



Guido Rau
guido.rau@dataserv-training.de

Diese PDF ist für Sie persönlich codiert. Sie steht Ihnen für Lern- und Ausbildungszwecke zur Verfügung. Jeglicher Inhalt, einschließlich der Layouts und Anordnungen, unterliegt dem Schutz des Urheberrechts sowie weiterer Schutzrechte.

Ohne schriftliche Genehmigung des HERDT-Verlags für Bildungsmedien sind Reproduktion und Weitergabe der PDF – auch in Teilen – ausdrücklich verboten und ziehen zivil- und strafrechtliche Konsequenzen nach sich.

Elmar Fuchs

1. Ausgabe, 1. Aktualisierung, Mai 2021

ISBN 978-3-86249-645-7

Java 9

Grundlagen Programmierung

JAV9



1 Informationen zu diesem Buch	4	6 Klassen, Attribute, Methoden	62
1.1 Voraussetzungen und Ziele	4	6.1 Klassen	62
1.2 Aufbau und Konventionen	5	6.2 Die Attribute einer Klasse	64
2 Einführung in Java	7	6.3 Objekte erzeugen	65
2.1 Die Programmiersprache Java	7	6.4 Methoden – die Funktionalität der Klassen	69
2.2 Das Java Development Kit (JDK)	9	6.5 Methoden mit Parametern erstellen	71
3 Ein Programm mit Java erstellen	11	6.6 Methoden mit Rückgabewert definieren	73
3.1 Ablauf bei der Programmerstellung	11	6.7 Methoden überladen	74
3.2 Aufbau einer Anwendung	14	6.8 Statische Variablen und Methoden	75
3.3 Ein Java-Programm mit dem Java-Compiler javac kompilieren	15	6.9 Übung	78
3.4 Ein Java-Programm mit dem Interpreter java ausführen	17	7 Kapselung und Konstruktoren	79
3.5 Problembehandlung	18	7.1 Kapselung	79
3.6 Übung	20	7.2 Zugriffsmethoden	80
4 Grundlegende Sprachelemente	21	7.3 Konstruktoren	83
4.1 Was ist die Syntax?	21	7.4 Übung	88
4.2 Bezeichner und reservierte Wörter	21	8 Vererbung	89
4.3 Quelltext dokumentieren	23	8.1 Grundlagen zur Vererbung	89
4.4 Anweisungen in Java erstellen	25	8.2 Klassen ableiten und erweitern	90
4.5 Primitive Datentypen	25	8.3 Konstruktoren aufrufen	92
4.6 Literale für primitive Datentypen	27	8.4 Geerbte Methoden überschreiben	94
4.7 Mit lokalen Variablen arbeiten	28	8.5 Vererbungsketten und Zuweisungskompatibilität	98
4.8 Werte zuweisen	30	8.6 Polymorphie in der Vererbung	101
4.9 Typkompatibilität und Typkonversion	31	8.7 Die Superklasse <code>Object</code>	102
4.10 Konstanten – unveränderliche Variablen	33	8.8 Finale Klassen	105
4.11 Arithmetische Operatoren und Vorzeichenoperatoren	34	8.9 Abstrakte Klassen und abstrakte Methoden	106
4.12 Vergleichsoperatoren und logische Operatoren	38	8.10 Übung	109
4.13 Daten aus- und eingeben	40	9 Packages und Module	110
4.14 Übung	44	9.1 Klassen in Packages organisieren	110
5 Kontrollstrukturen	45	9.2 Zugriffsrechte in Packages	113
5.1 Kontrollstrukturen einsetzen	45	9.3 Packages einbinden	114
5.2 <code>if</code> -Anweisung	46	9.4 Statisches Importieren	117
5.3 <code>if-else</code> -Anweisung	48	9.5 Mit dem JDK mitgelieferte Packages	119
5.4 <code>switch</code> -Anweisung	50	9.6 Module	121
5.5 Schleifen	52	9.7 Anwendung des Modulsystems	125
5.6 <code>while</code> -Anweisung	53	9.8 Übung	128
5.7 <code>do-while</code> -Anweisung	55	10 Interfaces und Adapterklassen	129
5.8 <code>for</code> -Anweisung	56	10.1 Interfaces	129
5.9 Weitere Anweisungen in Kontrollstrukturen	59	10.2 Adapterklassen	134
5.10 Java-Kontrollstrukturen im Überblick	60	10.3 Direkte Methodenimplementierung im Interface	136
5.11 Übung	61	10.4 Übung	142

11 Mit Strings und Wrapper-Klassen arbeiten	143	14 Ausnahmebehandlung mit Exceptions	182
11.1 Die Klasse <code>String</code>	143	14.1 Auf Laufzeitfehler reagieren	182
11.2 Strings verketteten und vergleichen	145	14.2 Exceptions abfangen und behandeln	185
11.3 Weitere Methoden der Klasse <code>String</code>	147	14.3 Exceptions weitergeben	190
11.4 Die Klassen <code>StringBuffer</code> und <code>StringBuilder</code>	149	14.4 Abschlussarbeiten in einem <code>finally</code> -Block ausführen	193
11.5 Wrapper-Klassen	150	14.5 Exceptions auslösen	194
11.6 Übung	153	14.6 Eigene Exceptions erzeugen	195
		14.7 Übung	198
12 Arrays und Enums	154	15 Nützliche Klassen und Packages	199
12.1 Arrays	154	15.1 Zufallszahlen	199
12.2 Mit Arrays arbeiten	157	15.2 Grundlagen zu Datum und Zeit	201
12.3 Mehrdimensionale Arrays	160	15.3 Zeitpunkte – Klassen <code>Instant</code> und <code>Duration</code>	202
12.4 Spezielle Methoden zur Arbeit mit Arrays	162	15.4 Datumsangaben – Klassen <code>LocalDate</code> , <code>ZonedDateTime</code> und <code>Period</code>	204
12.5 Parameterübergabe an die <code>main</code> -Methode	163	15.5 Zeiten – die Klasse <code>LocalTime</code>	208
12.6 Methoden mit variabler Anzahl von Parametern	163	15.6 Datums- und Zeitangaben formatiert ausgeben – die Klasse <code>DateTimeFormatter</code>	210
12.7 Mit Aufzählungstypen (Enumerations) arbeiten	165	15.7 Die Klasse <code>System</code>	211
12.8 Übungen	170	15.8 Weitere Methoden der Klasse <code>System</code>	213
		15.9 Die Klasse <code>Console</code>	216
13 Collections-Framework – Grundlagen	171	15.10 Übungen	219
13.1 Grundlagen zum Java-Collections-Framework	171		
13.2 Das Interface <code>Collection</code>	173	Stichwortverzeichnis	222
13.3 Mit Listen arbeiten	174		
13.4 Listen sequenziell durchlaufen	176		
13.5 Übung	180		

