attribute nachname und vorname der Klasse Person aus den Subklassen nicht erlaubt ist. Die Attribute wurden als private deklariert und sind nur über die entsprechenden Zugriffsmethoden ansprechbar.

Syntax:

```
sichtbarkeit class Subklasse extends Superklasse{  
    // Klassenkörper mit Attributen und Methodendefinitionen  
}
```

Vererbung erweitert den Typ der Subklasse um den Typ der Basisklasse.

- Die Subklasse ist vom Typ Basisklasse.
Die Bedeutung der Schlüsselwörter public und private ändert sich durch die Einführung der Vererbung nicht.
- Die Subklasse hat keinen Zugriff auf die privaten Attribute und Methoden der Basisklasse.
Innerhalb der Subklasse können beliebig viele neue Attribute und Methoden hinzugefügt werden.
- Die Subklasse spezialisiert die Basisklasse.

5.3.1 Sichtbarkeit in der Vererbungshierarchie

Im Zusammenhang mit der Vererbung kommt noch eine weitere Art der Zugriffsbeschränkung hinzu. Protected-Attribute und Methoden, in UML-Notation gekennzeichnet durch die Raute #, erlauben den Zugriff innerhalb der Vererbungshierarchie. Protected-Elemente sind für Subklassen und innerhalb des Paketes sichtbar, außerhalb der Klassenhierarchie weiterhin nicht.

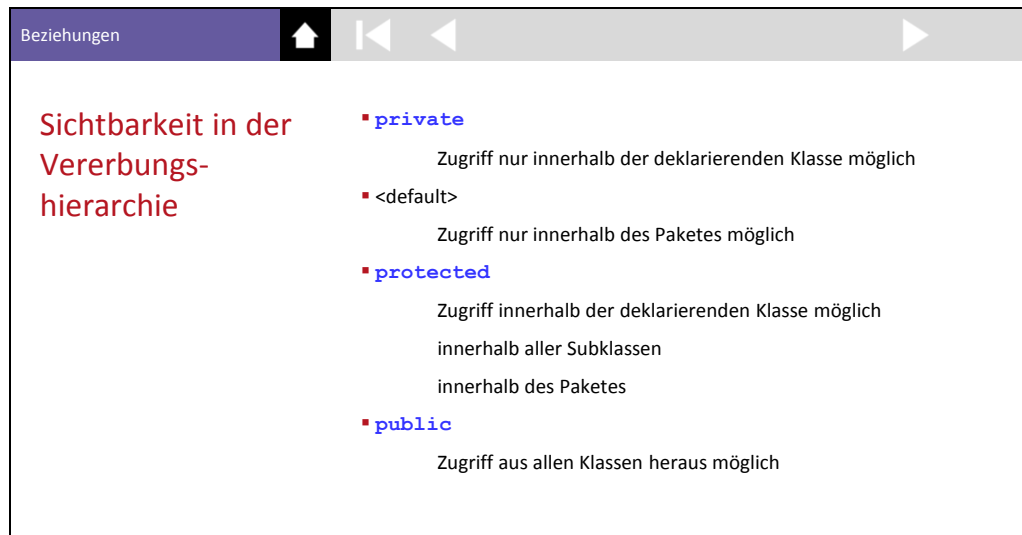


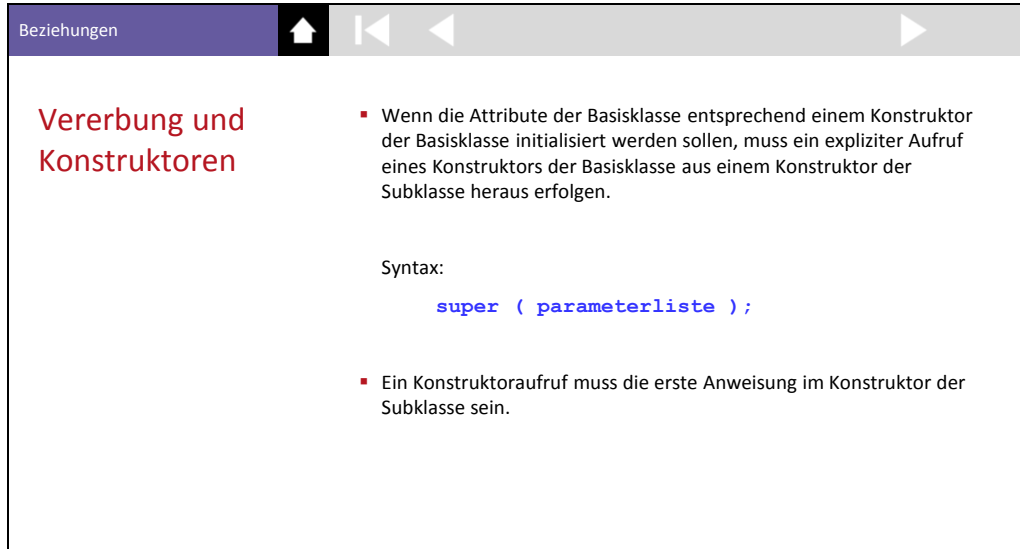
Abb. 5-5: Sichtbarkeit

5.3.2 Vererbung und Konstruktoren

Wie im vorhergehenden Abschnitt gesehen, werden alle Attribute und Methoden der Basisklasse an die Subklasse vererbt, nicht aber die Konstruktoren. Konstruktoren werden nicht vererbt, weil es dem Prinzip der Kapselung zuwider laufen würde. Aus der abgeleiteten Klasse darf nicht auf die Attribute zugegriffen werden.

Andererseits wird nun ein Objekt der Subklasse bildhaft so erzeugt, dass zusätzlich zu den Attributen der Basisklasse die Attribute der Subklasse angefügt werden. Eine Instanziierung der Subklasse bedingt also auch stets einen Konstruktoraufruf der Basisklasse. Wird aus der Subklasse kein Konstruktor der Basisklasse explizit aufgerufen, so wird der von System erzeugte Teil des default-Konstruktor der Basisklasse implizit verwendet.

Wenn aber die Attribute der Basisklasse entsprechend einem Konstruktor der Basisklasse initialisiert werden sollen, muss ein expliziter Aufruf eines Konstruktors der Basisklasse aus einem Konstruktor der Subklasse heraus erfolgen.



Beziehungen

Vererbung und Konstruktoren

- Wenn die Attribute der Basisklasse entsprechend einem Konstruktor der Basisklasse initialisiert werden sollen, muss ein expliziter Aufruf eines Konstruktors der Basisklasse aus einem Konstruktor der Subklasse heraus erfolgen.

Syntax:

```
super ( parameterliste );
```

- Ein Konstruktoraufruf muss die erste Anweisung im Konstruktor der Subklasse sein.

Abb. 5-6: Vererbung und Konstruktoren

5.3.3 Überschreiben von Attributen

Eine Überlagerungstechnik in Java ist das **Überschreiben**. Überschrieben werden Attribute. Wenn ein Attribut in einer Superklasse mit einem bestimmten Typ deklariert wird, so kann in der Subklasse durchaus ein Attribut gleichen Namens (mit einem identischen oder auch abweichenden Typ) deklariert werden, das dann dort das Attribut der Superklasse überlagert. Das nachfolgende Beispiel skizziert das Verfahren:

Beispiel:

```
class Superklasse() {  
    int berechnung;  
}  
  
class Subklasse extends Superklasse {  
    double berechnung;  
}
```

Der Zugriff auf die Variable der Superklasse kann unter anderem wieder mit dem Schlüsselwort `super` erreicht werden.

5.3.4 Überschreiben von Methoden

In den abgeleiteten Klassen können Methoden der Basisklasse überschrieben werden. Das heißt, es werden der Subklasse Elemente gleichen Namens hinzugefügt. Die Methoden der Basisklasse sind jedoch auch in den Subklassen durch einen Verweis auf die Basisklasse weiter verfügbar.

Hat eine Methode in einer abgeleiteten Klasse die absolut gleiche Signatur einer Methode der Basisklasse, spricht man von überschreiben, überschatten oder überdecken.

Überschreiben (Overriding) heißt im Englischen eigentlich Overwriting. Overriding ist jedoch kein Schreibfehler, sondern es bedeutet im Grunde Überdefinieren, wird jedoch im Deutschen zum besseren (?) Verständnis als Überschreiben übersetzt.

Soll aus der Methode der abgeleiteten Klasse die gleichnamige der Basisklasse aufgerufen werden, muss das Schlüsselwort `super` als Referenz verwendet werden. Ohne `super` handelt es sich um einen rekursiven Aufruf.

Beispiel:

```
public class Mitarbeiter extends Person{
    ...
    //Überschreiben der Methode aus Person
    public String vorstellen(){
        return super.vorstellen()
            + ", ich bin Mitarbeiter";
    }
}
```

Überschriebene Methoden sind die Voraussetzung zur Nutzung der Polymorphie, wie sie im folgenden Abschnitt vorgestellt wird.

5.3.5 Polymorphie

Instanzen der Klasse Person und Mitarbeiter sind von unterschiedlichem Typ. Der Zugriff auf Objekte erfolgt in Java über Referenzen.

```
Person p = new Person();
```

```
Mitarbeiter m = new Mitarbeiter();
```

Beziehungen
⬆
⬅
➡

Vererbung und Polymorphie

- Überschreiben von Methoden und Attributen
In den abgeleiteten Klassen können Methoden und Attribute der Basisklasse überschrieben/überschattet werden. Das heißt, es werden in der Subklasse Elemente gleichen Namens hinzugefügt.
Hat eine Methode in einer abgeleiteten Klasse die absolut gleiche Signatur einer Methode der Superklasse, spricht man von **überschreiben** oder überdecken.
- Polymorphie
Instanzen der Klasse Person und Mitarbeiter sind von unterschiedlichem Typ. Der Zugriff auf Objekte erfolgt in Java bekannter Weise über Referenzen.

```
Person p = new Person();
```



```
Mitarbeiter m = new Mitarbeiter();
```


Bei Verwendung des Zuweisungsoperators darf einer Referenz einer allgemeinen Basisklasse auch die Referenz einer abgeleiteten Klasse zugewiesen werden, ohne dass der Cast-Operator notwendig wird.

```
p = m; // erlaubt, weil m von p abgeleitet ist
```



```
p.vorstellen(); // welches vorstellen wird hier aufgerufen?
```

Abb. 5-7: Polymorphie

Was wird nun bei folgenden Aufrufen ausgegeben:

```
p.vorstellen();
```

```
m.vorstellen();
```

Hier ist leicht vorstellbar, dass die jeweilige Methode der Klasse aufgerufen wird, zu der die Referenz links vom Punkt gehört. Beim 2. Aufruf also die überschriebene Methode aus der Klasse Mitarbeiter.

Nun folgende entscheidende Überlegung:

Bei Verwendung des binären Zuweisungsoperators darf einer Referenz einer Basisklasse auch die Referenz einer abgeleiteten Klasse zugewiesen werden, ohne dass der Cast-Operator notwendig wird.

```
p = m; // erlaubt, weil m von p abgeleitet ist
```

```
p.vorstellen();
```

Die Referenz p wurde zwar als Typ Person deklariert, verweist zurzeit aber auf ein Objekt der abgeleiteten Klasse Mitarbeiter. Das System prüft zur Laufzeit die Bindung und ruft die jeweils „richtige“ Methode auf, in diesem Beispiel die Methode vorstellen() aus der abgeleiteten Klasse Mitarbeiter.

Dieser Effekt wird Spätes Binden genannt und stellt eine Form der Polymorphie dar.

Die Polymorphie, das Späte Binden, wird in der Programmiersprache Java automatisch ohne zusätzlichen Mehraufwand bei der Programmierung realisiert.

Dieser enorm wichtige Effekt kann nur mit Hilfe von überschriebenen Methoden realisiert werden.

5.3.6 Vererbung und der Cast-Operator

Durch den Effekt des Späten Bindens wird deutlich, dass eine Referenz zur Laufzeit sozusagen die Information mitführt, auf welchen Datentyp es gerade verweist. Dies kann auch programmtechnisch verwendet werden, um zur Laufzeit Objekte zu analysieren. Man „fragt“ eine Referenz, auf welchen Typ sie zeigt.

Dazu wird der binäre Operator `instanceof` verwendet.

Syntax:

referenz instanceof Klasse

Der Ausdruck `a instanceof b` liefert genau dann `true`, wenn `a` eine Instanz der Klasse `b` oder einer ihrer Subklassen ist.

The screenshot shows a presentation slide with a purple header bar containing the word 'Beziehungen' and navigation icons. The slide title is 'Vererbung und der Cast-Operator'. It contains a bulleted list, a 'Syntax:' section with the code 'Referenz instanceof Klasse', and another bullet point about casting. A code snippet shows a method 'einmieten' that uses 'instanceof' to check if a 'Person' object is a 'Mitarbeiter' before casting it.

Vererbung und der Cast-Operator

- Durch den Effekt des Späten Bindens wird deutlich, dass eine Referenz zur Laufzeit sozusagen die Information mitführt, auf welchen Datentyp es gerade verweist. Dies kann auch programmtechnisch verwendet werden, um zur Laufzeit Objekte zu analysieren. Man „fragt“ eine Referenz, auf welchen Typ sie zeigt.

Syntax:

Referenz `instanceof` Klasse

- Ein Cast von der Basisklasse in die Subklasse wird vom Compiler akzeptiert.

```
public void einmieten( Person p )
{
    if( p instanceof Mitarbeiter )
        Mitarbeiter a = ( Mitarbeiter ) p; // cast
    // ...
}
```

Abb. 5-8: instanceof

Durch die Abfrage stellen wir sicher, dass nur für Mitarbeiter die Bedingung erfüllt ist, wir können aber ohne Compilerfehler immer noch nicht die Methode `getAbt()` aufrufen.

```
p.getAbt(); // error, da getAbt()
           // keine Methode der Klasse Person
```

Die Lösung kennen wir bereits: Die Typumwandlung durch Cast.

Ein Cast von der Basisklasse in die Subklasse wird vom Compiler akzeptiert.

```
public void einmieten(Person p) {  
    //...  
    if(p instanceof Mitarbeiter) {  
        Mitarbeiter m = (Mitarbeiter) p;  
        System.out.println(  
            "Einmieten eines Mitarbeiters"  
            + "aus der Abteilung "  
            + m.getAbt() );  
    }  
    //...  
}
```

Ein Cast zwischen beliebigen Klassen ist nicht möglich. Es muss als Voraussetzung eine Vererbungshierarchie vorliegen. So kann ein Auto beispielsweise nicht in eine Person gecasted werden oder umgekehrt.

Die Laufzeitumgebung prüft den Cast auf Zulässigkeit. Unzulässige Casts werden durch das Werfen eines speziellen Exceptionobjektes, der `java.lang.ClassCastException` signalisiert.

5.3.7 Hierarchie der Exception-Klassen

In den bisherigen Beispielen und Demonstrationen haben wir nur die einfache Exception-Klasse verwendet. Nun wollen wir uns einen Überblick über die Hierarchie der Exception-Klassen verschaffen, beschränken uns dabei jedoch auf das Paket `java.lang`. Jedes Unterpaket definiert eine ganze Reihe weiterer Typen. Die Vererbungshierarchie ist in der Regel jedoch flach, so dass wir durch die Aufnahme weiterer Exceptions keine neuen Erkenntnisse gewinnen werden.

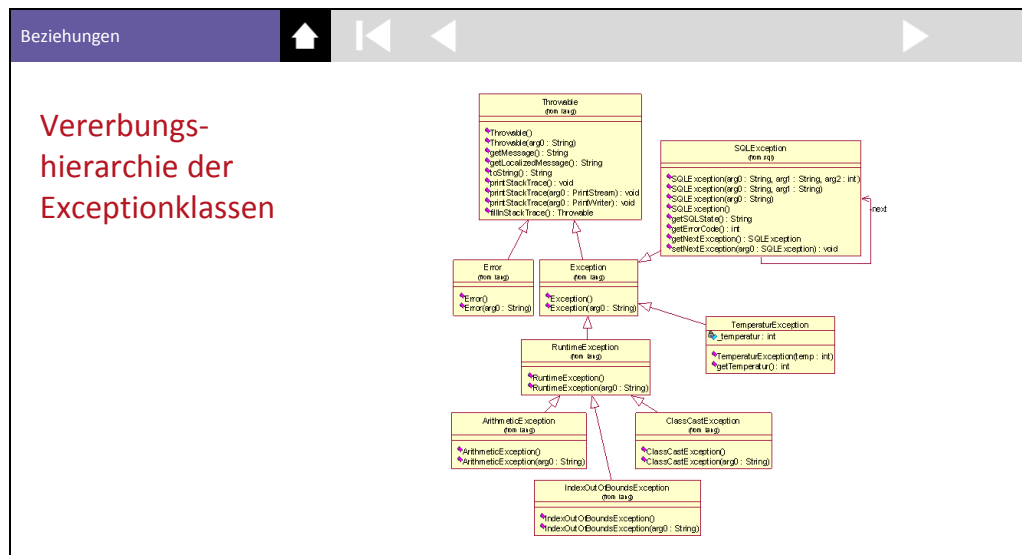


Abb. 5-9: Hierarchie der Exceptionklassen

In diesem Klassendiagramm sind einige Exception-Klassen aus `java.lang` aufgeführt.

- Die Klassenhierarchie beginnt bei `Throwable`. Darin sind die elementaren Methoden aller Ausnahme-Objekte definiert. Im Wesentlichen sind dies der parameterlose Konstruktor, der Konstruktor mit fehlerbeschreibendem Text, `getMessage()` und `printStackTrace()` zum Dump des Methodenaufrufs. Die Methode `fillInStackTrace()` wird intern beim Hochwerfen der Exception genutzt. Die Subklassen definieren meist nur einen eigenen Konstruktor, diese sind in obigem Klassendiagramm ausgeblendet.
- Neben der bereits bekannten Exception-Klasse existiert auf der gleichen Ebene der Hierarchie die Klasse `Error`. Sie wird im Gegensatz zu `Exception` zur Signalisierung „wirklicher“ Fehlersituationen genutzt. Referenzen auf `Error`-Objekte können gefangen werden. Das fehlerhafte Programm sollte dann aber kontrolliert beendet werden. `Error` sollte nicht erweitert werden, das ist den Entwicklern von Java vorbehalten.

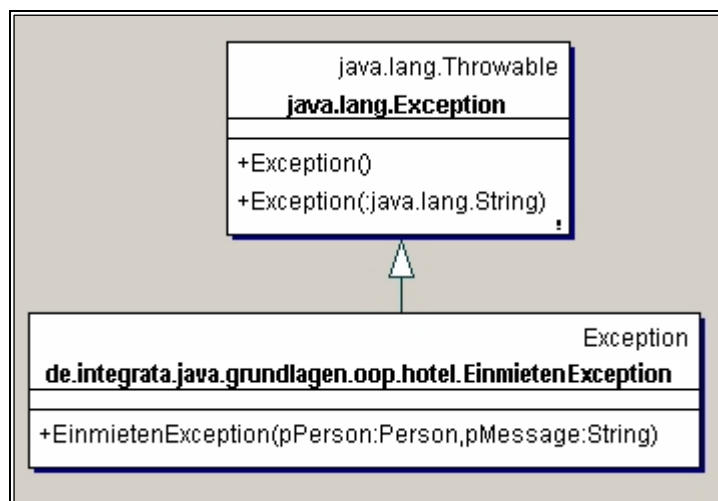
5.3.8 Die RuntimeException

Eine spezielle Exception-Subklasse ist ebenfalls noch hervorgehoben: Die RuntimeException. Sie wurde mit einigen weiteren Subklassen wie der NullPointerException eingeführt, um eine unnötig komplizierte Liste von Exceptions in der Liste der Methodendeklarationen zu vermeiden. Runtime-Exceptions müssen nicht in der throws-Klausel einer Methode deklariert werden, können aber in einem catch-Zweig aufgefangen werden.

5.3.9 Eigene Exception-Klassen

Die Klasse `java.lang.Exception` ist nicht als final deklariert. Eigene Exception-Klassen, die ganz spezielle anwendungsspezifische Fehlersituationen signalisieren sollen, können durch Vererbung erzeugt werden.

So können wir z. B. für unser Hotel eine `EinmietenException` definieren:



Diese kann nun in den unterschiedlichsten Situationen und aus den verschiedensten Gründen geworfen werden. Der aufrufenden Ebene werden folglich die Information über die einzumietende Person und ein beschreibender Text übergeben.