1

Informationen zu diesem Buch

1.1 Voraussetzungen und Ziele

Einsatzbereiche

Das Buch dient dem Erlernen der Grundlagen der Programmierung mit Java. Es kann sowohl im Unterricht (in Schulen und Universitäten, im Bereich der Erwachsenenqualifizierung) als auch als Lehrbuch im Selbststudium und als Begleitbuch in E-Learning-Kursen eingesetzt werden.

Darüber hinaus kann das Buch als Nachschlagewerk und zur punktuellen Beschäftigung mit einzelnen ausgewählten Themen genutzt werden.

Zielgruppe

Das Buch richtet sich an Personen, welche die Programmierung mit Java erlernen wollen. Dies betrifft sowohl Programmier-Anfänger als auch Sprachumsteiger, die bereits Kenntnisse in anderen Programmiersprachen besitzen.

Da Java eine objektorientierte Sprache ist, kann das Buch auch von Personen, welche generell an der objektorientierten Programmierung interessiert sind, zum Wissenserwerb genutzt werden.

Empfohlene Vorkenntnisse

Um sich problemlos die Grundlagen der Programmierung mit Java aneignen zu können, sollten Sie bereits über folgende Kenntnisse und Fähigkeiten verfügen:

- ✓ PC-Grundlagenkenntnisse
- ✓ Grundkenntnisse im Umgang mit Windows 10 oder in einem Linux-System

Lernziele

- ✓ Erlangung von grundlegenden generellen Kenntnissen der Programmierung
- ✓ Verständnis der Prinzipien der objektorientierten Programmierung
- ✓ Beherrschung der Sprachsyntax und -bestandteile von Java
- ✓ Kenntnis wichtiger Bibliotheken von Java
- ✓ Eigenständige Programmierung von Anwendungen auf dem Niveau des entsprechend dem im Buch vermittelten Kenntnisstandes

Hinweise zu Soft- und Hardware

In den Beschreibungen des Buches wird von der Erstinstallation der Software Oracle Java Plattform, Standard Edition 9.0 (J2SE 9.0 JDK) ausgegangen (<http://www.oracle.com/technetwork/java/javase/downloads/jdk9-downloads-3848520.html>). Die Beschreibungen sind, soweit möglich, Betriebssystem unabhängig gehalten. Sofern erforderlich erläutert das Buch die Verwendung von Windows 10 und bei gravierenden Unterschieden zusätzlich auch die von Linux.

Als Texteditor für die Bearbeitung der Quelltexte (des Programmcodes) wurde der Editor Text-Pad Version 7.6.4 verwendet, der als Shareware unter <http://www.textpad.com/> erhältlich ist. Der Einsatz alternativer Editoren ist problemlos möglich. Einen Überblick dazu bietet Kapitel 3.

Installations- und Konfigurationshinweise zur Software finden Sie zum Download auf den Webseiten des HERDT-Verlags unter www.herdt.com im Bereich Buchplus, Ergänzende Lerninhalte. Nutzen Sie hierzu den **Webcode**, der sich rechts oben auf der Titelseite dieses Buches befindet.

Abhängig von der Bildschirmauflösung bzw. der Hardware Ihres Computers können die Symbole und Schaltflächen in den Programmen sowie die Fensterdarstellung unter Windows 10 gegebenenfalls von den Abbildungen im Buch abweichen.

1.2 Aufbau und Konventionen

Typographische Konventionen

Damit Sie bestimmte Elemente auf einen Blick erkennen und zuordnen können, werden diese im Text durch eine besondere Formatierung hervorgehoben. So werden beispielsweise Bezeichnungen für Programmelemente wie Register oder Schaltflächen immer *kursiv* geschrieben und wichtige Begriffe **fett** hervorgehoben.

<i>Kursivschrift</i>	kennzeichnet alle von Programmen vorgegebenen Bezeichnungen für Schaltflächen, Dialogfenster, Symbolleisten, Menüs bzw. Menüpunkte (z. B. <i>Datei - Schließen</i>) sowie alle vom Anwender zugewiesenen Namen wie Dateinamen, Ordernamen, eigene Symbolleisten wie auch Hyperlinks und Pfadnamen.
<code>Courier New</code>	kennzeichnet Programmcode.
<i>Courier New Kursiv</i>	kennzeichnet Zeichenfolgen, die vom Anwendungsprogramm ausgegeben oder in das Programm eingegeben werden.
[]	Bei Darstellungen der Syntax einer Programmiersprache kennzeichnen eckige Klammern optionale Angaben.
	Bei Darstellungen der Syntax einer Programmiersprache werden alternative Elemente durch einen senkrechten Strich voneinander getrennt.

HERDT BuchPlus – unser Konzept:

Problemlos einsteigen – Effizient lernen – Zielgerichtet nachschlagen

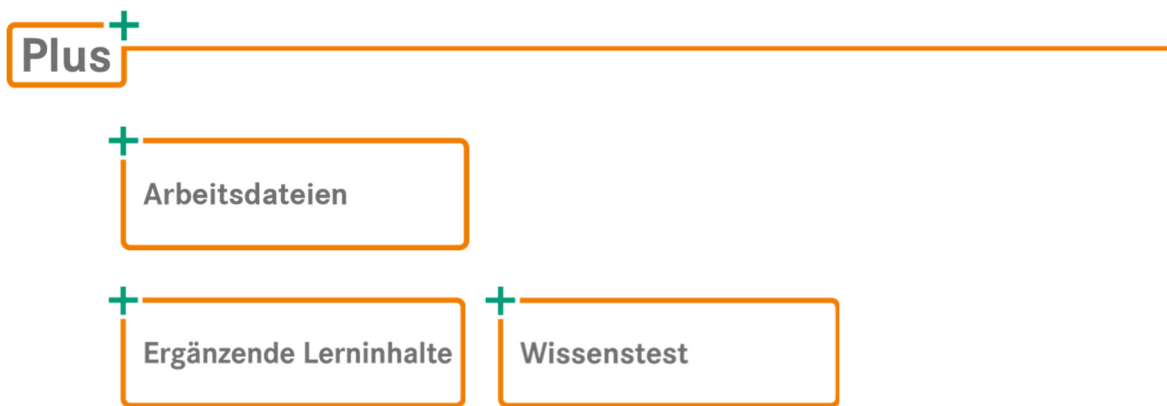
(weitere Infos unter www.herdtd.com/BuchPlus)

Für die meisten Kapitel stehen Ihnen Beispiel-, Übungs- und Ergebnisdateien zur Verfügung. Diese finden Sie im Ordner **Arbeitsdateien**.

Spezielle oder weiterführende Inhalte stehen Ihnen unter **Ergänzende Lerninhalte** zur Verfügung.

Anhand zweier **Wissenstests** können die Inhalte aus Kapitel 2 – 6 rekapituliert werden.

Nutzen Sie dabei unsere maßgeschneiderten, im Internet frei verfügbaren Medien:



- Rufen Sie im Browser die Internetadresse www.herdtd.com auf.

The screenshot shows the HERDT website. At the top, there is a navigation bar with links: 'Katalog', 'Shop', 'DE', 'AT', 'CH'. Below this, the HERDT logo is visible. A dropdown menu is open, showing options: 'Alles', 'Titel', 'Kategorien', 'Autor', and 'Codes'. The 'Codes' option is highlighted with a red box. To the right of the website screenshot, there is a green arrow pointing to a search bar. The search bar has a 'Codes' dropdown and a text input field. Below the search bar, there is a box with the text: '2 Geben Sie den folgenden Matchcode ein: JAV9.'

1 Wählen Sie Codes.

2

Einführung in Java

2.1 Die Programmiersprache Java

Entwicklung von Java

Java ist eine **objektorientierte plattformunabhängige Programmiersprache**, deren Grundlage 1991 bei Sun Microsystems gelegt wurde. Ziel war es, eine Programmierung für unterschiedliche Geräte zu ermöglichen, die wenig Speicher und eine geringe Rechenleistung besitzen. 1995 wurde der Name Java™ für die neue Programmiersprache vergeben. Java wurde und wird ständig weiterentwickelt. 1996 hielt Java Einzug auf Websites, weil Browser in der Lage waren, Java-Code zu interpretieren. Das erste offizielle JDK 1.0 (Java Development Kit – Java-Entwicklungswerkzeuge) wurde von Sun Microsystems veröffentlicht. Seitdem hat sich der Einsatz von Java auf verschiedene Bereiche ausgedehnt. Neben kleinen Anwendungen, die direkt im Browser ausgeführt werden, werden mittlerweile viele sehr umfangreiche Anwendungen mit Java entwickelt. Java ist damit heute eine der führenden Entwicklungsplattformen für Software. Aufbauend auf den reinen Sprachgrundlagen existiert eine Vielzahl sogenannter **Frameworks**, die für bestimmte Anwendungsfälle vorgefertigte Routinen beinhalten. In einer speziell dafür aufbereiteten Version fand Java auch Einzug in Kleingeräte wie beispielsweise Mobiltelefone und in eingebettete Systeme wie beispielsweise beim Fahrzeugbau.

Seit der Übernahme von Sun durch den Datenbankspezialisten Oracle in den Jahren 2009 und 2010 wird Java von Oracle weiterbetreut und -entwickelt.

Eigenschaften von Java

- ✓ Ein Java-Programm läuft prinzipiell auf allen Plattformen und Betriebssystemen, für die eine Java-Laufzeitumgebung vorliegt. Dadurch reduzieren sich die Entwicklungskosten, und die Entwicklungszeit wird verkürzt.
- ✓ Mit Java wird standardmäßig eine große Anzahl von Bibliotheken mitgeliefert, das sind fertige Programmteile, z. B. zur Grafikprogrammierung, zum Netzwerkzugriff oder zur Arbeit mit Dateien und Datenbanken.
- ✓ Was Sie zur Arbeit mit Java und zur Weitergabe Ihrer Programme benötigen, steht auf allen Plattformen meist kostenfrei zur Verfügung. Außerdem stehen Ihnen zahlreiche kostenfreie Entwicklungsumgebungen zur Verfügung.

- ✓ Die Sprache Java wird durch Oracle zentral standardisiert, sodass Sie nicht mit Inkompatibilitäten rechnen müssen.
- ✓ Für Java-Programme, die über einen Browser ausgeführt werden, gelten besondere Sicherheitsaspekte, die Manipulationen und Eingriffe über das Betriebssystem verhindern sollen.

Wann eignet sich Java für die Entwicklung einer Anwendung?

Wie bei jeder Software müssen Sie auch bei Java prüfen, ob der Einsatz für Ihren Verwendungszweck geeignet ist. Typischerweise werden Sie dazu auch Kompromisse eingehen und Schwerpunkte auf bestimmte Eigenschaften Ihres Projekts und das Einsatzgebiet der Software legen. Im Folgenden werden einige Hinweise gegeben, worauf Sie bei der Verwendung von Java als Programmiersprache achten sollten.