Die Einführung eigener Exception-Klassen sollte moderat durchgeführt werden. Eine eigene Exception-Klasse

`HotelWithNegativeRoomCountException` zum Signalisieren des Fehlers bei einer negativen Anzahl von Zimmern bei der Instanziierung bringt keine wesentlich neue Information und kann auch nur an einer einzigen Stelle geworfen werden.

5.4 Finale Elemente

Finale Elemente werden bei der Deklaration durch das Schlüsselwort `final` gekennzeichnet.

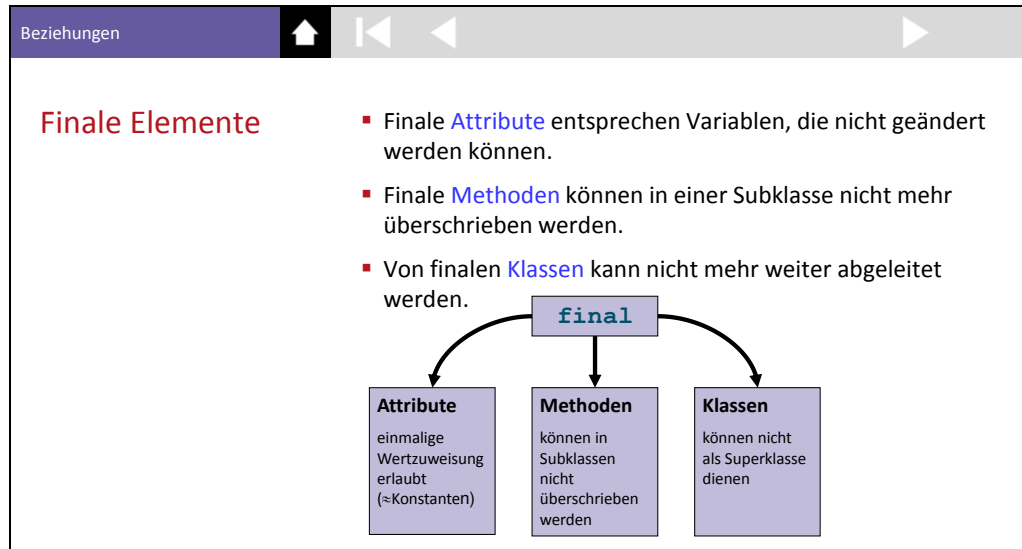


Abb. 5-10: Finale Elemente

6

Abstrakte Klassen, Interfaces und Pakete

6.1	Abstrakte Klassen.....	6-3
6.2	Mehrfachvererbung	6-5
6.3	Interfaces (Schnittstellen)	6-5
6.3.1	Allgemeines	6-5
6.3.2	Funktionales Interface.....	6-7
6.3.3	Interfaces und Mehrfachvererbung	6-8
6.4	Zusammenfassung der Deklarationen.....	6-9
6.4.1	Deklaration einer Klasse	6-9
6.4.2	Deklaration von Attributen.....	6-9
6.4.3	Deklaration von Methoden	6-10
6.4.4	Deklaration von Konstruktoren.....	6-10
6.5	Pakete	6-11
6.5.1	Eigene Pakete erstellen	6-12
6.5.2	Import von Klassen	6-13
6.5.3	Statische Importe	6-14
6.6	Zusammenfassung der Zugriffsrechte	6-16

6 Abstrakte Klassen, Interfaces und Pakete

6.1 Abstrakte Klassen

In Objektorientierten Programmiersprachen gibt es die Möglichkeit, sogenannte abstrakte Klassen und Methoden zu deklarieren. Abstrakte Klassen sind solche, die nicht instanziiert werden können (von denen man also keine Objekte erzeugen kann). Es lassen sich allerdings Referenzen vom Typ solcher abstrakter Klassen deklarieren.

Abstrakte Klassen werden mit Attributen und konkreten Methoden ausgestattet und dienen in der Vererbungshierarchie als Basisklassen. Die davon abgeleiteten Subklassen können wieder abstrakt oder aber konkret sein.

Auch Methoden können abstrakt deklariert werden. Sie besitzen dann keinen Funktionsrumpf, nicht einmal einen Funktionsblock (ein leeres geschweiftes Klammern Paar). Anstelle des Funktionsblockes wird ein Semikolon angefügt. Eine abstrakte Methode wird also nur deklariert, nicht definiert. Die Deklaration hat das Aussehen eines Prototyps in C++.

Enthält eine Klasse mindestens eine abstrakte Methode, so muss auch die Klasse als abstrakt deklariert werden. Abstrakte Methoden müssen in der Subklasse überschrieben werden, anderenfalls wird die Subklasse ebenfalls abstrakt. Mit abstrakten Methoden kann somit während der Modellierung der Klassenhierarchie Funktionalität auf einer Ebene vorgesehen werden, in der die konkrete Implementierung noch gar nicht bekannt ist.

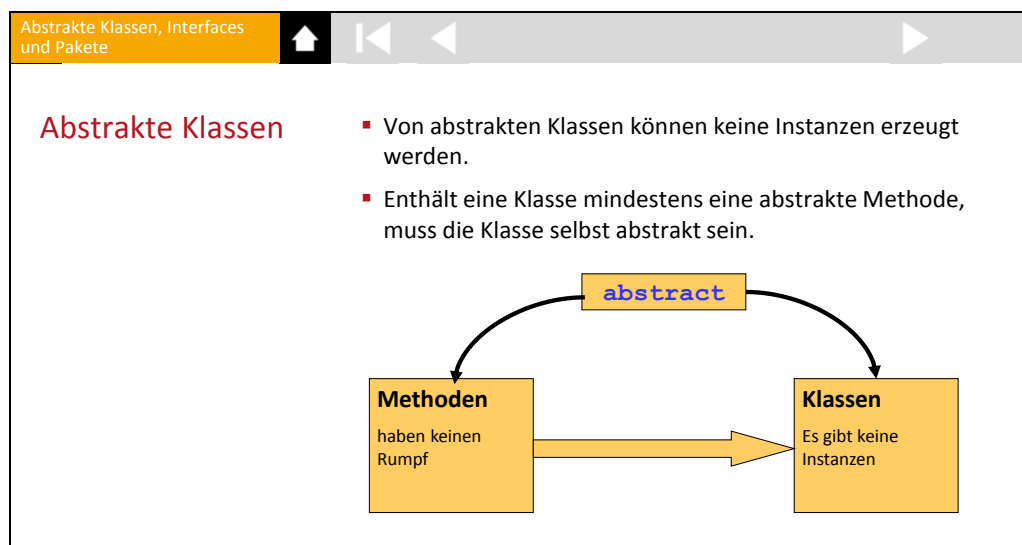


Abb. 6-1: abstrakte Elemente

In Java werden abstrakte Klassen durch das Schlüsselwort `abstract` bei der Klassendeklaration gekennzeichnet.

Abstrakte Klassen, Interfaces und Pakete

⬆
⬅
➡

Abstrakte Klassen

Abstrakte Methoden müssen in abgeleiteten Klassen implementiert werden, sonst ist die Subklasse ebenfalls abstrakt.

- Klassendeklaration


```
sichtbarkeit abstract class Klassenname
{
    // Klassenkörper mit Attributen
    // und Methodendeklarationen und -definitionen
}
```
- Methodendeklaration


```
sichtbarkeit abstract typ methodenname ( parameter );
// kein Methodenblock
```

Abb. 6-2: abstract

In unserem Beispiel der Klasse `Person` könnte z. B. eine Superklasse `Lebewesen` eingeführt werden, von der die Klasse `Person` und andere Subklassen (Hunde, Katzen, Fische...) abgeleitet werden. Die Klasse `Lebewesen` ist sinnvollerweise als abstrakt zu deklarieren: In der Realität ist der Begriff `Lebewesen` zu wenig konkret, als dass die Existenz eines Objekts vom Typ `Lebewesen` sinnvoll wäre.

Beispiel:

```
public abstract class Lebewesen {
    // Attribute:
    private int anzahlBeine;
    // Methoden
    public int getAnzahlBeine() {
        return anzahlBeine;
    }
    public void setAnzahlBeine(int anzahl) {
        anzahlBeine = anzahl;
    }
    // abstrakte Methoden
    public abstract String getGattung();
}
```

6.2 Mehrfachvererbung

Um Mehrfachvererbung handelt es sich, wenn eine Klasse mehr als eine Oberklasse hat.

Problematisch wird bei der Mehrfachvererbung, wenn mehrere der Oberklassen wieder eine gemeinsame Basisklasse haben.

Die in der Objektorientierten Programmierung geforderte Mehrfachvererbung wird in Java nicht unterstützt, auch nicht bei abstrakten Klassen.

Java kennt nur die Einfachvererbung!

Die Vorteile der abstrakten Klassen inklusive der Mehrfachvererbung aber sollen auch in Java genutzt werden. Anstelle von abstrakten Klassen werden in Java die Interfaces angeboten.

6.3 Interfaces (Schnittstellen)

6.3.1 Allgemeines

Das Verständnis der abstrakten Methoden und Klassen liefert uns nun die Möglichkeit, das Konzept der Schnittstellen oder „Interfaces“ in Java zu behandeln.

Ein Interface ist per Definition eine Sammlung abstrakter Methoden und statischer, finaler Attribute, auf den ersten Blick also der abstrakten Klasse sehr ähnlich. Der Unterschied liegt darin begründet, dass bis Java 7 keine konkreten Methoden und keine Instanz Variablen erlaubt sind. Da die Attribute final und private sein müssen, wird man in Interfaces keine Attribute definieren. Interfaces enthalten üblicherweise ausschließlich abstrakte Methoden. Seit JDK 8 können Interfaces auch implementierte Methoden (Default-Methoden) haben. Dies ermöglicht die Erweiterung von Interfaces ohne die implementierenden Klassen zu verändern.

Ein Interface wird analog einer Klasse deklariert, jedoch wird anstelle von class das Schlüsselwort interface verwendet.

Abstrakte Klassen, Interfaces und Pakete

Interfaces

- Ein Interface ist per Definition eine Sammlung abstrakter Methoden und statischer, finaler Attribute, auf den ersten Blick also der abstrakten Klasse sehr ähnlich. Der Unterschied liegt darin begründet, dass keine Instanzattribute erlaubt sind.
- Erweiterungen seit JDK8: Interfaces können statische Methoden definieren und Default-Methoden implementieren

Syntax:

```
interface Interfacename {  
    // Deklaration statische, finale Attribute  
    // und abstrakte Methoden  
    // statische Methoden  
    // und default Methoden  
}
```

Abb. 6-3: Interface

Wenn nun eine Klasse von einem Interface abgeleitet wird, so sind die abstrakten Methoden des Interfaces in der Klasse zu implementieren.

„Erben“ verschiedene Klassen von einem Interface, so wird dadurch eine Gemeinsamkeit geschaffen. Genau das Ziel, welches mit Interfaces erreicht werden soll.

Wird eine Klasse von einem Interface abgeleitet, so muss anstelle des Schlüsselwortes `extends` das Schlüsselwort `implements` verwendet werden.

Abstrakte Klassen, Interfaces und Pakete

Implementierung von Interfaces

- Wird eine Klasse von einem Interface abgeleitet, so muss anstelle des Schlüsselwortes `extends` das Schlüsselwort `implements` verwendet werden.
- Für Schnittstellen wird die Mehrfachvererbung unterstützt.

Syntax:

```
sichtbarkeit class KlassenName implements Interface1,  
                                     Interface2  
{  
    // Klassendefinition  
}
```

- Eine Klasse, die eine Schnittstelle implementiert, ist solange abstrakt, bis alle im Interface deklarierten abstrakten Methoden definiert wurden.

Abb. 6-4: Implementierung von Interfaces

Beispiel:

```
public interface Adressierbar {
    public String getAnschrift(); // abstrakte Methode
}

public class Hotel implements Adressierbar {
    //...
    private String ort;
    //...
    public String getAnschrift() {
        return getHotelName() + ", in " + ort;
    }
}
```

Eine Klasse, die eine Schnittstelle implementiert, bleibt solange abstrakt, bis alle im Interface deklarierten abstrakten Methoden definiert wurden.

Da ein Interface ebenso wie eine abstrakte Klasse die Deklaration eines neuen Datentyps darstellt, können zwar keine Objekte instanziiert werden, sehr wohl aber Referenzen.

6.3.2 Funktionales Interface

Seit JDK8 können Interfaces auch implementierte Methoden (Default-Methoden) beinhalten. Wenn ein Interface lediglich eine abstrakte Methode und beliebig viele Default-Methoden enthält, so nennt man dieses Interface ein Funktionales Interface.

Beispiel:

```
@FunctionalInterface
public interface Adressierbar {

    // abstrakte Methode:
    public String getAnschrift();

    // default Methode:
    public default void druckOrt( ) {
        // Implementierung
    }
}
```

Eine Klasse, die dieses Interface implementiert, muss nur die Methode `getAnschrift` implementieren. Wenn die default-Methode `druckOrt` nicht von der Klasse implementiert (überschrieben) wird, dann werden die Anweisungen der Default-Implementierung des Interfaces ausgeführt.

6.3.3 Interfaces und Mehrfachvererbung

Für Interfaces wird in Java die „Mehrfachvererbung“ unterstützt.

Eine Klasse kann mehrere Schnittstellen implementieren.

```
public class Klasse implements I1, I2 {  
    // Attribute  
  
    // Methoden  
  
}
```

Ebenfalls kann ein Interface von mehreren Interfaces gleichzeitig erben.

```
interface I extends I1, I2 {  
  
    // Deklaration abstrakter Methoden  
  
}
```

6.4 Zusammenfassung der Deklarationen

6.4.1 Deklaration einer Klasse

Abstrakte Klassen, Interfaces und Pakete

Übersicht der Deklarationen: Klassen

- Die mit eckigen Klammern umschlossenen Angaben sind optional.

Syntax der Klassendeklaration:

```
[sichtbarkeit] [final] [abstract] class Klassenname
    [extends Oberklasse]
    [implements Interface1, Interface2 ...]
{
    // Klassenkörper
}
```

- Sichtbarkeit:
 - `public` Sind auch außerhalb des Paketes sichtbar
 - `<default>` Sind nur innerhalb des Paketes sichtbar

Abb. 6-5: Klasse

6.4.2 Deklaration von Attributen

Abstrakte Klassen, Interfaces und Pakete

Übersicht der Deklarationen: Attribute

- Syntax der Attributdefinition

```
[sichtbarkeit] [final] [gültigkeit] typ attributname;
```
- Sichtbarkeit
 - `public` Sind auch außerhalb der Klasse sichtbar
 - `protected` Sind innerhalb der Klasse, allen Klassen der Klassenhierarchie und des selben Paketes sichtbar
 - `<default>` Sind innerhalb der Klasse und desselben Paketes sichtbar
 - `private` Sind nur innerhalb der Klasse sichtbar
- Gültigkeit
 - `static` Klassenattribut
 - `<default>` Instanzattribut

Abb. 6-6: Attribute

6.4.3 Deklaration von Methoden

Abstrakte Klassen, Interfaces
und Pakete

Übersicht der Deklarationen: Methoden

- Syntax der Methodendefinition

```

[sichtbarkeit] [final] [abstract] [gültigkeit]
typ methodenname ( parameterliste )

{
    // Methodenkörper
}

```
- Sichtbarkeit

```

public           protected
<default>       private

```
- Gültigkeit

```

static           <default>

```

Abb. 6-7: Methoden

6.4.4 Deklaration von Konstruktoren

Abstrakte Klassen, Interfaces
und Pakete

Übersicht der Deklarationen: Konstruktoren

- Syntax der Konstruktordefinition:

```

[sichtbarkeit] Klassenname ( parameterliste )
{
    // Aufruf von this( ) oder super( )
    // Konstruktorkörper
}

```
- Sichtbarkeit
 - public** in allen Klassen sichtbar
 - protected** innerhalb der Klasse, in allen Subklassen und in allen Klassen desselben Pakets sichtbar
 - <default>** innerhalb der Klasse und in allen Klassen desselben Pakets sichtbar
 - private** nur innerhalb der Klasse sichtbar

Abb. 6-8: Konstruktoren

6.5 Pakete

Die Festlegung der Zugehörigkeit einer Klasse zu einem Paket wird durch die Anweisung:

```
package paketname;
```

erreicht, welche als erste Anweisung im Quellcode stehen muss.

Pakete

- Pakete (Packages) bündeln Gruppen von Klassen
 - java.lang
 - java.io
 - java.awt
 - de.integrata.grundlagen.oop
- Klassen können auf zweierlei Arten angesprochen werden:
 - über vollqualifizierten Klassennamen
 - de.integrata.grundlagen.oop.personen.Person
 - über Importieren und mit kurzen Klassennamen ohne Paketnamen
 - nur eine Klasse importieren
 - import de.integrata.grundlagen.oop.personen.Person;
 - oder alle Klassen eines Paketes importieren (Achtung: nicht rekursiv!)
 - import de.integrata.java.grundlagen.oop.personen.*;

Abb. 6-9: Pakete

Beispiel:

```
package de.integrata.java.grundlagen.oop;
```

Der Java-Compiler javac kennt die Option `-d` plus Pfadname, mit der eine Umleitung der Bytecode-Files möglich ist. Eine Standard-Einstellung wäre somit

```
javac -d c:\_training\classes *.java
```

Seit JDK 1.2 kann den einzelnen Werkzeugen des JDK der Klassenpfad auch durch die Option `-classpath` als Parameter mitgegeben werden.

Die Bytecode-Dateien werden dann im Verzeichnis

```
c:\_training\classes
```

gesucht, vorausgesetzt dass die Variable `CLASSPATH` diesen Wert enthält.

6.5.1 Eigene Pakete erstellen

Wir hatten bereits gesehen, dass in der ersten Anweisung einer Java-Sourcdatei die Zugehörigkeit einer Klasse zu einem Paket (package) definiert wird. In einem Paket werden verwandte Klassen und Interfaces zusammengefasst. Vorteile dieser Bündelung sind:

- Leichtes Auffinden von Klassen.
- Keine Namenskonflikte mit Klassen in anderen Paketen.
- Festlegung von Beziehungen zwischen verwandten Klassen.
- Eine zusätzliche Ebene der Kapselung.

Wird keine Paketfestlegung getroffen, werden die im Programm enthaltenen Informationen in einem anonymen Paket gesammelt und die Klassen können nur von Programmen im selben Verzeichnis benutzt werden. Dies ist für einfache Testprogramme zulässig, in der Praxis aber nicht häufig anzutreffen. Deshalb sollten die verwendeten Klassen auch sofort den Paketen

```
de.integrata.java.grundlagen.sprache bzw.  
de.integrata.java.grundlagen.oop
```

zugeordnet werden. Eine feinere Unterteilung ist jedoch sicherlich möglich und auch sinnvoll.

Führen wir nun aber die Unterpakete

```
de.integrata.java.grundlagen.oop.person  
de.integrata.java.grundlagen.oop.hotel  
de.integrata.java.grundlagen.oop.adresse  
de.integrata.java.grundlagen.oop.adressbuch  
de.integrata.java.grundlagen.oop.anwendung  
de.integrata.java.grundlagen.oop.mitarbeiter
```

ein, so bekommen wir ein Problem dadurch, dass die einzelnen Klassen nun statt des innerhalb des gleichen Paketes gültigen verkürzten Klassennamens, nun häufig den unpraktisch langen voll qualifizierten Namen verwenden müssen. Es ist jedoch möglich, innerhalb einer Quellcode-Datei eine Liste von Klassen zu importieren.

6.5.2 Import von Klassen

Durch den Import von Klassen aus Paketen können diese auch verkürzt angesprochen werden.

Zur Importierung von Klassen steht das Schlüsselwort `import` zur Verfügung:

```
import de.integrata.java.grundlagen.oop.Person;
```

Hier wird die Klasse `Person` aus dem Paket

```
de.integrata.java.grundlagen.oop
```

verkürzt ansprechbar.

Importierung aller Klassen eines Pakets ist durch die Angabe eines `"*"` möglich:

```
import de.integrata.java.grundlagen.oop.* ;
```

Hier ist anzumerken, dass die Typen eines Unterpaketes nicht mit importiert werden. Diese müssten in weiteren Anweisungen angegeben werden.

Importe der Form

```
import de.integrata.java.grundlagen.oop.Ar*;
```

oder

```
import de.integrata.java.grundlagen.*.*
```

sind nicht zulässig und liefern einen Compilerfehler.

```
import java.*
```

importiert keine einzige Klasse.

Das Java Laufzeitsystem importiert defaultmäßig zwei Pakete:

- das Paket ohne Namen
- das Paket `java.lang`

Pakete bündeln Gruppen von Klassen in eine Bibliothek, die unter dem Namen des Paketes angesprochen wird.