- ✓ Durch die etwas geringere Ausführungsgeschwindigkeit eignen sich Java-Programme nur bedingt für zeitkritische Anwendungen, die die maximale Performance eines Rechners nutzen.
- ✓ Besitzt die Zielgruppe für Ihr Programm eine heterogene Umgebung (Windows, Linux ...), können Sie durch den Einsatz von Java Entwicklungszeit sparen. Der übersetzte Programmcode ist auf allen Plattformen lauffähig, für die eine Java-Laufzeitumgebung verfügbar ist. Um eine plattformunabhängige Programmoberfläche zu entwickeln, ist allerdings mehr Vorarbeit notwendig als bei plattformspezifischen Anwendungen. Außerdem ist ein Test des Programms auf den entsprechenden Plattformen erforderlich. Eine Schaltfläche kann z. B. unter Windows quadratisch mit einem breiten Rand dargestellt werden, während sie unter Linux wesentlich breiter als hoch angezeigt wird und nur einen dünnen Rand besitzt. Dadurch kann sich das Erscheinungsbild Ihres Programms stark ändern. Grafikkomponenten, die in sogenannten **Bibliotheken** (z. B. JavaFX) zur Verwendung bereitgestellt werden, erleichtern Ihnen allerdings die Arbeit hier sehr.
- ✓ Viele Anwender verwenden Browser in ihren Arbeitsumgebungen. Durch den Einsatz von Java-Applets konnten in früheren Java- und Browser-Versionen aktuelle Informationen wie beispielsweise Aktienkurse und Verkaufsstatistiken angezeigt und grafisch ausgewertet werden. Mit der Weiterentwicklung der Webtechniken in Browsern wird das dafür notwendige Java-Plugin seit 2015 kaum noch unterstützt. Oracle plant deshalb, die Weiterentwicklung des Java-Browser-Plugins mit der Java Standard Edition Development Kit 9 (JDK 9) zu beenden.

Informationen dazu finden Sie unter

https://www.java.com/de/download/faq/jdk9_plugin.xml.

2.2 Das Java Development Kit (JDK)

Bestandteile des JDK

Das JDK enthält eine Vielzahl einzelner Programme (Tools), beispielsweise zum Übersetzen und Ausführen Ihrer Java-Programme. Außerdem enthält das JDK Bibliotheken, aus denen Sie Programmcode in Ihren Programmen nutzen können. Die Programme des JDK werden fast alle von der Kommandozeile der betreffenden Plattform (der Eingabeaufforderung unter Windows bzw. der Konsole unter Linux) gestartet.

Eine grafische Entwicklungsumgebung ist in dem JDK nicht enthalten. Seit der Entstehung von Java haben aber verschiedene Hersteller grafische Entwicklungsumgebungen für Java entwickelt. In diesem Buch wird darauf nicht weiter eingegangen.

Einige wichtige Elemente des JDK

- ✓ Der Java-Compiler `javac` zum Übersetzen des Java-Programms
- ✓ Der Java-Interpreter `java` zum Ausführen des vom Compiler erzeugten Bytecodes
- ✓ Bibliotheken, d. h. Sammlungen von Komponenten, beispielsweise zur Entwicklung von grafischen Anwendungen oder von Anwendungen mit Zugriff auf Dateien bzw. Ein- und Ausgabe über Tastatur und Bildschirm
- ✓ Weitere Programme, beispielsweise zur Erstellung von Dokumentationen und zur Erzeugung sogenannter Archive

Einsatzgebiete des JDK

Das JDK **Java 9** liegt für verschiedene Einsatzgebiete vor, die im Folgenden kurz vorgestellt werden.

Java™ SE Development	Diese Standard Edition (Java SE, früher J2SE genannt) wird zur Entwicklung von Programmen für Desktop-Computer eingesetzt. Sie enthält als Systemumgebung alle notwendigen Komponenten für die Entwicklung und Ausführung von Java-Programmen.
Java™ Enterprise Edition (Java™ EE)	Java EE ist ein Zusatz zum Java SE Paket. Mit der Enterprise Edition werden zusätzliche Bibliotheken für die Erstellung von Enterprise Applikationen (Geschäftsanwendungen) angeboten. Bestandteile der EE sind beispielsweise Enterprise JavaBeans (EJBs, eine Komponententechnologie). Einige in älteren Versionen zu Java EE gehörende Komponenten wie beispielsweise JAX-WS (Web-Services) und JNDI (Verzeichnisservice) sind mittlerweile Bestandteil der Standard Edition.
Java™ Micro Edition (Java™ ME)	Die Micro Edition basiert ebenfalls auf Java SE. Sie war für Anwendungen gedacht, die in kleinen Geräten wie Telefonen, Handheld-Computern (PDA), Kreditkarten etc. betrieben werden können, da diese Geräte in den ersten Generationen wenig Speicher und eine geringe Rechenleistung besaßen. Mit der fortschreitenden Entwicklung der Hardwarebasis sinkt die Bedeutung dieser Version.

Das Google Betriebssystem Android hat in den letzten Jahren eine immer stärkere Bedeutung im Markt der Mobilgeräte erreicht. Mit dem Software-Kit **Android SDK** unterstützt Google die Entwicklung von Java-basierten Anwendungen für Android. Das Android SDK enthält spezielle Bibliotheken, um beispielsweise Daten aus sozialen Netzen zu nutzen, Bilder und Videos anzuzeigen und den Datenaustausch mit anderen Geräten zu realisieren. Zum Testen der Anwendungen gehört zum SDK eine sogenannte **Virtual Machine** zur Simulation eines Android Gerätes auf dem PC.

- ✓ Dieses Buch beschreibt die Java 2 Standard Edition, Version 9.0 JDK (J2SE 1.XX). Die meisten Beispiele werden aber auch unter älteren Versionen laufen.
- ✓ Die Bezeichnung des JDK wechselte bei den bisherigen Versionen zwischenzeitlich zu SDK (Software Development Kit).

Java Runtime Environment (JRE)

Bereits im JDK ist eine Laufzeitumgebung (ein Interpreter) enthalten, mit der Sie während der Entwicklung Ihre Java-Programme testen können.

Aus lizenzrechtlichen Gründen benötigen Sie jedoch eine spezielle Laufzeitumgebung (JRE), wenn Sie diese zusammen mit Ihrem fertiggestellten Java-Programm weitergeben möchten. Die JRE ist weniger umfangreich als das JDK, denn sie enthält nur die Teile, die zum Ausführen von Java-Programmen benötigt werden.

- ✓ Gewöhnlich wird die JRE zusammen mit dem JDK installiert (*c:\Programme\Java\jre*).
- ✓ Falls Sie die Laufzeitumgebung (JRE) nicht zusammen mit dem JDK installiert haben, können Sie das J2SE 9.0 JRE (JRE 9.0) unter der folgenden URL beziehen:
<http://www.oracle.com/technetwork/java/javase/downloads/jre9-downloads-3848532.html>.

3

Ein Programm mit Java erstellen

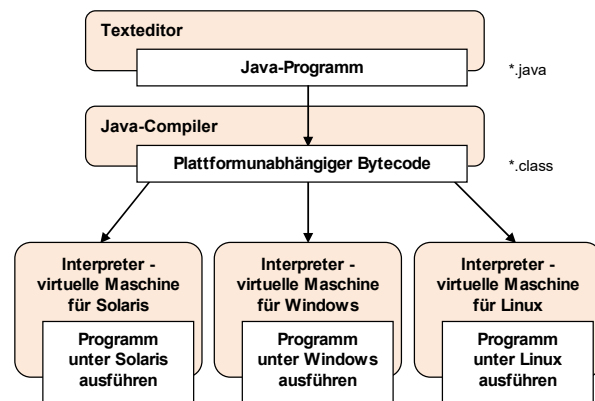
3.1 Ablauf bei der Programmerstellung

Schritte zur Erstellung eines Java-Programms

Die Entwicklung und Ausführung von Java-Programmen erfolgt vereinfacht in drei Schritten.

Java-Programm schreiben

Ein Java-Programm wird mit einem Texteditor, beispielsweise TextPad, geschrieben und als Textdatei mit der Dateinamenerweiterung `*.java` gespeichert. Diese Textdatei wird als **Quelltext** bzw. **Quellcode** oder Sourcecode bezeichnet. Ein Java-Programm besteht aus mindestens einer Quelltextdatei. In der Quelltextdatei werden die Anweisungen, die später auf dem Zielrechner ausgeführt werden sollen, mithilfe der Programmiersprache Java formuliert.



Java-Programm mit dem Java-Compiler übersetzen

Nach der Eingabe des Quelltextes wird das Java-Programm mit dem **Java-Compiler** übersetzt (kompiliert, engl. compile). Bei erfolgreicher Übersetzung wird ein **plattformunabhängiger** Zwischencode, der sogenannte **Bytecode**, erzeugt. Die Datei, in der der Bytecode gespeichert wird, besitzt denselben Namen wie die Quelltextdatei, jedoch die Dateinamenerweiterung `*.class`.

Die Plattformunabhängigkeit bedeutet, dass sich der Bytecode für die verschiedenen Rechnerplattformen (wie beispielsweise Windows und Linux) nicht unterscheidet. Allerdings kann der Bytecode auch nicht direkt ausgeführt werden.

Bytecode mit dem Java-Interpreter (der Java-Laufzeitumgebung) ausführen

Um das Programm ausführen zu können, benötigt der Anwender einen **Interpreter**, der in der Lage ist, den Bytecode auf der jeweiligen Plattform (z. B. Windows) auszuführen. Der Interpreter ist somit **plattformabhängig**. Für nahezu alle Plattformen existieren entsprechende Interpreter. Ein Java-Interpreter wird auch virtuelle Maschine (VM) genannt.

Ein Java-Interpreter ist in der Laufzeitumgebung (JRE) der Entwicklungsumgebung integriert. Wenn Sie die Laufzeitumgebung jedoch zusammen mit Ihren Programmen weitergeben möchten, benötigen Sie aus lizenzrechtlichen Gründen eine spezielle Laufzeitumgebung.

Anwendungen und Applets




Über das JDK haben Sie prinzipiell zwei Möglichkeiten, ein Programm auszuführen. **Applets** laufen im Kontext eines Webbrowsers, während **Anwendungen** selbstständig ablauffähig sind. Beide benötigen eine Java-Laufzeitumgebung, die beim Applet im Webbrowser integriert oder durch ein Plugin bereitgestellt wird. Für Anwendungen muss zusätzlich das JRE für die entsprechende Plattform (Windows, Linux, Solaris) installiert sein.



Die Unterstützung für Applets und das Java-Plugin ist in modernen Browsern, welche ca. ab dem Jahr 2015 veröffentlicht wurden, nicht mehr gegeben. Oracle empfiehlt deshalb die Verwendung alternativer Techniken, wie die **Java Web Start Technik**. Informationen dazu finden Sie unter https://www.java.com/de/download/faq/jdk9_plugin.xml. Die Programmierschnittstellen zur Applet Entwicklung und für das Java-Plugin sind ab der Version als **deprecated** gekennzeichnet. Dies bedeutet, dass sie noch genutzt werden können, für spätere Versionen aber keine Sicherheit besteht, dass sie noch unterstützt werden.

Für Anwendungen ist zusätzlich zu unterscheiden, ob diese als lokale Desktop-Anwendung direkt über die JRE gestartet oder über das Internet von einem Server bereitgestellt werden. Die zweite Variante wird mittels der **Java Web Start Technik** realisiert. Diese bietet eine Verwaltungsplattform, welche es ermöglicht, über das Internet bereitgestellte Java-Anwendungen mit einem Klick zu starten. Nach dem Herunterladen der Anwendung auf den lokalen PC verbleibt sie im Cache und steht ab diesem Moment zur sofortigen Ausführung bereit. Dabei kann bei jedem Start der Anwendung geprüft werden, ob es von dieser eine neuere Version gibt. Notwendige Updates werden in diesem Fall automatisch durchgeführt.

Java Web Start ist seit der Version Java 5.0 in der Java Runtime Environment (JRE) enthalten.

- ▶ Öffnen Sie die Kommandozeile über  .
- ▶ Starten Sie die Konsole. Geben Sie dazu `cmd` ein und betätigen Sie die Taste .
- ▶ Wechseln Sie mit dem Befehl `cd \programme\java\jdk-9\bin` ins Unterverzeichnis `bin` der Java-Installation.
- ▶ Mit dem Aufruf des Befehls `javaws` erhalten Sie Hilfeinformationen.

Editoren mit Java-Unterstützung und Java-Entwicklungsumgebungen

Für das Erfassen des Java Quelltextes existiert eine Vielzahl von Texteditoren, welche Sie bei der Programmierung unterstützen. Grundlegende, von den meisten Werkzeugen umgesetzte Funktionen sind dabei:

- ✓ Die Darstellung des Quelltextes in der Form, dass Java-spezifische Ausdrücke hervorgehoben werden.
- ✓ Die Möglichkeit des Aufrufs des Compilers und des Interpreters direkt aus dem Texteditor heraus. Deren separate Ansprache über das Eingabeaufforderungsfenster entfällt damit.