Beispiel:

```
java.lang
```

```
java.io
```

```
java.awt
```


Erzeugen eines eigenen Paketes:

```
package meinpaket;  
[Klassen- und Schnittstellendefinitionen]
```

Referenzieren einer Klasse des Paketes über den voll qualifizierten Klassennamen:

```
meinpaket.Klassenname
```

Importieren des gesamten Namenraumes eines Paketes:

```
import meinpaket.*;
```


6.5.3 Statische Importe

Mit der Sprachversion 5.0 wurde die import-Anweisung erweitert: Neben der oben angesprochenen Möglichkeit, ganze Klassen zu importieren, können auch statische Attribute und Methoden angegeben werden. Dazu dient die Anweisung `import static`.

Ein Beispiel hierfür liefert die Klasse `java.lang.Math`, die eine mathematische Funktionsbibliothek in Form statischer Methoden und Attribute enthält. Die folgende Klasse zur Berechnung von Kreisparametern verwendet intern `Math`:

```
public class CircleUtil {  
    public static void main(String[] args) {  
        CircleUtil util = new CircleUtil();  
        double radius = 2;  
        System.out.println("Area: "  
                           + util.area(radius));  
    }  
  
    public double area(double radius) {  
        return Math.pow(radius, 2d) * Math.PI;  
    }  
}
```

Mit der neuen Anweisung `import static` ist eine direkte Benutzung möglich, falls keine Namenskonflikte auftreten können:



The screenshot shows a presentation slide with a title bar 'Abstrakte Klassen, Interfaces und Pakete' and navigation icons. The main title is 'Statische Importe'. The content includes a list of bullet points and a code example.

- Import für direkten Zugriff auf statische Attribute und Methoden
`import static`
- Beispiel: Zugriff auf Konstante `PI` und auf Methode `pow()` der Klasse `java.lang.Math`
- Beispiel:

```
import static java.lang.Math.*;

public class CircleUtil {
    public static void main( String[] args ) {
        CircleUtil util = new CircleUtil( );
        System.out.println( "Area: " + util.area(2.5) );
    }

    public double area( double radius ) {
        return pow( radius, 2d ) * PI;
    }
}
```

Abb. 6-10: statischer Import

Hinweis: Wie beim Importieren von Klassen ist auch hier der " * " zulässig, alle statischen Attribute und Methoden in einer Anweisung anzugeben. Im obigen Beispiel hätten auch statische Imports der Form
`import static java.lang.Math.PI;`
`import static java.lang.Math.pow;`
genügt. Er ist nicht mit einem Platzhalter-Zeichen zu verwechseln:
`import static java.lang.Math.P*;`
ist syntaktisch falsch.

Überladenen Methoden werden komplett importiert. So würde ein Import der Form `import static java.lang.Math.abs` insgesamt 4 Methoden statisch importieren (mit `double`, `float`, `int` und `long` als Parameter).

Bei statische Imports müssen die Klassennamen stets voll qualifiziert angegeben werden. `import static Math.PI` ist syntaktisch falsch.

6.6 Zusammenfassung der Zugriffsrechte

Nach der Einführung des Begriffs Import können wir nun die vollständige Bedeutung des Schlüsselwortes `protected` nachreichen bzw. eine allgemeine Übersicht über die Zugriffsrechte in Java liefern:

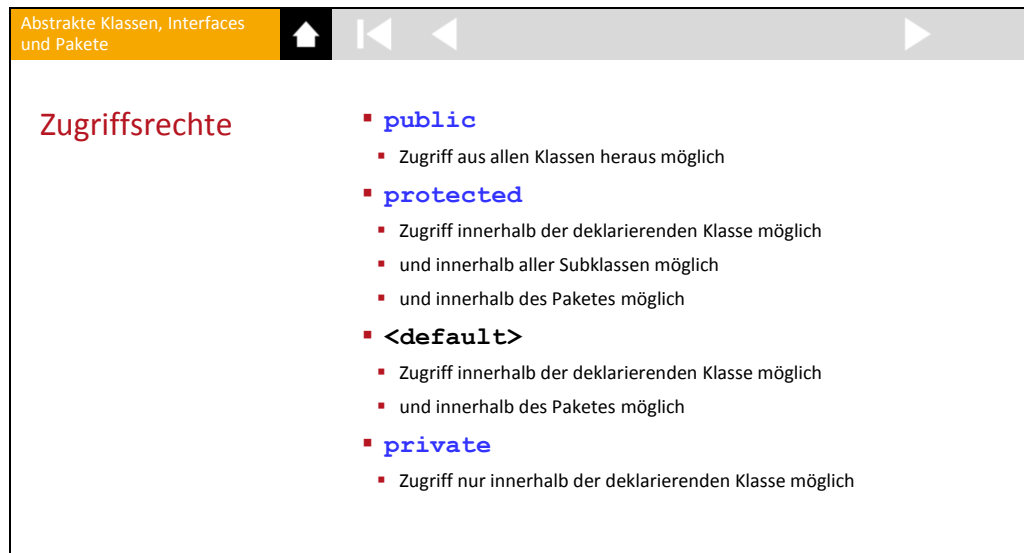


Abb. 6-11: Zugriffsrechte

Auch Klassen werden durch Pakete gekapselt: Klassen können mit Hilfe des Schlüsselwortes `public` als öffentlich deklariert werden. Nur dann ist eine Verwendung in anderen Paketen möglich.

Wird dieses Schlüsselwort weggelassen, wird die Paketsichtbarkeit verwendet. Diese ist beschränkt auf die Klassen des eigenen Paketes. Dies ist natürlich erneut sehr praktisch, weil auf diese Art und Weise beispielsweise der Zugriff auf Hilfsklassen anderen Paketen verboten werden kann. Wichtiger ist hier jedoch die Möglichkeit, aus Modellierungsgründen den Zugriff zu verbieten. So ist es möglich, eine Paketsignatur zu definieren und Pakete als Ganzes über die `public`-Klassen zu koppeln.

7

Weiterführende Themen

7.1	Singleton	7-3
7.2	Assertions.....	7-5
7.3	Wrapper-Klassen.....	7-7
	7.3.1..... Autoboxing/Unboxing	7-8
7.4	Enumeration	7-9
7.5	Variable Argumentlisten	7-11
7.6	Metadaten	7-12
	7.6.1..... Vordefinierte Annotations und Meta-Annotations.....	7-13
	7.6.2..... Deklaration eigener Metadaten und Tool apt	7-14

7 Weiterführende Themen

7.1 Singleton

Wie kann man sicherstellen, dass von einer Klasse nur maximal eine Instanz gebildet werden kann? Diese Frage taucht in vielen Projekten immer wieder auf. Wir hatten in unserem einfachen Personenbeispiel den Personenzähler eingeführt und diesen bei jeder Instanziierung erhöht. Dieser Ansatz ist jedoch nur solange sinnvoll, wie wir uns auf diesen einfachen Personenzähler beschränken. Eine etwas kompliziertere „Anmeldung“ eines neu erzeugten Objektes, beispielsweise unter Erzeugung einer ID-Nummer, sollte nicht in der Klasse Person erfolgen, sondern in einer dafür spezialisierten Klasse. So könnten wir in unserer Applikation ein zentrales „Einwohnermeldeamt“ verwenden, das die Neuanmeldung übernimmt. Wir haben nun jedoch das Problem, dass wir bei der Erzeugung einer Person eine Referenz auf das Einwohnermeldeamt mit übergeben müssten oder aber die Klasse nur mit statischen Attributen oder Methoden ausstatten. Beide Alternativen sind alles andere als überzeugend: Das Einwohnermeldeamt ist unbestreitbar ein Objekt mit Methoden, es soll aber nur eines geben. Die erste Alternative führt in kurzer Zeit zu mächtigen Parameterlisten.

Es gibt nun eine Reihe von Musterproblemstellungen und Musterlösungen, die in einer objektorientierten Applikation verwendet werden können. Diese Sammlung von Musterlösungen haben den Begriff „Design Pattern“ (Entwurfsmuster). Eines dieser Muster hat den Namen „Singleton“ und ist für unsere eben definierte Problemstellung wunderbar geeignet.

Wir definieren die Klasse EinwohnerMeldeamt wie folgt:

Beispiel:

```
public class EinwohnerMeldeamt {  
    private static EinwohnerMeldeamt instance;  
    // weitere Attribute  
  
    private EinwohnerMeldeamt() {  
        // Initialisierung der Attribute  
    }  
  
    public static EinwohnerMeldeamt getInstance() {  
        if (instance == null) {
```

```

        instance = new EinwohnerMeldeamt();
    }
    return instance;
}

public void anmelden(Person neuePerson) {
    // Anmeldeformular ausfüllen...
}
}

```

Der Zugriff auf unser Einwohnermeldeamt ist nun aus allen Programnteilen heraus möglich über den Ausdruck:

EinwohnerMeldeamt.getInstance()

Die UML unseres Einwohnermeldeamtes zeigt die folgende Abbildung:

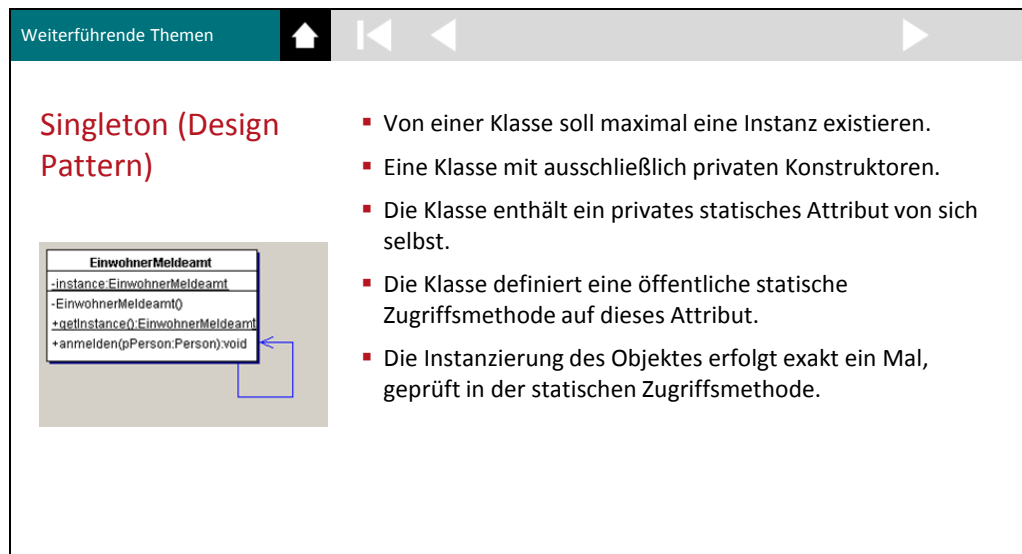


Abb. 7-1: Singleton

7.2 Assertions

Neben den `Exception`-Klassen bietet Java auch ein Schlüsselwort an, das während des Programm-Tests sehr sinnvoll eingesetzt werden kann: `assert`.

The screenshot shows a presentation slide with a title bar 'Weiterführende Themen' and navigation icons. The slide title is 'Assertions'. The content is a bulleted list:

- Schlüsselwort `assert`
- `assert` <boolescher Ausdruck>
- Ausdruck `true`: Fortführung des Programmlaufs
- Ausdruck `false`: Abbruch mit `AssertionError`
- Assertions sollen Programmierfehler der Anwendung signalisieren ("darf nie auftreten").
- Wenn eine Assertion nicht erfüllt ist, ist das Programm nicht mehr sinnvoll weiter zu führen.
- Gegensatz: Exceptions für Reaktionen auf Bedienungsfehler.
- Assertions können zur Laufzeit aktiviert oder deaktiviert werden
- `-enableassertions` oder kurz `-ea`
- `-disableassertions`, oder kurz `-da`
- Steuerung bis auf Klassenebene möglich
- `java -ea:de.integrata.java.sprache.assertions.AssertDemo`

Abb. 7-2: Assertion

Eine Assertion, zu Deutsch: Behauptung, wird ähnlich wie eine `if`-Abfrage geschrieben:

```
assert <Bedingung>;
```

Ist die Bedingung erfüllt, so wird die nächste Programmzeile ausgeführt. Ist die Assertion jedoch falsch, bricht das Programm mit einem `AssertionError` ab.

Assertions wurden erst mit der Java-Version 1.4 eingeführt und es stellt sich die Frage, was diese Konstruktion denn soll.

Weiterführende Themen

Verwendung der Assertions

- Vorbedingungen prüfen
 - Assertion nur sinnvoll für private Methoden: Der Entwickler der Klasse verwendet eine interne Methode falsch
 - Bei öffentlichen Methoden: Exceptions für Reaktion auf externe Fehlbenutzung
- Nachbedingungen prüfen
 - Assertion: Unvorhergesehener Programmablauf führt zu inkonsistentem Objektzustand.
 - Exception: Fehlerhafte Benutzung wird dem Aufrufer mitgeteilt.

Abb. 7-3: Verwendung von Assertions

Assertions sollen einen klaren Fehler der Anwendung signalisieren. Nach einer Assertion ist das Programm nicht mehr sinnvoll weiter zu führen. Damit würde sich das Werfen einer speziellen `RuntimeException` anbieten, die dann jedoch in der normalen Fehlerbehandlung untergehen würde. Die Verletzung einer Assertion muss beim Test des Programms ein Error sein!

Daneben sind Assertions tief in die Laufzeitumgebung integriert worden. Assertions müssen nach erfolgreichem Test von der Virtuellen Maschine nicht mehr beachtet werden. Deshalb wurde der Aufruf der Virtuellen Maschine durch die Optionen `-ea` (enable assertions) und `-da` (disable assertions) erweitert. Beim Programmstart können Assertions bis hin zur Klassenebene aktiviert oder deaktiviert werden:

```
java -ea:de.integrata.java.sprache.assertions.AssertDemo
```

7.3 Wrapper-Klassen

Der Begriff des Wrappers ist wiederum ein Entwurfsmuster wie das Singleton. Ein Wrapper ist eine Klasse, die einen anderen Datentyp als Attribut enthält und den Zugriff auf dieses regelt bzw. mit neuen Methoden erweitert. Die Wrapper-Klassen aus `java.lang` werden verwendet, wenn einfache Datentypen wie `short`, `boolean`, `int`, `double` usw. als Objekte behandelt werden sollen. Diese Klassen umfassen nützliche Werkzeuge und vor allem Konvertierungsmethoden.

The screenshot shows a presentation slide with a title bar 'Weiterführende Themen' and navigation icons. The slide content is as follows:

Wrapper-Klassen

- Zu einfachen Datentypen entsprechende Wrapper-Klassen:

<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>char</code>	<code>Character</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>
- Eigenschaften und Verwendung
 - enthalten den einfachen Typ als Attribut
 - stellen Konvertierungsmethoden zur Verfügung
 - ermöglichen call-by-reference für die einfachen Datentypen
- Seit der Sprachversion 5.0 konvertiert der Compiler automatisch zwischen den einfachen Datentypen und den Wrapper-Klassen
 - „Autoboxing/Unboxing“

Abb. 7-4: Wrapperklassen

Die Klassennamen der Wrapper-Klassen ergeben sich aus den elementaren Datentypen, indem in Konsistenz mit der Namenskonvention der erste Buchstabe groß geschrieben wird. So wird z. B. aus dem elementaren Datentyp `double` die Wrapper-Klasse `java.lang.Double`. Ausnahmen dieses Verfahrens sind die Datentypen `int` und `char`;

`int` wird in die Klasse `java.lang.Integer`, und

`char` in `java.lang.Character` abgebildet.

Die Wrapperklassen (`Integer`, `Long`, `Character`, `Double`, ...) erlauben, einfache Datentypen (`int`, `double`, ...) als Objekte zu behandeln. Der einfache Datentyp stellt den Wert eines Objektes dar. Der Datentyp wird von einer Objekthülle "umwickelt".

Im folgenden Beispiel wird der Gebrauch der Wrapper-Klassen am Beispiel `Integer` exemplarisch aufgezeigt:

Es wird eine gewöhnliche Integergröße `wert` und eine Instanz `x` vom Typ der Wrapper-Klasse `Integer` angelegt:

```
int wert;  
Integer x;
```

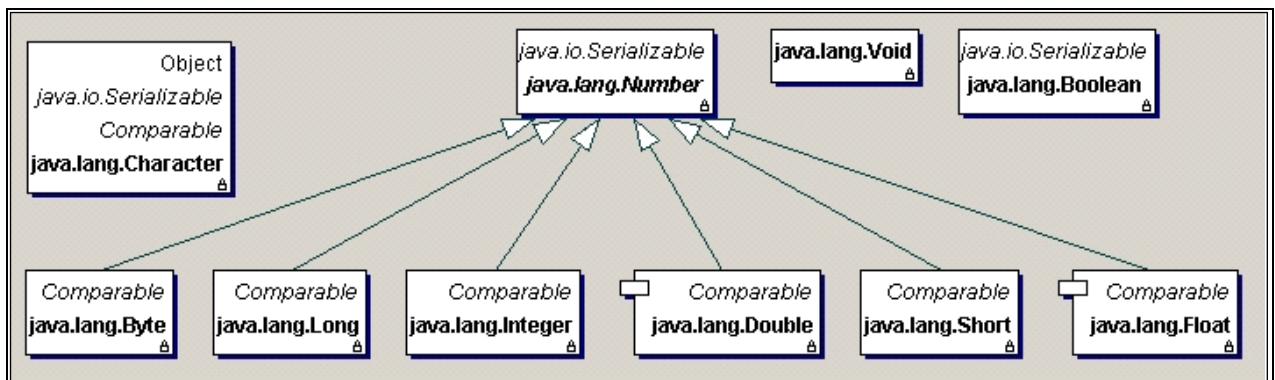
Im Wrapper gibt es eine Methode `valueOf()`, um einen String in ein Objekt vom Typ `Integer` umzuwandeln:

```
x = Integer.valueOf( "123" );
```

Des Weiteren gibt es eine Methode namens `intValue()`, um ein Objekt vom Typ `Integer` in einen `int`-Wert zu konvertieren:

```
i = x.intValue();
```

Gemäß dem hier Aufgeführten, implementieren die anderen Wrapper-Klassen ebenfalls entsprechend die statischen Klassenmethoden `valueOf()` bzw. `typeNameValue`, wobei `typeName` durch den Namen des jeweiligen elementaren Datentyps zu ersetzen ist, also z. B. `doubleValue()`, `booleanValue()`...



7.3.1 Autoboxing/Unboxing

Seit der Sprachversion 5.0 konvertiert der Java Compiler automatisch bei Bedarf zwischen den einfachen Datentypen und den Wrapper-Klassen. So sind die beiden folgenden Zeilen syntaktisch korrekt:

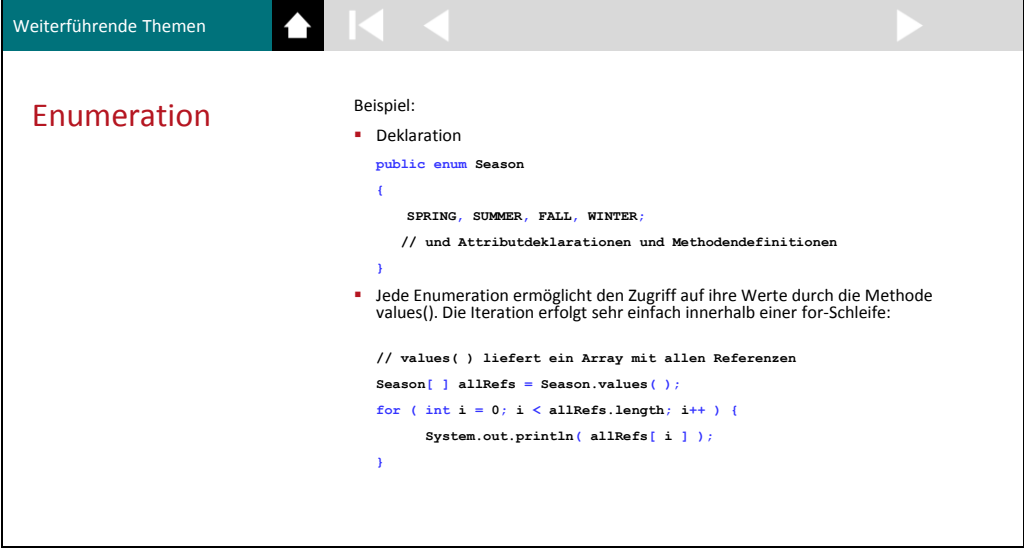
```
Integer wrapper = 5;
```

```
int primitive = new Integer(5);
```

Dieses Verfahren wird als „Autoboxing/Unboxing“ bezeichnet und ist natürlich auch bei der Signatur von Methoden (Rückgabewerte und Parameter) erlaubt.