Der Umfang, der von den einzelnen Werkzeugen bereitgestellten Funktionen ist sehr unterschiedlich. Neben kleinen einfachen Texteditoren gibt es komplexe Entwicklungsumgebungen, welche im professionellen Bereich zum Einsatz kommen.

Die Wahl des Texteditors hat auf das zu erstellende Java-Programm keinen Einfluss und erfolgt nach persönlichen Vorlieben des Entwicklers oder gesetzten Anforderungen im Arbeitsumfeld.

Die Tabelle listet einige Editoren und Entwicklungsumgebungen auf. Die Liste stellt keine Empfehlung oder Wertung der einzelnen Texteditoren dar. Wird Java als erste Programmiersprache erlernt, ist es oft günstig, einen einfacheren Editor zu nutzen. Mit fortschreitenden Kenntnissen kann jederzeit auf eine komplexere und damit auch leistungsfähigere Entwicklungsumgebung umgestiegen werden.

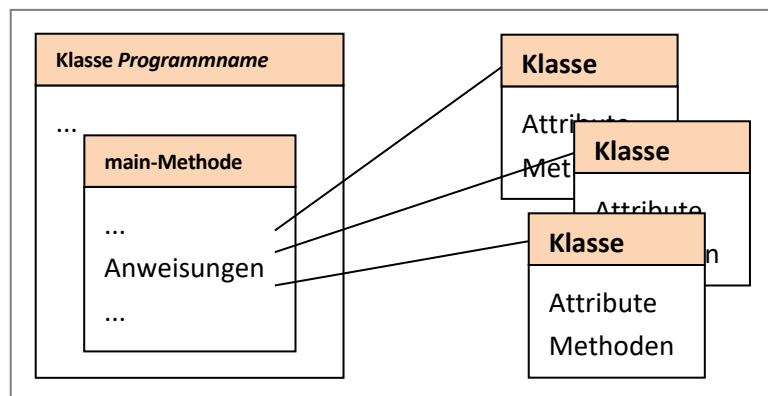
Eine komplette Beschreibung der Funktionen und Möglichkeiten der einzelnen Werkzeuge ist auf Grund der Informationsmenge hier nicht möglich. Viele Funktionen wie beispielsweise die von einigen Werkzeugen unterstützte Softwaremodellierung mittels der Unified Modelling Language (UML) setzen weiteres Spezialwissen voraus. UML-Kenntnisse können Sie z. B. mit dem HERDT-Buch „Objektorientierter Softwareentwurf mit UML“, siehe www.herdt.com, erwerben.

TextPad (Shareware) http://www.textpad.com	<ul style="list-style-type: none"> ✓ allgemeiner, nicht Java-spezifischer Texteditor ✓ ermöglicht das Starten von Compiler und Interpreter über eigene Menübefehle ✓ bequemes Einbinden von Startparametern
PSPad (Freeware) http://www.pspad.com/de/	<ul style="list-style-type: none"> ✓ allgemeiner, nicht Java-spezifischer Texteditor ✓ Starten des Compilers über das Menü, danach – wenn gewünscht – direkte Programmausführung
JavaEditor (Freeware) http://www.javaeditor.org	<ul style="list-style-type: none"> ✓ spezieller Editor für Java aus dem Bildungsbereich ✓ bietet erweiterte Umgebung für Java Entwicklung (auch für grafische Oberflächen) ✓ unterstützt interaktives Testen (Codeschnipsel)
BlueJ (Freeware) http://www.bluej.org/	<ul style="list-style-type: none"> ✓ spezieller Editor für Java aus dem Bildungsbereich ✓ der generelle Ansatz liegt auf der Unterstützung des Erlernens der objekt-orientierten Programmierung mit Java ✓ schnelles Testen von Klassen und Erstellen von Objekten möglich
Eclipse (Freeware) http://www.eclipse.org	<ul style="list-style-type: none"> ✓ komplexe Entwicklungsumgebung für den professionellen Einsatz ✓ Open Source Entwicklung ✓ bietet neben der eigentlichen Quellcodeerfassung eine Vielzahl von Funktionen zur Unterstützung des gesamten Softwarelebenszyklus ✓ mit zusätzlichen Funktionen erweiterbar
NetBeans (Freeware) https://netbeans.org/	<ul style="list-style-type: none"> ✓ komplexe Entwicklungsumgebung für den professionellen Einsatz ✓ Entwicklung mit Unterstützung von Oracle ✓ bietet neben der eigentlichen Quellcodeerfassung eine Vielzahl von Funktionen zur Unterstützung des gesamten Softwarelebenszyklus ✓ mit zusätzlichen Funktionen erweiterbar

IntelliJ IDEA (Freeware) https://www.jetbrains.com/idea/	<ul style="list-style-type: none"> ✓ komplexe Entwicklungsumgebung für den professionellen Einsatz ✓ bietet neben der eigentlichen Quellcodeerfassung eine Vielzahl von Funktionen u.a. einen GUI-Editor sowie Möglichkeiten zum Refactoring und Decompiling ✓ Erweiterung des Funktionsumfangs durch Plugins (auch selbstentwickelte) möglich ✓ Neben der Freeware Variante existiert eine kommerzielle Version.
---	---

3.2 Aufbau einer Anwendung

Java-Programme bestehen aus **Klassen** (engl. class). Diese Klassen beinhalten alle Informationen, die zur Ausführung notwendig sind. Die Informationen der Klassen umfassen **Attribute** (engl. attribute) (Daten), und **Methoden** (engl. method) (Funktionalität). Methoden beinhalten die **Anweisungen** (engl. statement), die ausgeführt werden sollen.



Einfache Java-Anwendung: alle Quelltextdateien in einem Ordner

Die einfachste Java-Anwendung besteht aus einer Klasse mit dem Namen der Anwendung. Diese Klasse enthält das Hauptprogramm, die sogenannte **main-Methode**.

Wenn Sie eine Anwendung mit dem Interpreter ausführen, sucht der Interpreter in der angegebenen Datei nach der Klasse mit dem Programmnamen und darin nach der main-Methode. Diese main-Methode wird ausgeführt. Dabei werden gewöhnlich weitere Klassen genutzt, die in separaten Dateien gespeichert sein können.

Zunächst speichern Sie alle Quelltextdateien in einem einzigen Ordner. Im Verlauf dieses Buches werden Sie weitere Möglichkeiten kennenlernen.

Beispiel: Java-Programm *HalloWelt.java* erstellen

- Erstellen Sie mit einem Texteditor den folgenden Quelltext zur Ausgabe des Textes *Hallo Welt!* und speichern Sie ihn unter dem Namen *HalloWelt.java*.

Der Quelltext wird im Beispiel hier nur kurz beschrieben, eine ausführlichere Erläuterung der einzelnen Bestandteile finden Sie in den nachfolgenden Kapiteln.

```

① class HalloWelt                                //Definition der Klasse
                                           Hallo Welt
② {
③   public static void main(String[] args) //Starten mit der Methode main
   {
④     System.out.println("Hallo Welt!"); ⑤//Ausgabe eines Textes
   }
}

```

Beispieldatei „HalloWelt.java“

- ① Mit dem Schlüsselwort `class` wird eine Klassendefinition eingeleitet. Danach folgt der Name der Klasse. Der Name der Klasse muss genauso geschrieben werden wie der Name der Datei, in der sie sich befindet, jedoch ohne Dateinamenerweiterung. Im Beispiel lautet der Name der Klasse `HalloWelt` und der Name der Datei `HalloWelt.java`.
- ② Innerhalb von geschweiften Klammern `{ }` steht die Definition einer Klasse.
- ③ Damit eine Applikation ausgeführt werden kann, benötigt sie die Methode `main`.
- ④ Um einen vorgegebenen Text, im Beispiel *Hallo Welt!*, am Bildschirm auszugeben, wird die Methode `System.out.println` verwendet. Der auszugebende Text wird in Anführungszeichen `" "` innerhalb runder Klammern `()` angegeben. Jede Java-Anweisung wird durch ein Semikolon `;` beendet, damit der Compiler erkennt, wo die auszuführende Anweisung zu Ende ist.
- ⑤ Wenn Sie innerhalb des Programmquelltextes Bemerkungen aufnehmen möchten, die z. B. den Code beschreiben, verwenden Sie Kommentare. Kommentare werden vom Compiler überlesen, wenn sie als solche beispielsweise mit `/ /` gekennzeichnet sind.

3.3 Ein Java-Programm mit dem Java-Compiler `javac` kompilieren

Mithilfe des Java-Compilers **javac** übersetzen Sie Ihre Java-Programme in Bytecode. Hierbei ist Folgendes zu beachten:




- ✓ Die Dateinamen der zu übersetzenden Java-Programme müssen mit der Endung `*.java` versehen sein. Außerdem unterscheidet der Compiler auch die Groß- und Kleinschreibung in den Dateinamen.
- ✓ Der Compiler erzeugt aus einer `*.java`-Datei eine `*.class`-Datei gleichen Namens. Befinden sich in einer `*.java`-Datei mehrere Klassendefinitionen, wird pro Klasse eine `*.class`-Datei erzeugt. Die `*.class`-Datei ist nicht mehr mit einem Editor lesbar.
- ✓ Beim Übersetzen überprüft der Compiler alle Abhängigkeiten einer `*.java`-Datei zu anderen Dateien, d. h., er prüft beispielsweise, ob die im Programm verwendeten Klassen vorhanden sind. Dazu geht der Compiler folgendermaßen vor:
 - ✓ Wird nur eine `*.class`-Datei gefunden, wird diese verwendet (da kein Quelltext vorliegt, kann dieser auch nicht übersetzt werden).
 - ✓ Wird nur eine `*.java`-Datei gefunden, wird diese übersetzt.
 - ✓ Findet der Compiler eine `*.java`- und eine `*.class`-Datei, überprüft er, ob die `*.java`-Datei neuer als die `*.class`-Datei ist. In diesem Fall wird die `*.java`-Datei neu übersetzt.

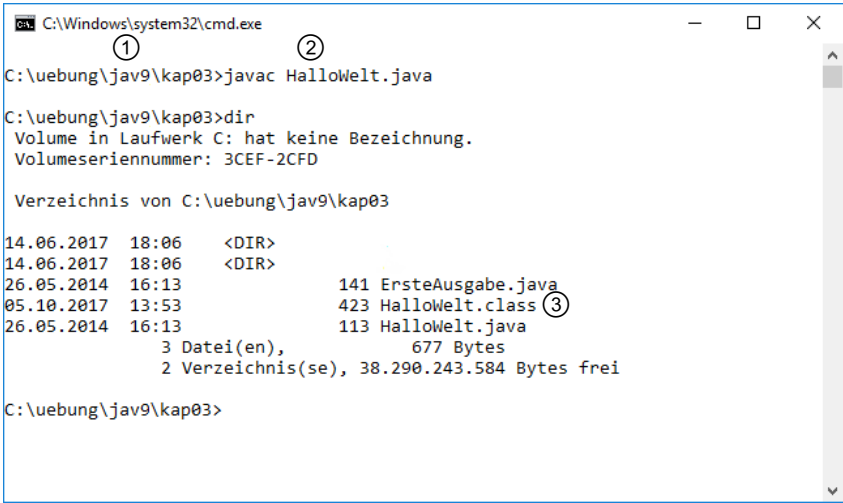
- ✓ Die neu erzeugte *.class-Datei wird vom Compiler automatisch in denselben Ordner wie die dazugehörige *.java-Datei gespeichert.

Im Folgenden wird neben den Werkzeugen des JDK an Hand von TextPad exemplarisch die Vorgehensweise in einem speziellen Texteditor gezeigt.

Beispielanwendung *HalloWelt.java* übersetzen

Wenn Sie ein Java-Programm über die Eingabeaufforderung (Windows) bzw. die Konsole (Linux) übersetzen möchten, geben Sie den dazu notwendigen Befehl in die Kommandozeile ein.