7.4 Enumeration

Eine diskrete Menge von Objekten eines bestimmten Typs wird als Enumeration bezeichnet.



The screenshot shows a presentation slide with a title bar 'Weiterführende Themen' and navigation icons. The main content is titled 'Enumeration' in red. It includes a 'Beispiel:' section with two bullet points. The first bullet point is 'Deklaration' followed by a code block for a public enum Season with values SPRING, SUMMER, FALL, and WINTER. The second bullet point states that every enumeration allows access to its values via the values() method and provides a simple iteration example using a for-loop.

Enumeration

Beispiel:

- Deklaration

```
public enum Season
{
    SPRING, SUMMER, FALL, WINTER;
    // und Attributdeklarationen und Methodendefinitionen
}
```

- Jede Enumeration ermöglicht den Zugriff auf ihre Werte durch die Methode values(). Die Iteration erfolgt sehr einfach innerhalb einer for-Schleife:

```
// values() liefert ein Array mit allen Referenzen
Season[] allRefs = Season.values();
for (int i = 0; i < allRefs.length; i++) {
    System.out.println( allRefs[ i ] );
}
```

Abb. 7-5: enum

Diese werden in Java durch das Schlüsselwort **enum** definiert:

```
public enum Season {
    ...
}
```

Die diskreten Werte der Enumeration werden nun einfach in die Definition als „Konstanten“ eingetragen:

```
public enum Season {
    SPRING, SUMMER, FALL, WINTER;
}
```

Auf die Werte kann dann natürlich z.B. in der Form `Seasons.FALL` zugegriffen werden und - das ist ein großer Vorteil der Enumeration - auch innerhalb einer `switch`-Anweisung.

Ansonsten kann eine Enumeration wie eine normale Klasse Attribute, Konstruktoren und Methoden deklarieren, wobei der Konstruktor nicht `public` sein kann.

```
public enum Season {
    SPRING("Spring"), SUMMER("Summer"),
    FALL("Fall"), WINTER("Winter");

    private double averageTemperature;
    private String name;

    Season(String name) {
        this.name = name;
    }

    public void setAverageTemperature(double average) {
        averageTemperature = average;
    }

    public double getAverageTemperature() {
        return averageTemperature;
    }

    public String getName() {
        return name;
    }
}
```

Jede Enumeration ermöglicht den Zugriff auf ihre Werte durch die Methode `values()`. Die Iteration erfolgt sehr einfach innerhalb einer `for`-Schleife:

```
// values() liefert ein Array mit allen Referenzen

Season[] allRefs = Season.values();
for (int i = 0; i < allRefs.length; i++) {
    System.out.println(allRefs[i]);
}
```

7.5 Variable Argumentlisten

Soll eine Methode eine beliebige Menge von Parametern übertragen bekommen, bietet sich ein Array, eine Liste oder eine Map als Parameter-Typ an. Diese Art der Übergabe ist aber für den Anwendungsprogrammierer nicht sonderlich komfortabel, da vor jeden Methodenaufruf erst einmal das Parameter-Objekt erzeugt werden muss. Dieser Mangel wird durch die variablen Argumentlisten beseitigt. Dabei ist zu unterscheiden zwischen der Implementierung einer Methode, die eine Parameterliste variabler Länge erwartet, und deren Aufruf.

Implementierende Methode

Das folgende Beispiel deklariert eine Methode, die einen normalen Parameter und eine variable Argumentliste erwartet:

```
public void dump(String header, Object... varargs)
```

Der Typ von `varargs` ist `Object[]` und kann innerhalb der Implementierung beliebig verwendet werden:

```
public void dump(String header, Object... varargs) {
    System.out.println(header);
    for(Object o:varargs) {
        System.out.println(o);
    }
}
```

Aufruf

Das aufrufende Programm sieht nicht das verwendete Objekt-Array, sondern liefert einfach eine beliebig lange Parameterliste:

```
package de.integrata.jdk5.varargs;
public class VarargsDemo {
    public static void main(String[] args) {
        new VarargsDemo();
    }
    public VarargsDemo() {
        dump("*** NONE ***");
        dump("*** One ***", "One");
        dump("*** Three ***", "One", 2, this);
    }

    public void dump(String header, Object... varargs) {
        System.out.println(header+" Param="+ varargs.length);
        for(Object o:varargs) {
            System.out.println(o);
        }
    }
}
```

7.6 Metadaten

Java erlaubt die Angabe von Metadaten. Metadaten also "Daten über Daten" bezogen auf die Programmierung in Java bedeutet, dass der Quellcode um Informationen erweitert wird, die nicht der eigentlichen Programmiersprache zuzuordnen sind. Bestehende Mechanismen zur Angabe solcher Daten sind z.B. JavaDoc-Kommentare (`@version`, `@author`, ...), XDoclet-Kommentare, die Reflection-API-Konventionen bzw. oft im Serverbereich verwendete Deployment-Deskriptoren.

Motivation und Anforderungen an Metadaten

Die bisher unterschiedlichen Methoden zur Angabe von Metadaten und der wachsende Wildwuchs verlangten nach einer Standardisierung. Im Java Specification request (JSR) 175 wurden hierzu Vorschläge erarbeitet, die zur Vereinfachung für den Entwickler und zum Wegfall spezieller Werkzeuge zur Verarbeitung von Metadaten führt.

Anforderungen für Metadaten:

- Definition und Erweiterbarkeit durch den Entwickler
- Benutzung zur Design- und zur Laufzeit
- Bestandteil der Sprachdefinition, also durch den Compiler prüfbar
- Anwendbar auf Pakete, Klassen, Methoden und Attribute
- Definition ohne zusätzliche Methoden bzw. Felder im Code
- Bestehende Metadatenangaben (JavaDoc-Kommentare, `transient` bzw. `java.io.Serializable`) werden nicht beeinflusst

Anwendungen für Metadaten sind z.B. gezielte Unterdrückung von Compilerwarnungen, generische Persistenzmechanismen, Test-Frameworks, RPC/RMI Interface-Generatoren bzw. EJB 3.0.

7.6.1 Vordefinierte Annotations und Meta-Annotations

7.6.1.1 Meta Annotations

Neben der Möglichkeit, eigene Metadaten zu definieren, existieren eine Reihe von vordefinierten Annotations und Meta-Annotations. Meta-Annotations sind – analog zu Daten von Daten – Annotations von Annotations. Es sind vier Meta-Annotations vordefiniert, die für den Compiler und Javadoc eine bestimmte Bedeutung haben.

Target

Erlaubt die Definition derjenigen Programmelemente, bei dem der spezifizierte Annotation-Typ erlaubt ist. Folgendes Listing zeigt die Aufzählung der erlaubten Element-Typen:

Retention

Durch diese Meta-Annotation wird die Lebensdauer des spezifizierten Annotation-Typs bestimmt. Es sind folgende Optionen erlaubt:

- **SOURCE** – Die Annotation wird nur in der aktuellen Source-Datei verwendet, wird also nicht in die compilierte Klasse übernommen.
- **CLASS** – Die Annotation wird vom Compiler in die generierte .class-Datei geschrieben, sie wird jedoch nicht zur Laufzeit ausgewertet.
- **RUNTIME** – Diese Annotation wird zur Laufzeit über die Reflection-API auslesbar zur Verfügung gestellt.

Documented

Eine Marker-Annotation, die definiert, dass diese Daten in der Javadoc Dokumentation erscheinen.

Inherited

Inherited ist eine weitere Marker-Annotation, die selten genutzt wird. Sie kann bei einer eigenen Annotation benutzt werden, um eine Klasse zu markieren, die noch in Bearbeitung ist. Falls diese durch die `Documented`-Annotation markiert ist, wird sie in der Javadoc angezeigt.

7.6.1.2 Vordefinierte Annotations

Override

Durch die Annotation `Override`, die nur bei Methoden verwendet wird, wird definiert, dass es bei der Methode um eine überschriebene Methode der Superklasse handelt. Der Compiler kann bei falscher Schreibweise einen Fehler anzeigen.

Deprecated

`Deprecated`, auch eine Marker-Annotation, bei Packages, Klassen, Interfaces und Methoden, kennzeichnet diese als ungültig (unerwünscht) und der Compiler generiert Warnungen, wenn sie trotzdem verwendet werden.

SuppressWarnings

Diese äußerst wichtige Annotation wird benutzt, um Compiler-Warnings zu unterdrücken. Wenn z.B. Code für die Compiler-Versionen 1.4 und 1.5 erzeugt werden soll, werden bei allen nicht generischen Operationen `type-safe`-Warnings erzeugt.

7.6.2 Deklaration eigener Metadaten und Tool `apt`

Neben den vordefinierten Metadaten können auch eigene Metadaten definiert werden. Metadaten werden ähnlich einem Interface deklariert, erben implizit von `java.lang.annotation.Annotation`. Die Deklaration eigener Metadaten soll an dieser Stelle nicht vertieft werden.

Für die Verarbeitung von Annotations ist im JDK ein neues Tool mit Namen `apt` enthalten. Es erlaubt die Auswertung von Annotations von der Kommandozeile aus und ermöglicht z.B. eigene Implementierungen zur Generierung von Java-Sourcecode einsetzen.

8

Klassenbibliothek

8.1	Die Klassenbibliothek	8-3
8.2	Zeichenketten	8-4
8.2.1	Klasse String	8-4
8.2.2	Klasse StringBuilder und Klasse StringBuffer	8-7
8.3	Klasse Object	8-9
8.3.1	Die equals()-Methode	8-9
8.3.2	Die toString()-Methode	8-11
8.3.3	Die clone()-Methode	8-11
8.4	Einige Klassen des Pakets java.util	8-13
8.5	System Properties	8-14
8.6	Formatierte Ausgaben	8-15
8.7	Eine weitere Auswahl aus der Klassenbibliothek	8-16
8.8	Online-Dokumentation	8-17

8 Klassenbibliothek

8.1 Die Klassenbibliothek

Die kleinste Einheit eines Bytecode-Blockes enthält im Wesentlichen die Definition von Konstanten, Variablen und die ausführbaren Programmmanweisungen. Dies entspricht in etwa dem Ansatz prozeduraler Programmiersprachen, Funktionen und Prozeduren in Modulen zu gruppieren. Aus dem objektorientierten Ansatz der Programmiersprache heraus wird diese atomistische Bytecode-Einheit "Klasse" genannt. Eine Anwendung besteht in der Regel aus mehreren korrespondierenden und abhängigen Klassen. Die Virtuelle Maschine wird solche Klassen stets dynamisch zur Laufzeit nach Bedarf laden. Ein statisches Linken mehrerer Klassen zu einer Anwendung ist nicht möglich.

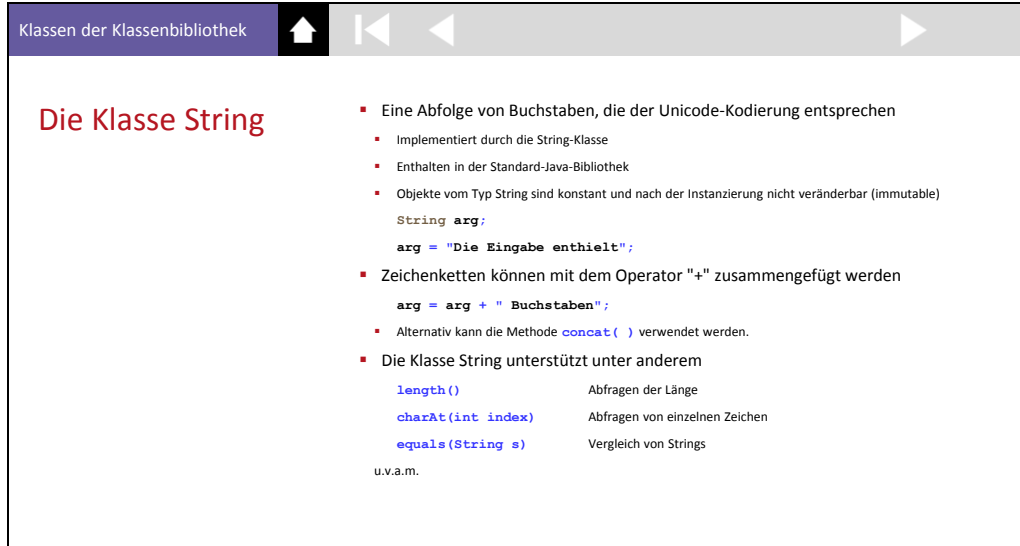
Die Laufzeitumgebung enthält einen Satz von Standard-Klassen, die prinzipiell jedem eigen entwickelten Programm zur Verfügung gestellt werden können. Aufgrund des dynamischen Ansatzes beim Laden von Klassen und der Mächtigkeit der Programmiersprache Java ist die Anzahl der ausgelieferten Klassen enorm: Die Java Standard Edition (Java SE) enthält mittlerweile tausende Klassen. Es ist offensichtlich, alleine aus Gründen der Übersichtlichkeit eine weitere Gruppierung der Java-Klassen vorzusehen. Dies sind die sogenannten "Pakete", in diesem Kontext aufzufassen als Container für Klassen und weitere Unterpakete. Mit Hilfe der Pakete können Klassen in eine baumähnliche Hierarchie eingeordnet werden.

In einem Dateisystem werden Pakete in Verzeichnisse abgebildet, Klassen in Dateien mit der Endung `.class`. Diese Abbildung ist jedoch für eine Installation einer Anwendung sehr unpraktisch und fehleranfällig. Es ist aber auch möglich und üblich, die Verzeichnisstruktur in eine einzige Datei im üblichen ZIP-Format zu verpacken. Enthält diese Datei noch Java-spezifische Metadaten, so spricht man von einem Java-Archiv mit der Endung `.jar`. Diese Strategie wird auch bei der Klassenbibliothek der Java-Laufzeitumgebung angewendet: Im `lib`-Unterverzeichnis der Java-Laufzeitumgebung ist die Datei `rt.jar` zu finden. Geöffnet mit einem Zip-Tool ist die Verzeichnisstruktur der Java-Klassen sichtbar.

Wie bereits erwähnt, ist die Herkunft der Klassendateien beliebig. Die Java-Laufzeitumgebung ist in der Lage, Daten aus dem Dateisystem zu lesen. Die Erweiterung auf das Laden von Klassen über das Netzwerk ist jedoch einfach möglich.

8.2 Zeichenketten

8.2.1 Klasse String



Die Klasse String

- Eine Abfolge von Buchstaben, die der Unicode-Kodierung entsprechen
- Implementiert durch die String-Klasse
- Enthalten in der Standard-Java-Bibliothek
- Objekte vom Typ String sind konstant und nach der Instanzierung nicht veränderbar (immutable)


```
String arg;
arg = "Die Eingabe enthielt";
```
- Zeichenketten können mit dem Operator "+" zusammengefügt werden


```
arg = arg + " Buchstaben";
```
- Alternativ kann die Methode `concat()` verwendet werden.
- Die Klasse String unterstützt unter anderem

<code>length()</code>	Abfragen der Länge
<code>charAt(int index)</code>	Abfragen von einzelnen Zeichen
<code>equals(String s)</code>	Vergleich von Strings

 u.v.a.m.

Abb. 8-1: String

int length() liefert die Länge eines Strings.

char charAt(int pos) liefert das Zeichen an der Position pos.

boolean equals(String s)

gibt "true" zurück, wenn die beiden Strings den selben Inhalt haben, also wenn die Längen, die Zeichen und deren Reihenfolgen in den Strings identisch sind.

boolean equalsIgnoreCase(String s)

wie zuvor, aber unabhängig von Groß- oder Kleinschreibung.

String trim()

entfernt führende und endende Leerzeichen.

int compareTo(String s)

Vergleich zweier Stringobjekte. Zurückgegeben wird ein Wert, der positiv ist, wenn das übergebene String-Argument in der Sortierreihenfolge vor dem aufrufenden String-Objekt kommt, null, wenn beide gleich sind, und negativ, wenn das Argument nach dem aufrufenden Objekt kommt.

```
boolean regionMatches(int toffset, String other,  
                      int ooffset, int len)
```

reicht "true" zurück, wenn der Bereich des einen Strings mit dem Bereich des anderen String übereinstimmt. Die Parameter sind:

toffset	Startposition im laufenden String
other	der andere String
ooffset	Startposition im anderen String (other)
len	Anzahl der zu vergleichenden Zeichen

```
String substring( int start, int ende )
```

reicht den String zwischen Start- und Ende-Position zurück, das Zeichen an der Position ende ist nicht mit enthalten.

```
String concat( String s )
```

Stringverkettung, ähnlich +-Operator.

```
String toLowerCase()
```

Umwandlung von Groß- nach Kleinbuchstaben.

```
String valueOf(Datentyp x)
```

liefert einen String, welcher die Darstellung des Wertes vom Typ x hat. Datentyp kann sein: `int`, `float`, `double`, `long`, `char`, `boolean` oder `Object`.

Das folgende Beispiel verdeutlicht die Anwendung dieser Methoden:

Beispiel:

```
class String1 {  
    public static void main( String [] args ){  
        String s1 = new String("Java and Coffee");  
        String s2 = new String("Java und Capuccino");  
        System.out.println("  Länge s1:" + s1.length() );  
        char c;  
        c = s1.charAt(2);  
        System.out.println("  3. Zeichen :" + c );  
  
        boolean b;  
        b = s1.equals( s2 );  
        System.out.println("  Vergleich 1: " + b );  
  
        b = s1.equalsIgnoreCase( "java and coffee" );
```

```
System.out.println("\tVergleich 2: " + b );
int i;
    i = s1.compareTo("Java and Coffee");
    System.out.println("\tVergleich 3: " + i );
    b = s1.regionMatches(0, s2, 0, 5 );
    System.out.println("\tVergleich 4: " + b );
    b = s1.regionMatches(0, s2, 0, 6 );
    System.out.println("\tVergleich 5: " + b );
    String s3 = s1.substring(3, 7);
    System.out.println("\tSubstring : " + s3 );

    s3 = s1.concat( s2 );
    System.out.println("\tVerkettung : " + s3 );

    s3 = s1.toLowerCase( );
    System.out.println("\ttoLower : " + s3 );

    String s4 = s3.toString();
    System.out.println("\tKopie : " + s4 );
    double x = -123.45;
    String s5 = s4.valueOf( x );
    System.out.println("\tValue of : " + s5 );
}
```

Die Ausgabe lautet:

Laenge s1: 15

3. Zeichen: v

Vergleich 1: false

Vergleich 2: true

Vergleich 3: 32

Vergleich 4: true

Vergleich 5: false

Substring: a an

Verkettung: Java and CoffeeJava und Capuccino

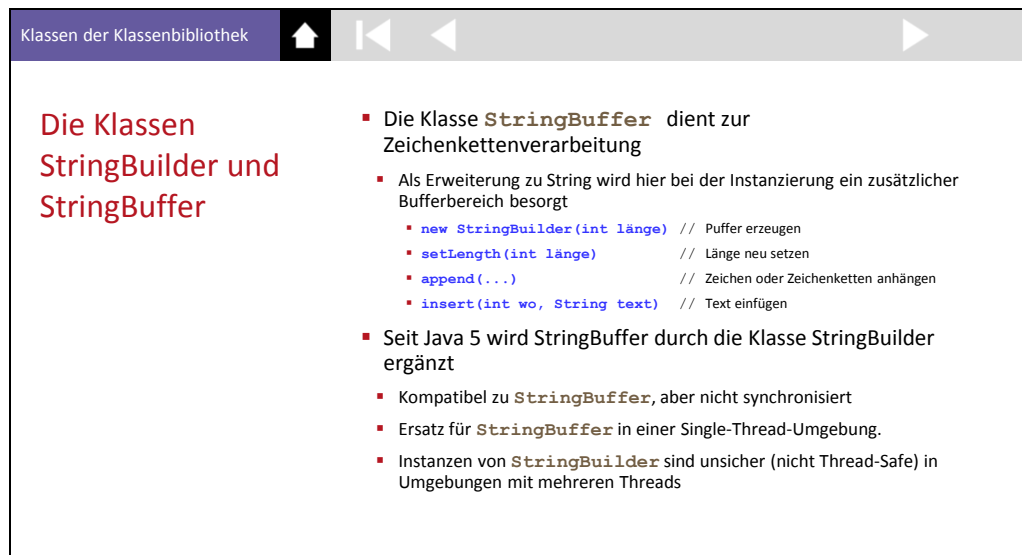
toLower: java and coffee

Kopie: java and coffee

Value of: -123.45

8.2.2 Klasse `StringBuilder` und Klasse `StringBuffer`

Strings sind immutable und somit unveränderbar. Dies hat zur Folge, dass bei jeder Verknüpfung von Strings mit dem `+` Operator ein neuer String erzeugt wird. Dies ist nicht gerade performant.



Klassen der Klassenbibliothek

Die Klassen `StringBuilder` und `StringBuffer`

- Die Klasse `StringBuffer` dient zur Zeichenkettenverarbeitung
- Als Erweiterung zu String wird hier bei der Instanzierung ein zusätzlicher Bufferbereich besorgt
 - `new StringBuilder(int länge)` // Puffer erzeugen
 - `setLength(int länge)` // Länge neu setzen
 - `append(...)` // Zeichen oder Zeichenketten anhängen
 - `insert(int wo, String text)` // Text einfügen
- Seit Java 5 wird `StringBuffer` durch die Klasse `StringBuilder` ergänzt
 - Kompatibel zu `StringBuffer`, aber nicht synchronisiert
 - Ersatz für `StringBuffer` in einer Single-Thread-Umgebung.
 - Instanzen von `StringBuilder` sind unsicher (nicht Thread-Safe) in Umgebungen mit mehreren Threads

Abb. 8-2: `StringBuffer` und `StringBuilder`

Verwenden Sie anstelle eines Strings einen `StringBuilder` oder einen `StringBuffer`, verschwindet das Geschwindigkeits-Problem.

```
public class Test {  
  
    public static void main(String[] args) {  
  
        StringBuilder str = new StringBuilder( );  
        for (int i = 0; i < 100000; i++) {  
            str.append(i);  
        }  
    }  
}
```

Aber sollten Sie deshalb immer einen `StringBuilder` oder `StringBuffer` anstelle eines Strings verwenden? Nein! Denn Strings werden vom Compiler automatisch in `StringBuffer` bzw. `StringBuilder` umgewandelt.

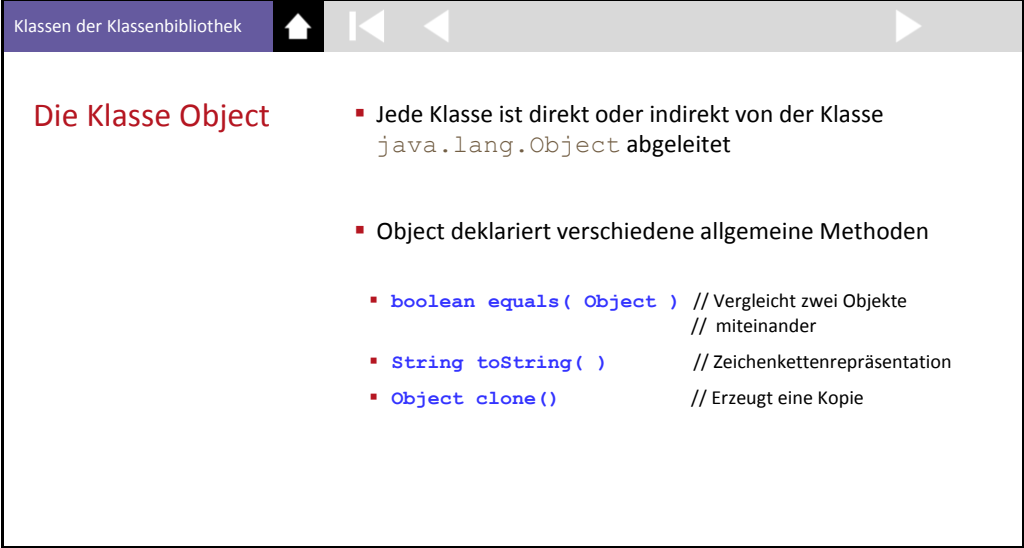
Sie sollten immer dann auf diese beiden Alternativen ausweichen, wenn es wichtig ist, dass Ihre Zeichenkette **dynamisch veränderbar** (z. B. in einer Schleife) ist, oder wenn Sie auf deren zusätzlichen Methoden wie z. B. `insert` (zusätzlichen Inhalt an eine beliebigen Stelle einfügen), `setCharAt` (ein Zeichen an einer bestimmten Stelle ersetzen), oder `reverse` (umkehren der Zeichenkette) zugreifen müssen/möchten.

`StringBuilder` und `StringBuffer` dienen beide demselben Zweck, nämlich um lange Strings zusammenzubauen, ohne dass bei jedem Konkatinierungsschritt ein neuer String instanziiert werden muss (Vermeidung des `new Operators`).

Der Unterschied zwischen `StringBuffer` und `StringBuilder` ist, dass `StringBuilder` unsynchronisiert. Das heißt, dass `StringBuffer` selbst sicherstellt, dass niemals ein parallel laufender Thread die Daten im `StringBuffer` inkonsistent macht. Das erkaufte man sich aber mit leichten Performanceeinbußen, sodass es sich lohnt `StringBuilder` zu verwenden wenn man weiß, dass nur ein Thread auf die `StringBuffer`-Instanz zugreifen wird.

8.3 Klasse Object

Die Klasse `java.lang.Object` ist die (implizite) Basisklasse aller anderen Klassen und bildet die Spitze der Klassenhierarchie, die in `java.lang.Object` definierten Methoden werden somit von allen Klassen geerbt.



Klassen der Klassenbibliothek

Die Klasse Object

- Jede Klasse ist direkt oder indirekt von der Klasse `java.lang.Object` abgeleitet
- Object deklariert verschiedene allgemeine Methoden
- `boolean equals(Object)` // Vergleicht zwei Objekte
// miteinander
- `String toString()` // Zeichenkettenrepräsentation
- `Object clone()` // Erzeugt eine Kopie

Abb. 8-3: Object

8.3.1 Die equals()-Methode

In Java-Programmen wird mit Objektreferenzen, nicht mit den Objekten selbst gearbeitet. Dies muss auch beim Vergleich zweier Objekte berücksichtigt werden, was folgendes Codefragment zeigen soll:

Beispiel:

```
Person p1 = new Person("Hans", "Meier"),
        p2 = new Person("Hans", "Meier");

if ( p1 == p2 )
    System.out.println("Personen sind identisch");
else
    System.out.println("Personen sind nicht gleich");
```

Die Ausgabe des Programms ist „Personen sind nicht gleich“, da die Objektreferenzen natürlich auf verschiedene Speicherbereiche zeigen.

Ein Vergleich auf gleichen Inhalt zweier Objekte ermöglicht die Methode `equals`.

Diese Methode ist in der Klasse `Object` definiert und liefert in der Standard-Implementierung nichts anderes zurück als

```
public boolean equals (Object o) {
```

```
        return this == o;
    }
```

also nichts anderes, als den Referenz-Vergleich.

Eigene Klassen können selbstverständlich die equals()-Methode auf geeignete Art überschreiben (überschatten), um z. B. den Vergleich auf gleichen Vor- und Nachnamen zu prüfen:

```
public boolean equals(Object obj) {
    Person pTemp = null;
    if(obj instanceof Person)
        pTemp = (Person)obj;
    else
        return false; // keine Person-Instanz

    if( super.equals( pTemp ) )
        return true;
    else
        if( this.vorname.equals( pTemp.vorname )
            && this.nachname.equals( pTemp.nachname ) )
            return true;
        else
            return false;

} //equals(Object)
```

Damit lässt sich das obige Beispiel umformulieren zu:

```
Person p1 = new Person("Hans", "Meier"),
        p2 = new Person("Hans", "Meier");
if ( p1.equals( p2 ) ){
    System.out.println("Personen sind gleich");
}
else{
    System.out.println("Personen sind nicht gleich");
}
```

Mit der Ausgabe „Personen sind gleich“. Das hier definierte Kriterium für die Objektgleichheit, nämlich die Gleichheit von Vor- und Nachnamen, ist in der realen Anwendung in dieser Form sicherlich zu bevorzugen.

8.3.2 Die toString() -Methode

Die `toString()`-Methode erzeugt eine Zeichenkettendarstellung des Zustands des aktuellen Objekts (desjenigen Objekts, für welches diese Methode aufgerufen wird). Sie sollte in eigenen Klassen auf geeignete Art überschrieben werden.

```
public String toString() {  
    return "Die Person " + this.getName();  
}
```

8.3.3 Die clone() -Methode

Die `clone()`-Methode der Klasse `Object` kann eine Kopie eines Objektes erzeugen. Zu beachten ist dabei, dass zur korrekten Verwendung der `clone()`-Methode jede Klasse, deren Instanzen kopiert werden sollen, die Schnittstelle `java.lang.Cloneable` implementieren und den Rückgabewert korrekt casten muss. Ist das `Cloneable`-Interface nicht implementiert, wird die `clone()`-Methode aus `Object` die `java.lang.CloneNotSupportedException` werfen.

Eine Überschattung der `clone()`-Methode ist in der Regel immer notwendig, da die `clone()`-Methode in `java.lang.Object` als `protected` deklariert wurde und folglich ein Aufruf nur in Subklassen und dem Paket `java.lang` heraus möglich ist.

Eine Alternative hierzu ist die Definition einer unabhängigen Methode in der Subklasse, die intern `clone()` verwendet.

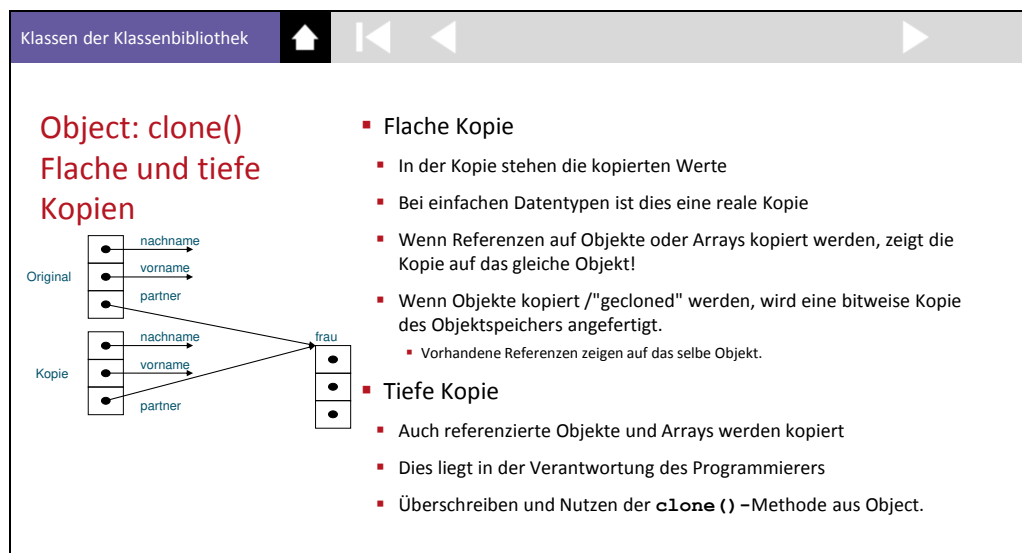


Abb. 8-4: Object: clone()

Beispiel:

```
public Object clone() throws CloneNotSupportedException {  
    return super.clone();  
}
```

Hier ist zur Verdeutlichung die `clone()`-Methode überschattet, es wird nur die Standard-Funktionalität aufgerufen.

Die `clone()`-Methode der Klasse `Object` macht nun aber nichts anderes als eine **flache** Kopie, d. h. die Attribute werden als Wert in den Klon kopiert. Wollen wir stattdessen eine **tiefe** Kopie erzeugen, müssen wir die `clone()`-Methode mit Funktionalität hinterlegen, beim Beispiel der Person mit Partner hätten wir:

```
public Object clone() throws CloneNotSupportedException {  
    Person lKlon = (Person) super.clone();  
    lKlon.partner = (Person) lKlon.partner.clone();  
    lKlon.partner.cNachname = this.partner.cNachname;  
    lKlon.partner.cVorname = this.partner.cVorname;  
    lKlon.partner.partner = lKlon;  
    return lKlon;  
}
```

Hinweis: Diese Methode führt so kodiert zu einer Endlos-Rekursion. Eine konkrete Implementierung würde eine "boolean isCloning"-Flagge benötigen. Zu bevorzugen ist die Einführung einer eigenen Methode, z. B.:

```
public Person deepCopy() throws CloneNotSupportedException  
{  
    Person lKlon = (Person) super.clone();  
    lKlon.partner = (Person) lKlon.partner.clone();  
    lKlon.partner.cNachname = this.partner.cNachname;  
    lKlon.partner.cVorname = this.partner.cVorname;  
    lKlon.partner.partner = lKlon;  
    return lKlon;  
}
```

8.4 Einige Klassen des Pakets java.util

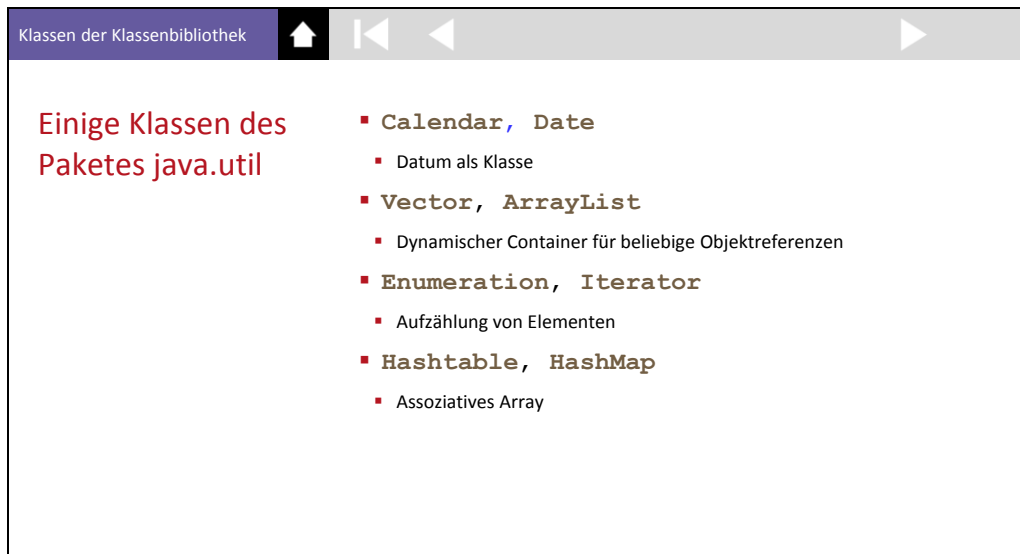


Abb. 8-5: Klassen aus java.util

8.5 System Properties

Klassen der Klassenbibliothek
🏠 ⏪ ⏩ ▶

Konfiguration mit System-Properties

- Neben den Aufrufparametern der main-Methode können auch System-Properties gesetzt und gelesen werden
- Auslesen durch
`System.getProperty(String propertyName)`
- Setzen durch Aufruf mit der Option „-D“
`java -Dkey=value ...`
- Hinweis: Auch eigene Properties können verwendet werden
- `java.util.Properties` Lesen/Schreiben aus Dateien
- `java.util.prefs.Preferences` zum Lesen benutzerabhängiger Eigenschaften

Abb. 8-6: System-Properties

Neben den Aufrufparametern der `main`-Methode können auch System-Properties gesetzt und gelesen werden:

Auslesen durch

```
System.getProperty(String propertyName)
```

Setzen durch Aufruf mit der Option "-D"

```
java -Dkey = value ...
```

Auch andere Properties können verwendet werden

`java.util.Properties` zum Lesen aus Dateien.

`java.util.prefs.Preferences` zum Lesen benutzerabhängiger Eigenschaften.

Die Verwendung von Datei-basierten Properties verlangt jedoch die Kenntnis des `java.io`-Paketes, das hier nicht behandelt werden kann.

8.6 Formatierte Ausgaben

Dazu dient das Paket `java.text` mit den Klassen

`NumberFormat`

`DateFormat`

Länderspezifische Einstellungen werden durch vordefinierte Konstanten der Klasse `Locale` vorgenommen.



Abb. 8-7: Formatierte Ausgaben

Die Ausgabe lautet ausgeführt mit einem deutsch konfigurierten Betriebssystem:

1.234,56

1.234,56

Zu beachten ist der Tausender-Punkt und das Komma als Dezimaltrennzeichen.

In der amerikanischen Einstellung ergibt sich jedoch:

1,234.56

1.234,56

8.7 Eine weitere Auswahl aus der Klassenbibliothek

Im JDK sind mittlerweile tausende Klassen standardmäßig integriert. Die hohe Zahl der mitgelieferten Klassen bedingt natürlich auch eine entsprechende Vielfalt an Funktionalität. Die folgende Tabelle soll eine kurze, zugegebenermaßen subjektive und unvollständige Auswahl wichtiger Klassen der Bibliothek liefern:

<u>Klasse</u>	<u>Bibliothek</u>	<u>Beschreibung</u>
Button	java.awt	Schaltfläche
Choice	java.awt	Auswahlbox
Font	java.awt	Zeichensatz
Frame	java.awt	Rahmen für ein Programmfenster
Image	java.awt	Grafik
Label	java.awt	Textanzeige
List	java.awt	Listenfeld
TextField	java.awt	Texteingabefeld
Event	java.awt.event	Ereignis
InputStream	java.io	Einlesen von Daten
Reader	java.io	Einlesen von Textdateien
OutputStream	java.io	Ausgeben von Daten
Writer	java.io	Ausgeben von Textdateien
Exception	java.lang	Ausnahme
Math	java.lang	Mathematische Funktionen
Number	java.lang	Zahl
Runnable	java.lang	In einem eigenen Thread lauffähig
Socket	java.net	Netzwerkverbindung
Connection	java.sql	Verbindung zu einer Datenbank
ResultSet	java.sql	Ergebnis einer Datenbankabfrage
Statement	java.sql	Ausführen einer SQL-Abfrage
Dictionary	java.util	Sortierbare Liste
Set	java.util	Eine Menge ohne Duplikate
Stack	java.util	Stapel mit push- und pop-Methode
ZipFile	java.util.zip	Lesen eines ZIP-Files

Alle Klassen sind selbstverständlich mit entsprechender Funktionalität ausgestattet, die ein einfaches und komfortables Programmieren ermöglichen.

8.8 Online-Dokumentation

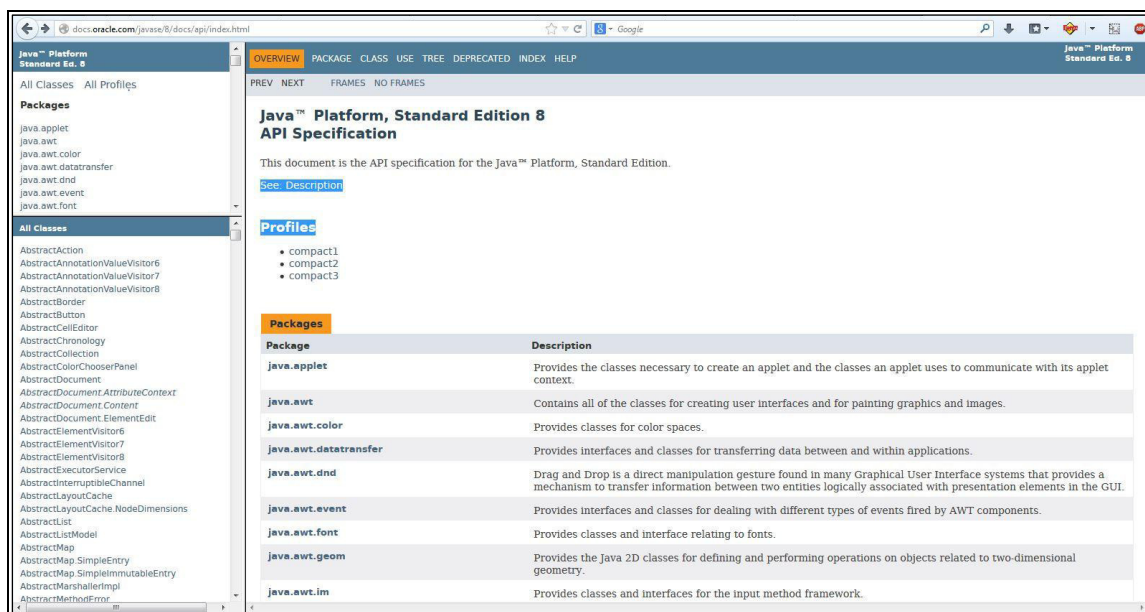
Klassen der Klassenbibliothek

Weitere Klassen

- Das JDK enthält die Dokumentation aller Klassen und Interfaces im HTML-Format
 - auf
 - <http://download.oracle.com/javase/7/docs/api/>
 - <http://download.oracle.com/javase/8/docs/api/>
 - <http://download.oracle.com/javase/9/docs/api/>
 - oder lokal installiert
 - `${Lokales-API-Installationsdirectory}\api\index.html`

Abb. 8-8: Weitere Klassen

Eine übersichtliche Darstellung und vollständige Dokumentation der Klassenbibliothek ist wie das JDK frei erhältlich. Sie kann auf dem Web-Server der Firma Oracle gelesen werden.



9

Literatur

9.1	Literatur und Webseiten	9-3
9.1.1	Allgemeine Literatur zu Java Grundlagen	9-3
9.1.2	Java im Netz	9-3
9.2	UML – Unified Modeling Language	9-4
9.2.1	UML-Literatur	9-4
9.2.2	Anwendungsfalldiagramm	9-5
9.2.3	Klassendiagramm	9-5
9.2.4	Paketdiagramm	9-6
9.2.5	Sequenzdiagramm	9-6

9 Literatur

9.1 Literatur und Webseiten

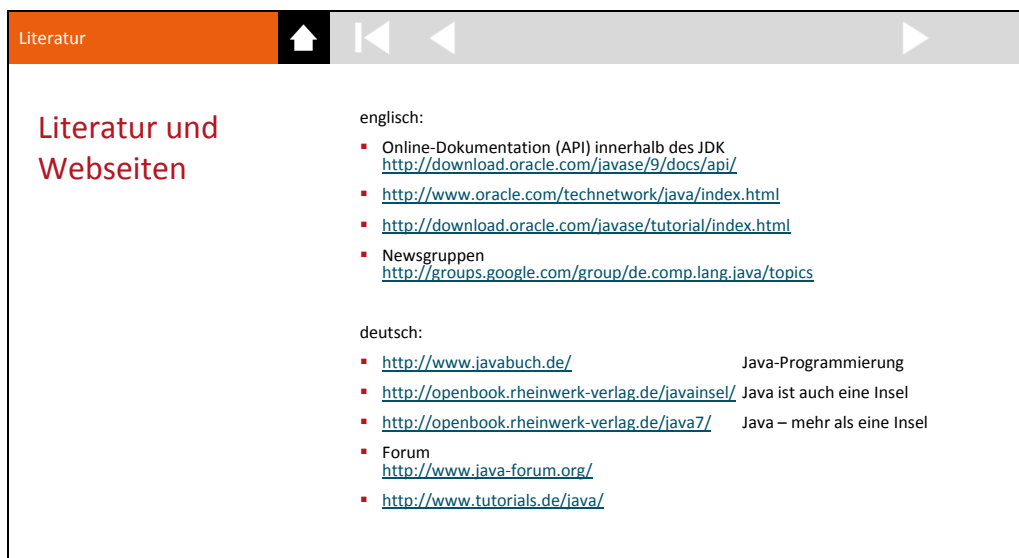


Abb. 9-1: Literatur

9.1.1 Allgemeine Literatur zu Java Grundlagen

Arnold, K. und Gosling, J.:

Die Programmiersprache Java, Addison-Wesley

Back, S., Beier, S., Bergius, K., Majorczyk, P.:

Professionelle Java Programmierung, Thomson

Cornell, G. und Horstmann, C.: *JAVA bis ins Detail*, Heise

Flanagan, D.: *Java in a Nutshell*, O'Reilly

Eckel, Bruce: *Thinking in Java*

9.1.2 Java im Netz

Deutsches Java-Forum

<http://www.java-forum.org>

Java FAQ von Eliotte Rusty Harold

<http://www.cafeaulait.org/javafaq.html>

Java Tutorial

<http://www.oracle.com/technetwork/java/index.html>

Thinking in Java von Bruce Eckel

<http://mindview.net/Books/TIJ4>

Handbuch der Java-Programmierung von Guido Krüger

<http://www.javabuch.de/>

Java ist auch eine Insel von Christian Ullenboom

<http://openbook.rheinwerk-verlag.de/javainsel/>

Java – mehr als eine Insel von Christian Ullenboom

<http://openbook.rheinwerk-verlag.de/java7/>

9.2 UML – Unified Modeling Language

UML ist ein Standard der OMG (<http://www.omg.org/uml>) und definiert eine Notation zur Visualisierung, Konstruktion und Dokumentation von Modellen für die objektorientierte Softwareentwicklung.

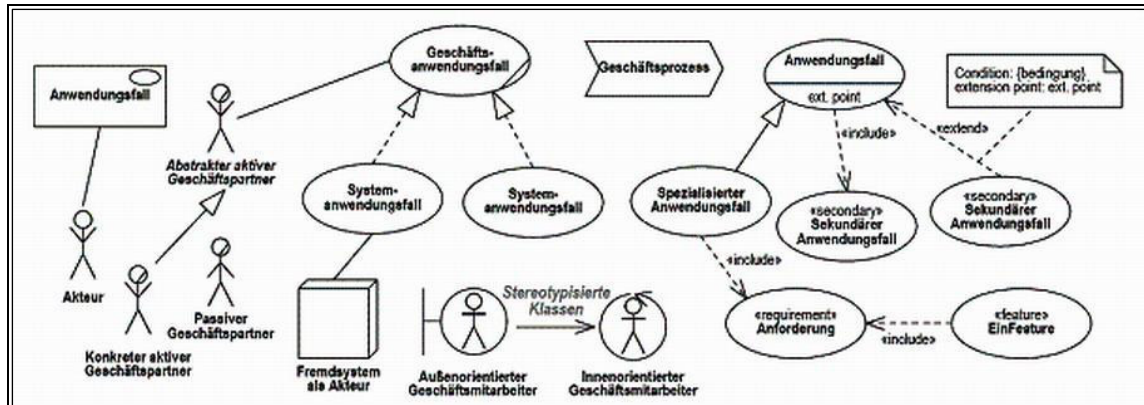
9.2.1 UML-Literatur

The screenshot shows a presentation slide with a navigation bar at the top containing the word 'Literatur', a home icon, and navigation arrows. The slide content is titled 'Literatur UML' in red. It contains a bulleted list of references:

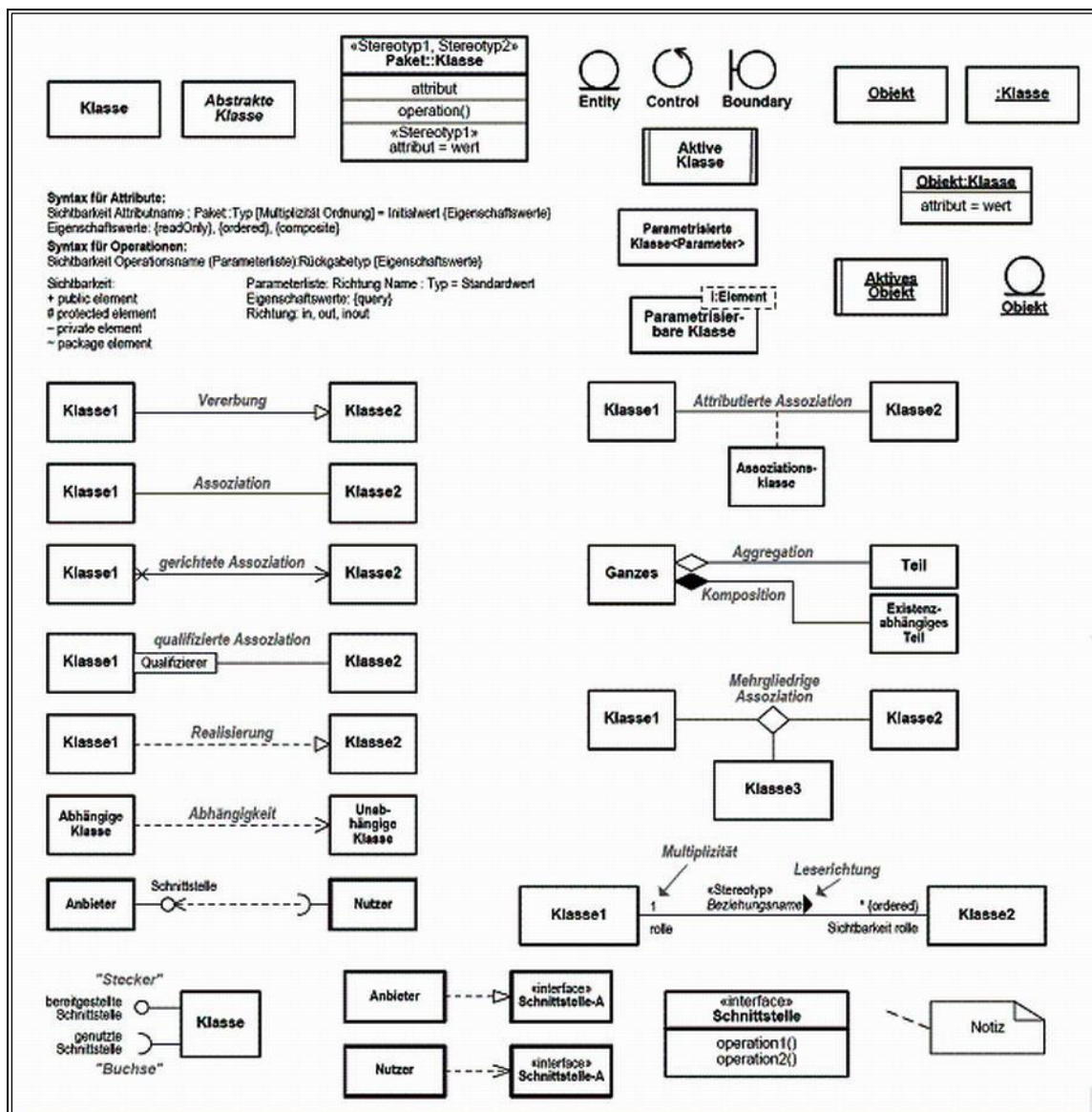
- UML ist ein Standard der OMG (<http://www.omg.org/uml>) und definiert eine Notation zur Visualisierung, Konstruktion und Dokumentation von Modellen für die objektorientierte Softwareentwicklung.
- Literatur:
 - [Booch 94] Grady Booch: Object oriented design with applications.
 - [Booch 96] Grady Booch: Object Solutions. Managing the Object-Oriented Projekt. Addison Wesley 1996
 - Booch, Rumbaugh, Jacobson: The Unified Modeling Language User Guide, Addison Wesley 1998
 - Booch, Rumbaugh, Jacobson: The Unified Modeling Language Reference Manual, Addison Wesley 1998
 - Booch, Rumbaugh, Jacobson: The Unified Software Development Process, Addison Wesley 1999
 - [Coad 93] Peter Coad / Jill Nicola: Object-oriented programming. Yourdon-Press 1993
 - [Gamma 92] Erich Gamma: Objektorientierte Software-Entwicklung am Beispiel von ET++. Springer 1992
 - [Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns: Elements of Object-Oriented Software Architecture. Addison Wesley 1994
 - Jacobson et.al.: Object-oriented Software engineering, Addison Wesley 1992
 - [Meyer 88] Bertrand Meyer: Object-oriented Software Construction. Prentice Hall 1988
 - James Rumbaugh et. al: Object-oriented modeling and design (OMT), Prentice Hall, 1994
 - [WWW 90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener: Objektorientiertes Software-Design. Prentice Hall 1990, in Deutsch beim Carl Hanser Verlag.

Abb. 9-2: UML Literatur

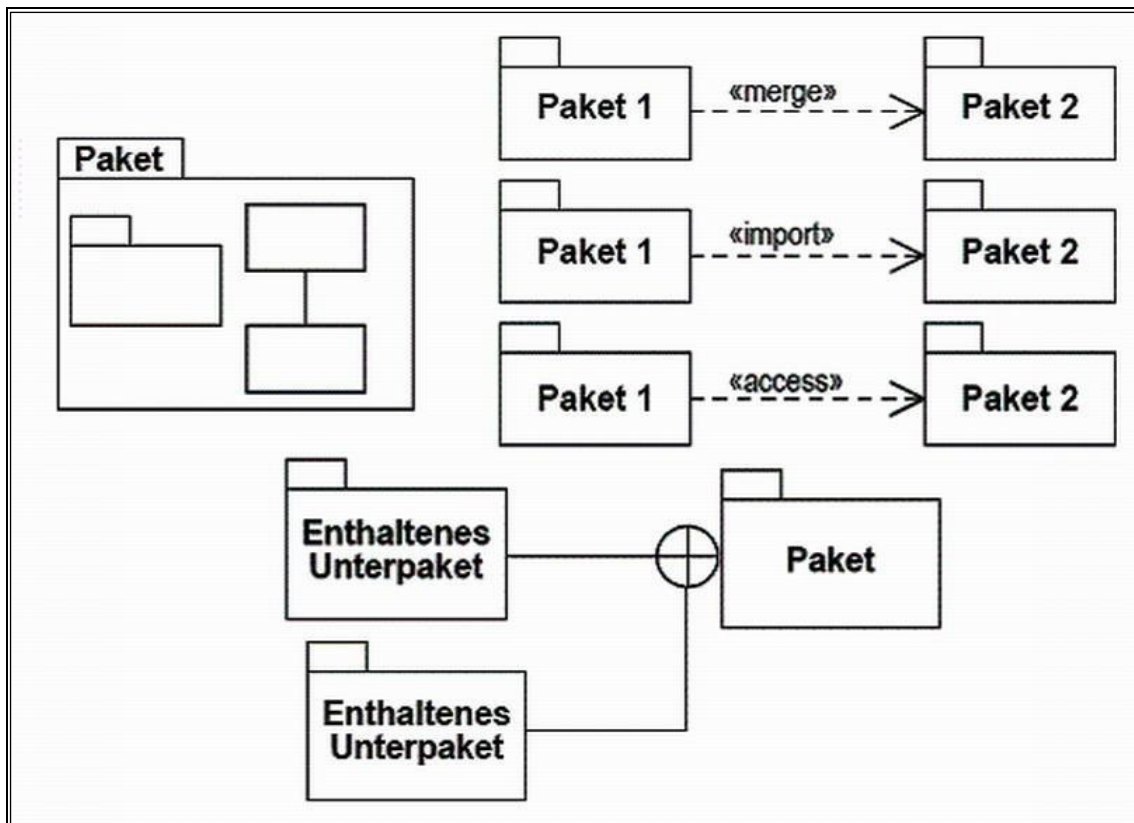
9.2.2 Anwendungsfalldiagramm



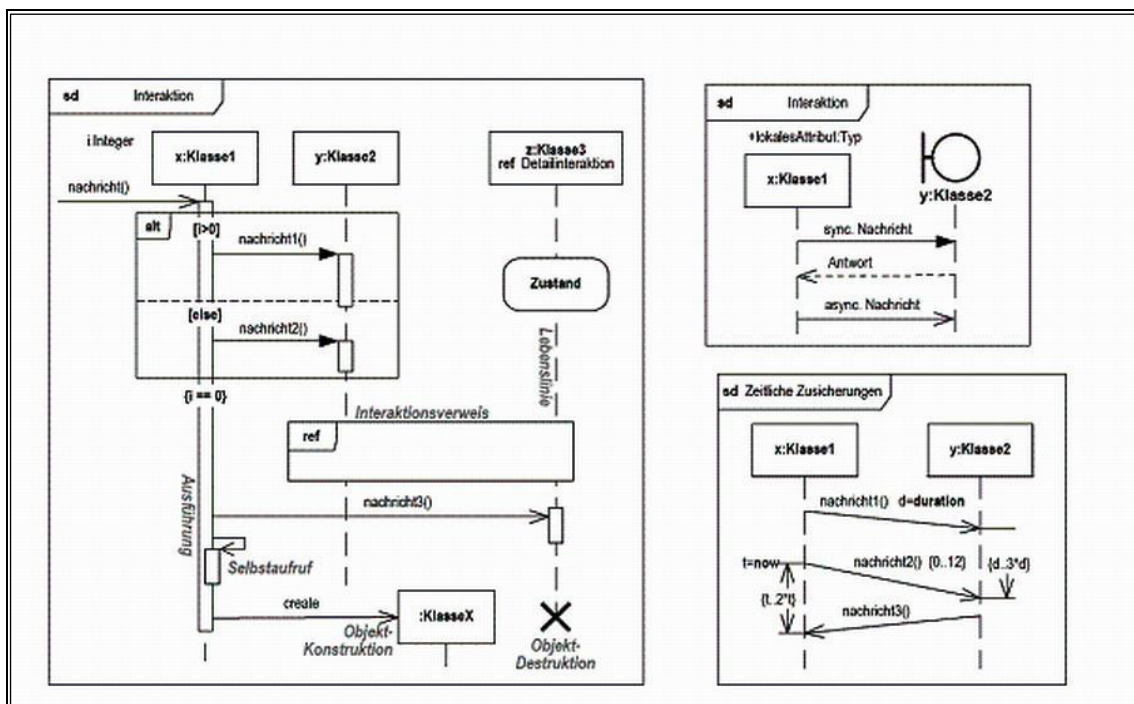
9.2.3 Klassendiagramm



9.2.4 Paketdiagramm



9.2.5 Sequenzdiagramm



10

Anhang

10.1	Java Schlüsselwörter.....	10-3
10.2	Namenskonventionen.....	10-3
10.3	Das Tool javadoc.....	10-4
10.4	Entwicklungstool Eclipse	10-6
	10.4.1 Download Eclipse	10-6
	10.4.2 Erzeugen eines ersten Java Projektes.....	10-6
10.5	Glossar	10-7

10 Anhang

10.1 Java Schlüsselwörter

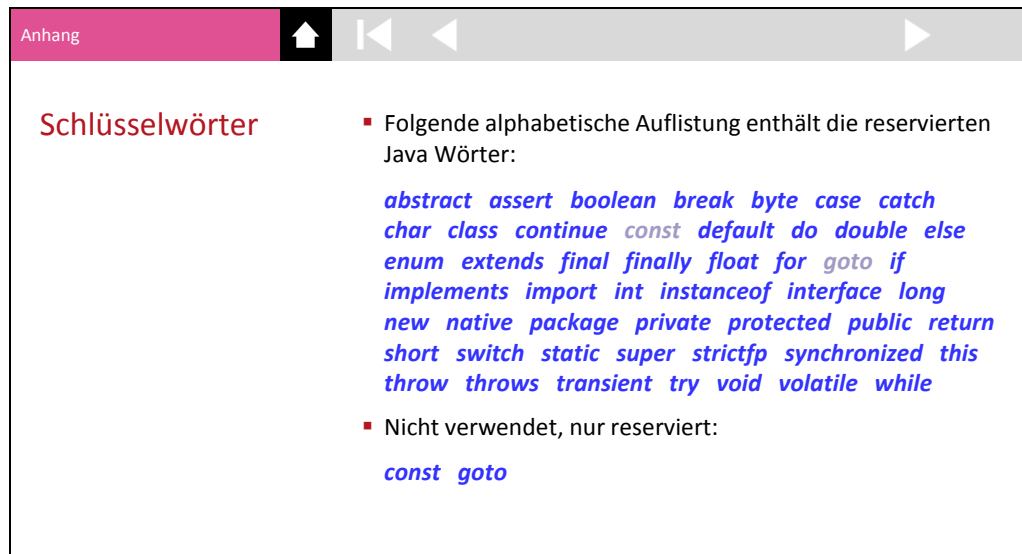


Abb. 10-1: Schlüsselwörter

10.2 Namenskonventionen

Folgende Namenskonventionen sollten bei der Programmierung mit Java eingehalten werden:

- Klassennamen beginnen mit einem Großbuchstaben,
- Paketnamen sind mit Kleinbuchstaben zu schreiben,
- Attribut- und Methodennamen beginnen mit einem Kleinbuchstaben,
- Methodennamen sind Verben.

Bei zusammengesetzten Wörtern werden zur besseren Lesbarkeit Großbuchstaben eingestreut.

Falls keine firmeneigenen Konventionen für die Codierung von Java vorhanden sind, empfiehlt es sich, die „Code Conventions for the Java Programming Language“ zu beachten. Sie sind im Web auf der Seite von Oracle verfügbar.

10.3 Das Tool javadoc

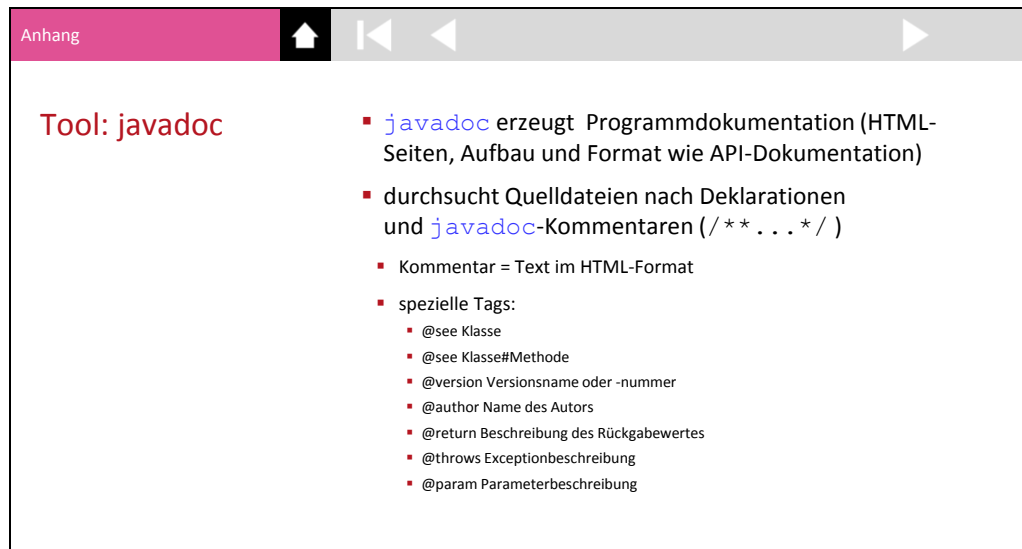


Abb. 10-2: javadoc

Eigene Dokumentationen können automatisch generiert werden durch das im JDK enthaltene Tool javadoc.

javadoc durchsucht Quelldateien nach Klassen, Methoden und Kommentaren.

javadoc-Kommentare werden mit dem speziellen Kommentar

`/** ... */`

eingegrenzt.

Ein Kommentar besteht aus beliebigem Text im HTML-Format.

Zusätzlich erkennt javadoc noch spezielle Tags:

`@see Klasse`

`@see Klasse#Methode`

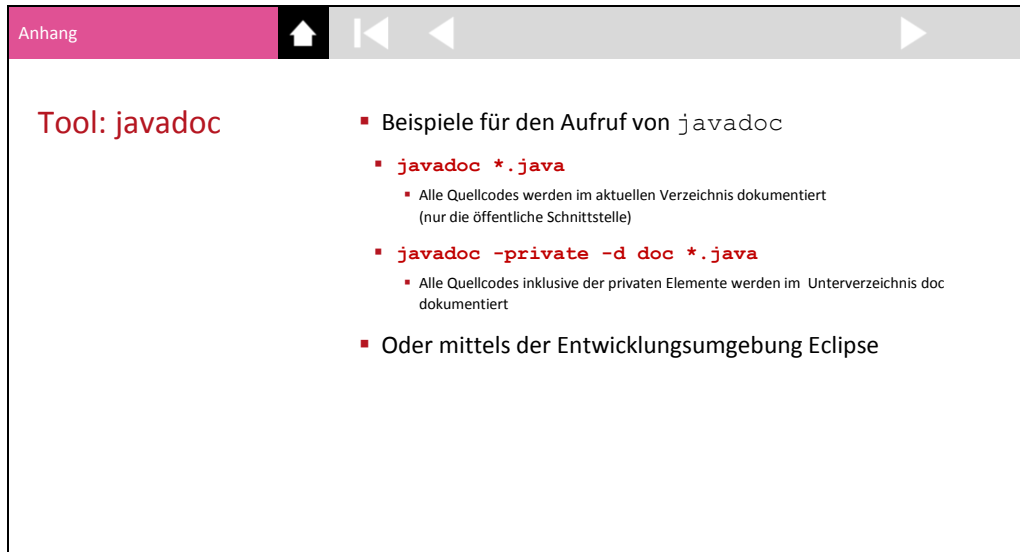
`@version Versionsname oder -nummer`

`@author Name des Autors`

`@return Beschreibung des Rückgabewertes`

`@throws Exceptionbeschreibung`

`@param Parameterbeschreibung`



The screenshot shows a presentation slide with a pink header bar labeled 'Anhang'. The slide content is titled 'Tool: javadoc' in red. It lists three examples for calling javadoc:

- Beispiele für den Aufruf von javadoc
 - `javadoc *.java`
 - Alle Quellcodes werden im aktuellen Verzeichnis dokumentiert (nur die öffentliche Schnittstelle)
 - `javadoc -private -d doc *.java`
 - Alle Quellcodes inklusive der privaten Elemente werden im Unterverzeichnis doc dokumentiert
 - Oder mittels der Entwicklungsumgebung Eclipse

Abb. 10-3: Beispiele javadoc

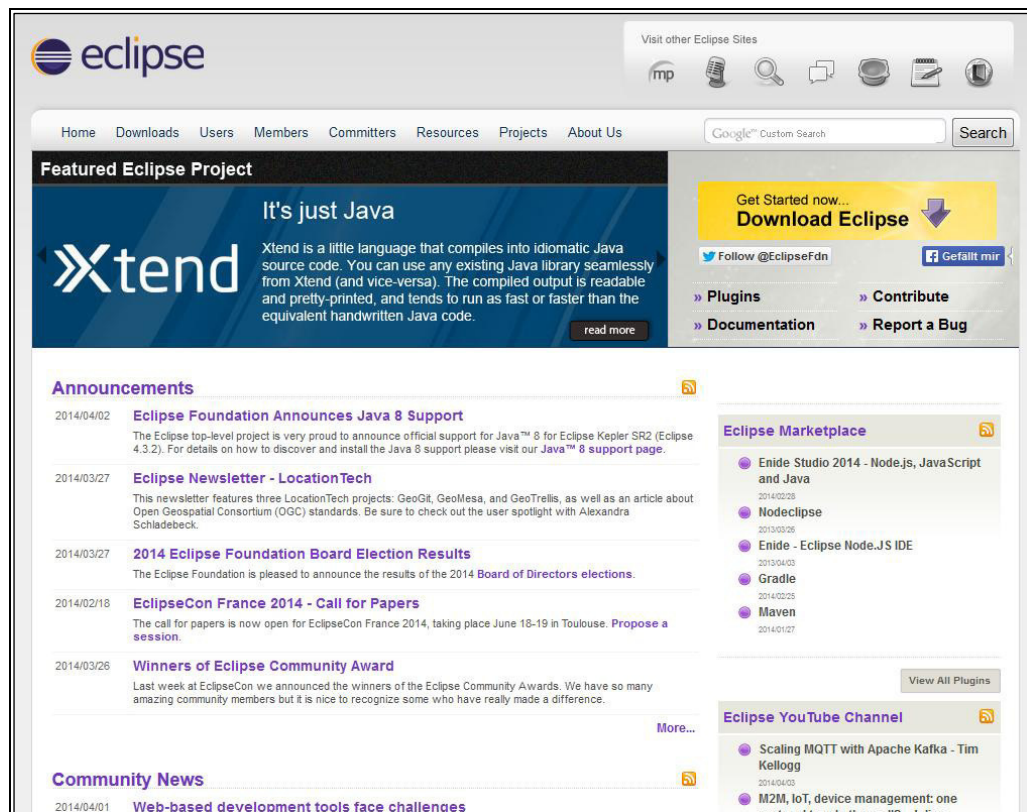
Auch ganze Pakete können kommentiert werden:

```
javadoc -private -sourcepath ..\src -d ..\doc  
de.integrata.java.grundlagen.sprache  
de.integrata.java.grundlagen.lang
```

10.4 Entwicklungstool Eclipse

10.4.1 Download Eclipse

Auf der Webseite: www.eclipse.org finden Sie das Tool Eclipse. Diese kostenlose Entwicklungsumgebung für Java bietet dem Programmierer viele Erleichterungen an. U.a. können Getter- u. Setter-Methoden generiert oder Konstruktoren auf Grundlage von Attributen erzeugt werden.



Eclipse bietet gleich auf dem Startbild eine Dokumentation und Beispiele an, so dass die Bedienung relativ leicht fällt.

10.4.2 Erzeugen eines ersten Java Projektes

Mit dem Projekt Wizard in der linken oberen Ecke des Programmfensters oder mit dem Menu File -> New Java Project öffnen sie den 'New Java Project' Dialog. Hier können sie den Projektnamen und ein bestehendes Projekt Directory (Workspace) auswählen.

10.5 Glossar

Anweisung

Eine Anweisung (engl. Statement) ist ein einzelner Befehl einer Programmiersprache, der dafür sorgt, dass etwas passiert. Anweisungen stehen für eine einzelne Aktion, die in einem Java-Programm ausgeführt wird. Jede Anweisung wird mit einem Strichpunkt (;) abgeschlossen.

API

Sogenannte APIs (Application Programming Interfaces) stellen innerhalb von Funktionsbibliotheken sämtliche zur Programmierung benötigten Routinen mehr oder weniger komfortabel zur Verfügung.

Applet

Ein Applet wird innerhalb eines WWW-Browsers ausgeführt, der Java unterstützt. Applets können zusätzlich mit dem Applet-Viewer, der im JDK enthalten ist, angezeigt werden. Um ein Applet auszuführen, muss es in eine Webseite eingefügt werden.

Applikation

Eine Applikation (Anwendung) ist ein Java-Programm, das eigenständig lauffähig ist. Applikationen werden mit dem Java-Interpreter ausgeführt, der die Haupt-.class-Datei der Applikation lädt. Dazu wird meist das java-Tool des JDK von der Kommandozeile aus aufgerufen.

Arrays

Arrays stellen eine Methode zur Speicherung einer Reihe von Elementen dar, die alle denselben primitiven Datentyp oder dieselbe Klasse aufweisen. Jedem Element wird innerhalb des Arrays ein eigener Speicherplatz zugewiesen. Diese Speicherplätze sind nummeriert, so dass Sie auf die Informationen leicht zugreifen können. Java implementiert Arrays anders als andere Programmiersprachen, nämlich als Objekte, die wie andere Objekte auch behandelt werden können.

Assoziativität

Die Assoziativität regelt die Abarbeitung von Operatoren gleicher Präzedenz. In Java gilt die Regel, dass alle zweistelligen Operatoren links-assoziativ sind. Dies bedeutet, dass eine Formel mit gleichen Präzedenzen immer von links nach rechts abgearbeitet wird. Einzige Ausnahme sind die Zuweisungsoperatoren, welche rechts-assoziativ sind.

Attribut

Attribute sind die einzelnen Dinge, die die einzelnen Klassen voneinander unterscheiden. Sie legen auch die Erscheinung, den Zustand und andere Qualitäten der Klasse fest. In einer Klasse werden Attribute über Variablen definiert.

Ausdruck

Ein Ausdruck (engl. Expression) ist der Teil einer Anweisung, die als Ergebnis einen Wert produziert. Dieser Wert kann zur späteren Verwendung im Programm gespeichert, direkt in einer anderen Anweisung verwendet oder überhaupt nicht beachtet werden.

AWT

Das Abstract Windowing Toolkit, auch AWT genannt, ist ein Satz von Klassen, der es Ihnen ermöglicht, eine grafische Benutzeroberfläche zu erstellen und Eingaben des Benutzers über die Maus und die Tastatur entgegenzunehmen. Da Java eine plattformunabhängige Sprache ist, haben die Benutzerschnittstellen, die mit dem AWT entworfen werden, auf allen Systemen die gleiche Funktionalität und abgesehen von den Plattformeigenheiten die gleiche Erscheinung.

Bedingung

Eine Bedingung ist eine Programmanweisung, die nur dann ausgeführt wird, wenn eine bestimmte Situation eintritt. Die elementarste Bedingung wird mit dem Schlüsselwort `if` erzeugt. Eine `if`-Bedingung verwendet einen booleschen Ausdruck, um zu entscheiden, ob eine Anweisung ausgeführt werden soll. Wenn der Ausdruck `true` zurückliefert, wird die Anweisung ausgeführt.

Blockanweisung

Eine Gruppe von Anweisungen, die sich in einem Paar geschweifter Klammern (`{ }`) befindet, wird als Block oder Blockanweisung bezeichnet.

boolesche Werte

Boolesche Werte sind spezielle Variablentypen, die nur die Werte `true` oder `false` aufnehmen können. Im Gegensatz zu anderen Sprachen haben boolesche Werte keine numerischen Werte, wobei 1 `true` und 0 `false` entspricht. Der Ausdruck `boolesch` geht auf George Boole, einen irischen Mathematiker zurück, der von 1815 bis 1864 lebte.

Casting

Casting ist ein Mechanismus, um einen neuen Wert zu erstellen, der einen anderen Typ aufweist als dessen Quelle. Casting ergibt ein neues Objekt oder einen neuen Wert. Casting wirkt sich nicht auf das ursprüngliche Objekt bzw. den ursprünglichen Wert aus.

Deadlock

Immer wenn zwei Threads eine Sperre auf zwei Objekte durchführen und dann anschließend versuchen, kreuzweise auf das andere, jeweils dann gesperrte Objekt zuzugreifen, besteht die Gefahr einer Verklemmung (eines Deadlocks). Beide Threads warten darauf, dass das jeweils andere Objekt seine Sperre verliert, was natürlich niemals geschieht. Ihr Programm bleibt stehen und arbeitet nicht mehr weiter.

Java hat keine Möglichkeiten, solche Deadlocks zu erkennen. Es ist Sache des Programmierers, Deadlocks zu vermeiden und dafür Sorge zu tragen, dass diese nicht auftreten.

dekrementieren

Eine Variable zu dekrementieren bedeutet, von deren Wert eins abziehen. Der Dekrement-Operator ist --. Dieser Operator wird direkt nach oder direkt vor einen Variablennamen platziert.

doppelte Pufferung

Die doppelte Pufferung ist ein Vorgang, bei dem alle Zeichenaktivitäten in einem Puffer abseits des Bildschirms vorgenommen werden. Anschließend wird der gesamte Inhalt dieses Puffers in einem Schritt am Bildschirm angezeigt. Diese Technik wird doppelte Pufferung genannt, weil es zwei Puffer für Grafikausgaben gibt, zwischen denen Sie wechseln.

Exception

Java stellt mit dem Sprachkonstrukt der Exception eine elegante Methode zur Verfügung, die Fehlerbehandlung in Programmen durchzuführen, ohne dass Klarheit und Einfachheit des Codes darunter leiden.

Expression

siehe Ausdruck

Felder

siehe Arrays

Finalizer

Finalizer-Methoden sind in gewissem Sinn das Gegenstück zu Konstruktor-Methoden. Während eine Konstruktor-Methode benutzt wird, um ein Objekt zu initialisieren, werden Finalizer-Methoden aufgerufen, kurz bevor das Objekt im Papierkorb landet und sein Speicher zurückgefordert wird. Finalizer-Methoden eignen sich am besten zur Optimierung des Entfernens von Objekten, z.B. durch Löschen aller Referenzen auf andere Objekte.

Garbage Collection

Garbage Collection ist ein Mechanismus, der nicht mehr benutzten, dynamisch angeforderten Speicherplatz auf dem Heap freigibt.

Gosling

James Gosling erfand 1991 die Programmiersprache Java bei Sun Microsystems.

Gültigkeitsbereich

Gültigkeitsbereich ist in der Programmierung der Begriff für den Teil eines Programms, in dem eine Variable existiert und verwendet werden kann. Wenn das Programm den Gültigkeitsbereich einer Variablen ver-

lässt, dann existiert diese nicht, und es treten Fehler auf bei dem Versuch, auf diese zuzugreifen. Der Gültigkeitsbereich einer Variablen ist der Block, in dem sie erzeugt wurde.

GUI

Das sogenannte GUI (Graphical User Interface) ist für alles verantwortlich, was der Benutzer von seinem Computer primär wahrnimmt: die grafische Benutzeroberfläche mit all ihren Fensterchen, Menüs, Icons, Dialogboxen und vielem anderen mehr.

Heap

Heap (oder Halde) ist ein Speicherbereich, aus dem dynamisch (zur Laufzeit des Programms) Speicherblöcke angefordert werden können.

Information Hiding

Objekte kapseln ihre Information (u.a. private Membervariablen), indem nur klasseneigene Funktionen auf diese Variablen zugreifen oder sie verändern können.

Siehe auch Kapselung.

Initialisierung

Datenvariable in Instanzen werden beim Erzeugen der Instanz mit Default-Werten versehen. Numerische Daten werden mit 0, char-Variable mit '\u0000', boolsche Größen mit false und Referenzvariable mit NULL vorbelegt. Eine Ausnahme bilden lokale MethodenvARIABLEN, die innerhalb eines Methodenrumpfes deklariert werden. Diese werden nicht initialisiert.

Es ist grundsätzlich anzuraten, Variablen eigenhändig zu initialisieren und sich nicht auf die automatische Initialisierung zu verlassen.

inkrementieren

Eine Variable zu inkrementieren bedeutet, zu deren Wert eins hinzuzuzählen. Der Inkrement-Operator ist ++. Dieser Operator wird direkt nach oder direkt vor einen Variablennamen platziert.

Inline-Code

Der Compiler erzeugt die Anweisungen bei Inline-Code direkt an der Stelle, an der die Funktion aufgerufen wird, d.h. es erfolgt kein Sprungbefehl. Das kostet i.d.R. mehr Speicher, bringt aber bessere Performance.

Instanz

Dasselbe wie ein Objekt. Jedes Objekt ist eine Instanz einer Klasse.

Instanzmethoden

Befindet sich das Schlüsselwort static nicht vor dem Namen einer Methode, so wird diese zur Instanzmethode. Instanzmethoden beziehen sich immer auf ein konkretes Objekt anstatt auf die Klasse selbst. Da

Instanzmethoden wesentlich häufiger verwendet werden als Klassenmethoden, werden Sie auch einfach nur als Methoden bezeichnet.

Instanzvariable

Eine Instanzvariable ist ein Stück Information, das ein Attribut eines Objekts definiert. Die Klasse des Objekts definiert die Art des Attributs, und jede Instanz speichert ihren eigenen Wert für dieses Attribut. Instanzvariablen werden auch als Objektvariablen bezeichnet.

Instanzvariablen gelten als solche, wenn sie außerhalb einer Methodendefinition deklariert werden. Üblicherweise werden die meisten Instanzvariablen direkt nach der ersten Zeile der Klassendefinition definiert.

Interface

siehe Schnittstelle

Introspection

siehe Reflection

iterativ

mehrere Male sich selbst wiederholend

JAR-Datei

Ein Java-Archiv, d.h. eine JAR-Datei, ist eine Sammlung von Java-Klassen oder anderen Dateien, die in eine einzige Archivdatei gepackt werden. JAR-Dateien eignen sich für die Verteilung und Installation von Java-Anwendungen und Klassenbibliotheken oder auch für einen schnelleren Download von Applets mit deren zugehörigen Dateien.

Java 2

In Verbindung mit dem JDK 1.2 wurde der Begriff "Java 2" eingeführt, das JDK 1.x wurde "Java 2 SDK 1.x" genannt, und die Editionen wurden J2SE, J2EE und J2ME genannt. Seit der Version 6 werden jedoch wieder die einfachen Bezeichnungen Java SDK, Java SE, Java EE und Java ME verwendet.

Kapselung

Wichtig für Objekte ist die Tatsache, dass diese Ihre Zustandsvariablen nach außen vor dem Benutzer verstecken. Ein Zugriff ist nur über die eigenen Objektmethoden zulässig. Damit ist das Objekt in der Lage, seine tatsächliche Realisierung vor dem Benutzer zu verbergen und kann diese damit jederzeit abändern. Solange sich die Methodenschnittstelle nicht ändert, hat dies keine Konsequenzen.

Klasse

Eine Klasse ist eine Vorlage, die zur Erzeugung vieler Objekte mit ähnlichen Eigenschaften verwendet wird. Sie beinhaltet Variablen, um das Objekt zu beschreiben, und Methoden, um zu beschreiben, wie sich

das Objekt verhält. Klassen können Variablen und Methoden von anderen Klassen erben.

Klassenbibliothek

Eine Klassenbibliothek ist eine Gruppe von Klassen, die zur Verwendung mit anderen Programmen entworfen wurden. Die Standard-Java-Klassenbibliothek beinhaltet Dutzende von Klassen.

Klassenmethoden

Eine Methode, die auf eine Klasse selbst angewendet wird und nicht auf eine bestimmte Instanz einer Klasse.

Klassenvariable

Eine Klassenvariable ist ein Stück Information, das ein Attribut einer Klasse definiert. Die Variable bezieht sich auf die Klasse selbst und all ihre Instanzen, so dass nur ein Wert gespeichert wird unabhängig davon, wie viele Objekte dieser Klasse erzeugt wurden. Sie definieren Klassenvariablen, indem Sie das Schlüsselwort `static` vor die Variable setzen. Eine Klassenvariable existiert nur einmal pro Klasse und nicht einmal pro Instanz.

Kommentare

Eine der wichtigsten Methoden, die Lesbarkeit eines Programms zu verbessern, sind Kommentare. Kommentare sind Informationen, die in einem Programm einzig für den Nutzen eines menschlichen Betrachters eingefügt wurden, der versucht, herauszufinden, was das Programm tut. Der Java-Compiler ignoriert die Kommentare komplett, wenn er eine ausführbare Version der Java-Quelldatei erstellt.

Die erste Methode, einen Kommentar in einem Programm einzufügen, ist, dem Kommentartext zwei Schrägstriche (`//`) voranzustellen. Dadurch wird alles nach den Schrägstrichen bis zum Ende der Zeile zu einem Kommentar. Wenn Sie einen Kommentar einfügen wollen, der länger als eine Zeile ist, dann starten Sie den Kommentar mit der Zeichenfolge `/*` und beenden ihn mit `*/`. Alles zwischen diesen beiden Begrenzern wird als Kommentar gesehen.

Komponententechnologie

Hierbei handelt es sich um das Konzept der modularen Programmierung: Vorgefertigte Komponenten werden zu einer Anwendung zusammengesetzt, das Rad muss so nicht für jedes Programm neu erfunden werden. In Java steht hierfür das Konzept der Java-Beans.

Konstanten

Neben Variablen werden in Java-Programmen auch Konstanten (Literals) verwendet. Literale sind Zahlen, Text oder andere Informationen, die direkt einen Wert darstellen und keinen Speicherplatz benötigen.

Konstruktor

Ein Konstruktor ist eine spezielle Methode zum Erstellen und Initialisieren neuer Instanzen von Klassen. Konstruktoren initialisieren das neue Objekt und seine Variablen, erzeugen andere Objekte, die dieses Objekt braucht, und führen im Allgemeinen andere Operationen aus, die für die Ausführung des Objekts nötig sind. Konstruktoren werden im Zusammenhang mit dem Schlüsselwort `new` aufgerufen.

Literale

Siehe Konstanten.

Locking

Sperren von Objekten, so dass nur ein Prozess, statt mehrerer gleichzeitig, auf bestimmte Daten zugreifen kann; wichtig für Datenkonsistenz.

Methoden

Methoden sind Funktionen, die zu einer Klassen gehören. Methoden sind Gruppen von miteinander in Beziehung stehenden Anweisungen in einer Klasse. Diese Anweisungen beziehen sich auf die eigene Klasse und auf andere Klassen und Objekte. Sie werden verwendet, um bestimmte Aufgaben zu erledigen. Objekte kommunizieren miteinander über Methoden. Der Aufruf von Methoden besteht aus dem Namen der Methode und Klammern. Die Klammern dürfen auf keinen Fall weggelassen werden. Die Klammern können leer bleiben oder einen oder mehrere Parameter enthalten.

new

Um ein neues Objekt einer Klasse zu erstellen, wird der Operator `new` und ein Konstruktor verwendet.

Objekt

Objekte werden auch als Instanzen einer Klasse bezeichnet. Ein Objekt ist ein abgeschlossenes Element eines Computerprogramms, das eine Gruppe miteinander verwandter Features darstellt und dafür ausgelegt ist, bestimmte Aufgaben zu erfüllen. Mehrere Objekte, die Instanzen derselben Klasse sind, haben Zugriff auf dieselben Methoden, aber oft unterschiedliche Werte für deren Instanzvariablen.

objektorientiert

Als Objektorientierte Programmierung (OOP) wird eine Vorgangsweise bezeichnet, bei der Computerprogramme als eine Reihe von Objekten aufgebaut werden, die miteinander interagieren. Im Wesentlichen ist es eine bestimmte Methode, Computerprogramme zu organisieren.

Objektvariable

siehe Instanzvariable

Operatoren

Operatoren sind spezielle Symbole, die für mathematische Funktionen, bestimmte Zuweisungsarten und logische Vergleiche stehen.

Operatorpräzedenz

Unter Operatorpräzedenz versteht man die Priorität der einzelnen Operatoren untereinander. Operatoren mit hoher Präzedenz werden vor solchen Operatoren mit niedriger Präzedenz abgearbeitet, d.h. dass bei einem klammerfreien Ausdruck sich die Abarbeitungsreihenfolge nach dieser Präzedenz richtet.

siehe auch Assoziativität

overloading

siehe Überladen

Pakete

Pakete (engl. Packages) sind eine Möglichkeit, um verwandte Klassen und Schnittstellen zu gruppieren. Pakete ermöglichen es, dass Gruppen von Klassen nur bei Bedarf verfügbar sind. Klassen, die sich nicht in dem Paket `java.lang` befinden, müssen explizit importiert oder direkt über den vollen Paket- und Klassennamen angesprochen werden.

parallel

In den meisten Programmiersprachen werden die Programmteile nacheinander, also sequentiell abgearbeitet. Java erlaubt nun mit Hilfe von Threads die Realisierung von Pogrammeinheiten, die parallel bzw. quasiparallel ablaufen.

siehe auch Threads

Pointer

Pointer sind Variablen, die als Inhalt Hauptspeicheradressen haben. Pointer sind in Java nicht direkt definierbar. Adressen können in Java nur indirekt über Referenzen verwendet werden.

Postfix-Operatoren

Inkrement- und Dekrement-Operatoren werden als Postfix-Operatoren bezeichnet, wenn sie sich hinter dem Variablennamen befinden.

Präfix-Operatoren

Inkrement- und Dekrement-Operatoren werden als Präfix-Operatoren bezeichnet, wenn sie vor dem Namen der Variablen aufgeführt werden.

Punktnotation

Die Punktnotation ist die Form, mit der auf Instanzvariablen und Methoden innerhalb eines Objekts mit dem Punkt-Operator zugegriffen wird.

Referenz

Eine Referenz ist eine Art Zeiger (Pointer), der auf ein Objekt verweist. Wenn in Java mit Objekten, also Instanzen von Klassen, gearbeitet wird, werden nur die Referenzen verwendet, nicht die Objekte selbst.

Wenn eine Methode mit Objektparametern aufgerufen wird, ist der Aufruf ein Call by Reference. Das bedeutet, dass sich alles, was Sie mit diesen Objekten in der Methode anstellen, gleichermaßen auf die Originalobjekte auswirkt.

Reflection

Reflection bzw. Introspection ermöglicht es einer Java-Klasse - beispielsweise einem von Ihnen geschriebenen Programm - Details über eine beliebige andere Klasse zu ermitteln. Mit Reflection kann ein Java-Programm zur Laufzeit eine Klasse laden, von der es zur Übersetzungszeit noch nichts weiß, die Variablen, Methoden und Konstruktoren dieser Klasse ermitteln und damit arbeiten.

Round-Robbin-Strategie

Hierbei handelt es sich um ein Prinzip der Zuteilung von Prozessorzeit an die Threads. First come, first server oder Ringelreihenprinzip oder FIFO (first in - first out) genannt. Ausnahme: Threads mit höherer Priorität verdrängen die mit niederer.

Scheduler

Der Java-Scheduler ist der Teil des Java-Interpreters, der bestimmt, welcher Prozess wann Prozessorzeit zugewiesen bekommt.

Schleife

Eine Schleife wiederholt eine Anweisung oder einen Anweisungsblock mehrmals, solange eine Bedingung zutrifft.

Schnittstelle

Eine Schnittstelle (engl. Interface) ist eine Sammlung von Methoden, die benannt aber nicht unbedingt implementiert (default-Implementierung) sind. Dadurch wird angezeigt, dass eine Klasse neben dem aus der Superklasse geerbten Verhalten noch zusätzliche Verhaltensweisen hat.

Semantik

Die Semantik oder Wortbedeutungslehre beeinflusst, wie Programme konzipiert, von anderen verstanden und vom Rechner ausgeführt werden: wie verhält sich ein Programm, wenn es auf einem Rechner ausgeführt wird.

Signatur

Die Signatur von Methoden besteht aus den Teilen: (1) Name der Methode, (2) Objekttyp oder der primitive Typ, den die Methode zurückgibt und (3) die Liste der Parameter.

Statement

siehe Anweisung

statisch

Die Bezeichnung statisch (über das Schlüsselwort `static`) für eine Variable bezieht sich auf eine Bedeutung des Wortes: ortsfest. Wenn eine Klasse eine statische Variable besitzt, dann existiert diese Variable nur ein Mal pro Klasse.

Strings

siehe Zeichenketten

Subklasse

Eine Klasse, die von einer andere Klasse abgeleitet wurde.

Superklasse

Eine Klasse, von der andere Klassen abgeleitet werden, wird in Java Superklasse genannt. Gleichbedeutend mit Basis- oder Oberklasse. Die Klasse, die die Funktionalität erbt, wird als Subklasse bezeichnet. Eine Klasse kann lediglich eine Superklasse besitzen. Jede Klasse kann allerdings eine uneingeschränkte Anzahl von Subklassen haben. Subklassen erben alle Attribute und sämtliche Verhaltensweisen ihrer Superklasse.

Syntax

Die Syntax legt fest, wie Programme geschrieben werden müssen, damit sie vom Rechner korrekt erkannt werden: wie werden Ausdrücke, Befehle, Vereinbarungen usw. zusammensetzt, um ein korrektes Programm zu bilden.

this

Um auf das aktuelle Objekt einer Klasse Bezug zu nehmen, können Sie das Schlüsselwort `this` verwenden. Da `this` eine Referenz auf die aktuelle Instanz einer Klasse ist, ist es sinnvoll, das Schlüsselwort nur innerhalb der Definition einer Instanzmethode zu verwenden. Klassenmethoden, d.h. Methoden, die mit dem Schlüsselwort `static` deklariert sind, können `this` nicht verwenden.

Thread

Ein Thread ist ein Teil eines Programms, der eingerichtet wird, um eigenständig zu laufen, während der Rest des Programms etwas anderes tut. Dies wird auch als Multitasking bezeichnet, da das Programm mehr als eine Aufgabe zur selben Zeit ausführen kann. Threads sind ideal für alles, was viel Rechenzeit in Anspruch nimmt und kontinuierlich ausgeführt wird, wie z.B. die wiederholten Zeichenoperationen, die eine Animation ausmachen. Indem Sie die Arbeitslast der Animation in einen Thread packen, machen Sie den Weg dafür frei, dass sich der Rest des Programms mit anderen Dingen beschäftigen kann.

Überladen

Methoden mit demselben Namen werden durch zwei Dinge voneinander unterschieden: 1. Die Anzahl der Argumente, die ihnen übergeben wird und 2. den Datentyp oder Objekttyp der einzelnen Argumente. Diese beiden Dinge bilden die Signatur einer Methode. Mehrere Methoden zu verwenden, die alle denselben Namen, aber unterschiedliche Signaturen haben, wird als Überladen (engl. overloading) bezeichnet.

Überschreiben

Zuweilen soll ein Objekt auf die gleichen Methoden reagieren, jedoch beim Aufrufen der jeweiligen Methode ein anderes Verhalten aufweisen. In diesem Fall können Sie die Methode überschreiben. Durch Überschreiben von Methoden definieren Sie eine Methode in einer Subklasse, die die gleiche Signatur hat wie eine Methode in einer Superklasse. Dann wird zum Zeitpunkt des Aufrufs nicht die Methode in der Superklasse, sondern die in der Subklasse ermittelt und ausgeführt.

Normalerweise gibt es zwei Gründe dafür, warum man eine Methode, die in einer Superklasse bereits implementiert ist, überschreiben will: Um die Definition der Originalmethode völlig zu ersetzen oder um die Originalmethode zu erweitern.

UML - Unified Modeling Language

UML ist eine [grafische Modellierungssprache](#) zur Spezifikation, Konstruktion und Dokumentation von Software-Teilen und anderen Systemen. Sie wird von der [Object Management Group](#) (OMG) entwickelt und ist von der ISO (ISO/IEC 19505 für Version 2.1.2) standardisiert.

Unicode-Zeichensatz

Java unterstützt auch den Unicode-Zeichensatz, der den Standardzeichensatz plus Tausende anderer Zeichen beinhaltet, um internationale Alphabete zu repräsentieren. Zeichen mit Akzenten und andere Symbole können in Variablennamen verwendet werden, solange diese über eine Unicode-Nummer oberhalb von 00C0 verfügen.

siehe auch Alan Wood's Unicode Resources

oder Unicode Organisation (www.unicode.org)

Variablen

Variablen sind Orte, an denen, während ein Programm läuft, Informationen gespeichert werden können. Der Wert der Variablen kann unter deren Namen von jedem Punkt im Programm aus geändert werden. Um eine Variable zu erstellen, müssen Sie dieser einen Namen geben und festlegen, welchen Typ von Informationen sie speichern soll. Sie können einer Variablen auch gleich bei der Erzeugung einen Wert zuweisen.

Siehe auch Instanzvariable und Klassenvariable.

Vererbung

Vererbung ist ein Mechanismus, der es einer Klasse ermöglicht, all Ihre Verhaltensweisen und Attribute von einer anderen Klasse zu erben. Über die Vererbung verfügt eine Klasse sofort über die gesamte Funktionalität einer vorhandenen Oberklasse. (siehe auch Subklasse und Superklasse)

Zeichenketten

Zeichenketten sind wie Arrays Objekte und damit ein Referenzdatentyp. Diese Objekte erlauben den effizienten Umgang mit Zeichenketten. Zu beachten ist, dass es zwei unterschiedliche Arten von Zeichenketten in Java gibt. Zum einen stellt Java Objekte der Klasse String (Zeichenketten mit fester Länge) und zum anderen Objekte der Klasse StringBuffer (Zeichenketten mit variabler Länge) während der Laufzeit des Programms zur Verfügung.

Gesamtindex

A

abgeleitete Klasse 5-6
abstract 6-4
Abstrakte Elemente 6-3
Abstrakte Klassen 6-3
abstrakte Methoden 6-3
Aggregation 5-4
Anonyme Objektreferenz 4-22
Anweisung 3-11
Arithmetische Operatoren 3-4
Array 3-22
Arrays 3-23, 3-24
Assoziation 5-3
Attribute 4-7, 4-9, 4-10, 4-11, 4-14, 4-20,
4-24, 4-33, 5-7, 8-12
Ausdruck 3-11
ausführbare Klasse 4-17
Ausgabe 1-17, 1-19, 8-9, 8-10
Ausnahmen 1-11, 4-36, 7-7
Autoboxing 7-8

B

Basisklasse 5-7, 5-9, 5-11
Bedingung 5-13
binäre Operatoren 3-3
boolean 7-7, 8-4, 8-5, 8-12
Botschaften 4-7, 4-8, 4-9
Browser 1-3
Bytecode 1-6, 1-9, 1-12, 1-13, 1-14, 1-15,
1-16, 1-18, 1-19, 4-33, 6-11, 8-3

C

Case-Struktur 3-13
cast 5-14
Cast 3-7, 5-13, 5-14
catch 4-36, 4-37
char 7-7, 8-4, 8-5
ClassLoader 1-15, 1-16, 1-17
CLASSPATH 1-16, 6-11
clone 8-11, 8-12
Code Conventions 10-3

Compiler 1-3, 1-6, 1-7, 1-9, 1-10, 1-18,
1-19, 6-11

Connection 9-4

Container 3-22, 8-3

D

Datentypen 1-14, 2-7, 7-7
default-Konstruktor 5-9
Default-Konstruktor 4-24
Definition von Variablen 2-9
Dekrement 3-5
Design Pattern 7-3
Dezimalform 2-11
double 7-7, 8-5, 8-6

E

else 3-11, 8-9, 8-10
Entwurfsmuster 7-3, 7-7
Enumeration 7-9, 7-10
equals 8-4, 8-5, 8-9, 8-10
Exception 4-36, 4-39, 5-14, 5-15, 5-16,
8-11
Exponent 2-12
extends 5-8, 6-8

F

false 8-6, 8-10
final 5-16, 5-17
Finale Elemente 5-17
Finale Variablen 2-10
finally 4-36, 4-37
flache Kopie 8-12
float 8-5
for 1-14
Funktion 3-21
Funktionen 4-15

G

Garbage Collection 1-10, 1-14

H

Hexadezimale Form 2-12

I

if 3-11
implements 6-7, 6-8
import 6-13
Inkrement 3-5
instanceof 5-13, 8-10
Instanz 4-19
Instanziierung 4-24, 4-26, 4-29, 5-16
interface 6-5, 6-8
Interfaces 6-5

J

Java 2 Platform 1-6
javadoc 2-4, 10-4
JDK 1-3, 1-6, 1-19, 2-4, 6-11, 8-16, 8-17,
10-4

K

Kapselung 4-9, 4-10, 5-9, 6-12
Klasse 1-10, 1-15, 1-16, 1-17, 1-18, 1-19,
2-3, 4-3, 4-9, 4-10, 4-11, 4-13, 4-33, 5-6,
5-7, 5-8, 5-12, 5-14, 5-15, 5-16, 6-11,
6-12, 6-13, 7-7, 8-4, 8-9, 8-11, 8-12, 8-16
Klassen 4-9
Klassenattribute 4-30
Klassenmethoden 4-30, 4-31, 4-33, 7-8
Kommentare 2-4, 10-4
komplexe Datentypen 3-22
Komponenten 1-14
Komposition 5-4
Konstanten 1-13, 2-11, 8-3
Konstruktoren 4-24, 5-9, 5-15
Konstruktorenüberladung 4-27

L

Laufzeitumgebung 1-7, 1-11, 1-15, 1-19,
1-20, 5-14, 8-3
long 8-5

M

main 4-17
Mantisse 2-12
Mehrfachvererbung 6-5, 6-8

Methoden 1-9, 4-7, 4-8, 4-9, 4-10, 4-11,
4-15, 4-20, 4-33, 5-7, 5-11, 5-15, 7-7,
8-5, 8-9, 10-4

Minus-Operator, unär 3-4

N

Namenskonventionen 10-3
new 4-19, 4-24

O

Object 8-5, 8-9, 8-10, 8-11, 8-12
Objekt 4-7, 4-8, 4-9, 4-19, 4-20, 7-8, 8-4
öffentlich 4-11, 6-16
Oktalform 2-12
Online-Dokumentation 8-17
Operatoren 3-3, 3-4

P

package 6-11
Paket 5-15, 6-11, 6-12, 6-13, 8-11
Plus-Operator, unär 3-4
Polymorphie 4-16, 5-12
Polymorphismus 5-12
println() 4-18
private 4-11, 5-8
Private Konstruktoren 4-29
protected 5-9, 6-16, 8-11
public 6-16, 8-12
Punktoperator 4-20

Q

Quellcode 1-3, 1-6, 1-12, 1-18, 1-19, 2-3,
4-16, 6-11, 6-12

R

Referenz 1-7, 3-22, 4-19, 4-20, 4-28, 8-10
Reinitialisierung 3-20
return 3-21, 4-22, 8-10, 8-12

S

Schnittstellen 6-5
short 7-7
Sicherheit 1-15, 4-11
Sichtbarkeit in der Vererbungshierarchie
5-9
Signatur einer Methode 4-15
Singleton 7-3, 7-7
Standard-Konstruktor 4-24

static 4-30, 4-31, 4-33
static-Block 4-33
statischer Initialisierungsblock 4-32
String 4-22, 7-8, 8-4, 8-5, 8-6, 8-11
Subklasse 5-6, 5-9, 5-11, 5-16, 8-11
switch 3-13

T

ternäre Operatoren 3-3
this 4-20, 4-28, 4-29, 8-12
Thread 1-11
throw 4-36
throws 4-36, 5-16, 8-12
tiefe Kopie 8-12
toString 8-11
true 8-4, 8-6
try 4-36, 4-37
Typumwandlung 3-7, 5-13

U

Überladen von Methoden 4-16
UML-Notation 4-10
unäre Operatoren 3-3
Unboxing 7-8

Unterklasse 5-6

V

Variable Argumentlisten 7-11
Variablen 2-9
Vererbung 5-6, 5-9, 5-16
Vergleich 8-4, 8-6, 8-9, 8-10
Version 1-16
Virtuelle Maschine 1-11, 1-12, 1-13, 1-14,
1-16, 1-17, 4-33, 8-3
VM 1-11, 1-14, 4-33
void 4-15, 4-17, 4-24, 4-29

W

Wartbarkeit 4-11
Wiederverwendbarkeit 4-11
Wrapper 7-7, 7-8
Wrapper-Klassen 7-7

Z

Zeichenketten 8-4
Zugriffsmethoden 4-11, 4-31, 5-8
Zugriffsrechte 4-12, 6-16
Zuweisung 3-4