- ▶ Öffnen Sie z. B. in Windows die Eingabeaufforderung über die Tastenkombination  , Eingabe von *cmd* und Betätigen der Taste .
- ▶ Wechseln Sie in den Ordner, in dem Sie die Datei *HalloWelt.java* gespeichert haben ①. In diesem Beispiel ist dies der Ordner *C:\Uebung\jav9\kap03*.
- ▶ Übersetzen Sie das Programm *HalloWelt.java* mit dem Befehl *javac HalloWelt.java* ②.
- ▶ Überprüfen Sie mit dem Befehl *dir* (Windows) oder *ls* (Linux), ob eine Datei *HalloWelt.class* erzeugt wurde ③.



```

C:\Windows\system32\cmd.exe
C:\Uebung\jav9\kap03>javac HalloWelt.java
C:\Uebung\jav9\kap03>dir
Volume in Laufwerk C: hat keine Bezeichnung.
Volumeseriennummer: 3CEF-2CFD

Verzeichnis von C:\Uebung\jav9\kap03

14.06.2017  18:06    <DIR>          .
14.06.2017  18:06    <DIR>          ..
26.05.2014  16:13             141 ErsteAusgabe.java
05.10.2017  13:53             423 HalloWelt.class
26.05.2014  16:13             113 HalloWelt.java
               3 Datei(en),          677 Bytes
               2 Verzeichnis(se), 38.290.243.584 Bytes frei

C:\Uebung\jav9\kap03>
  
```

Übersetzung eines Java-Programms mithilfe der Eingabeaufforderung

Bedeutung der Path-Variablen

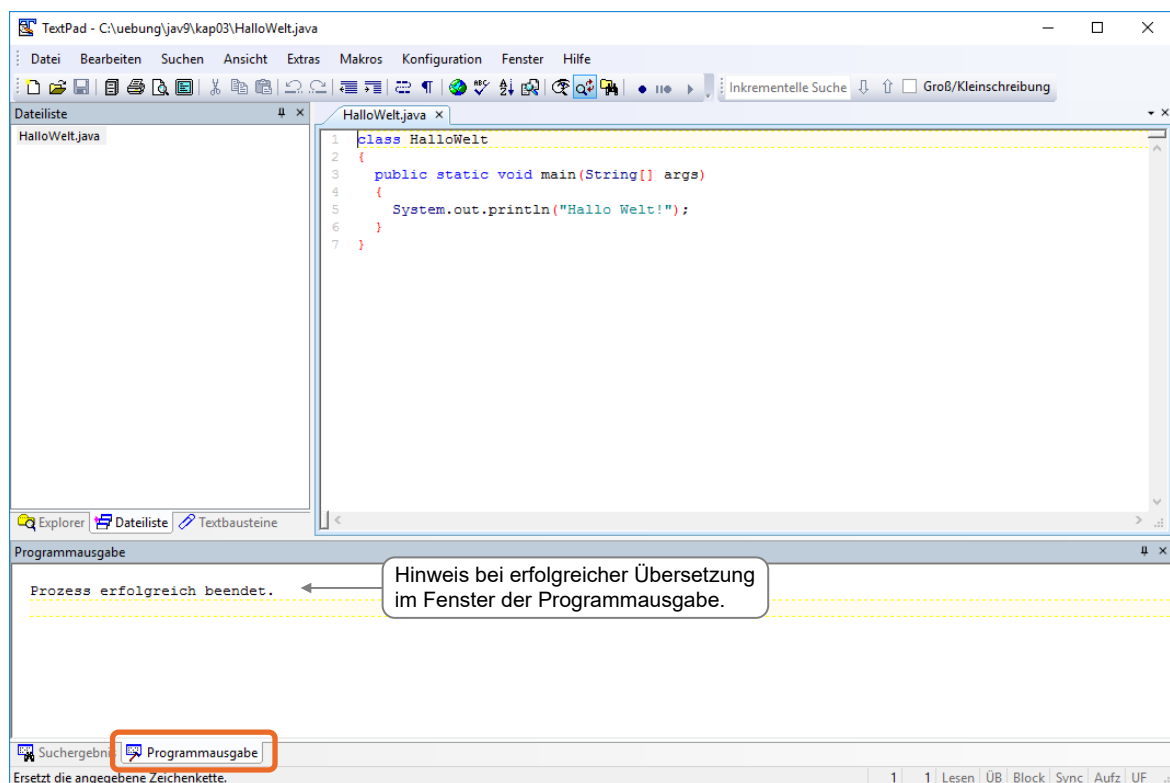
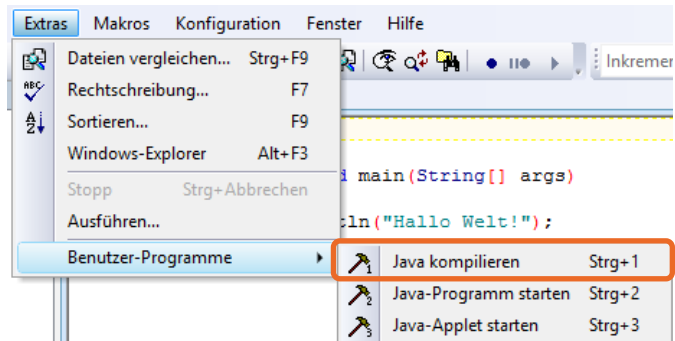
- ✓ Wenn Sie, wie im Anhang beschrieben, die Path-Variable gesetzt haben, wird der Compiler automatisch gefunden. Sie können den Compiler aus einem beliebigen Ordner heraus aufrufen.
- ✓ Wenn Sie die Path-Variable nicht entsprechend gesetzt haben, müssen Sie vor dem Namen des Compilers *javac* den Pfad angeben, in dem der Compiler gespeichert ist. Bei der Standardinstallation ist dies für Java 9 der Pfad *c:\Programme\java\jdk-9\bin*. Der Befehl für dieses Beispiel lautet dann:
c:\Programme\java\jdk-9\bin\javac HalloWelt.java

Beispielanwendung *HalloWelt.java* mit TextPad übersetzen

Wenn Sie mit dem Texteditor TextPad arbeiten, können Sie den Befehl zur Ausführung des Compilers direkt über den Editor aufrufen.

- ▶ Öffnen Sie die Quelltextdatei *HalloWelt.java*.
- ▶ Rufen Sie den Menüpunkt *Extras - Benutzer-Programme - Java kompilieren* auf, um die Bytecode-Datei mit der Erweiterung *.class* zu erzeugen.

Alternative: **Strg** **1**



3.4 Ein Java-Programm mit dem Interpreter java ausführen

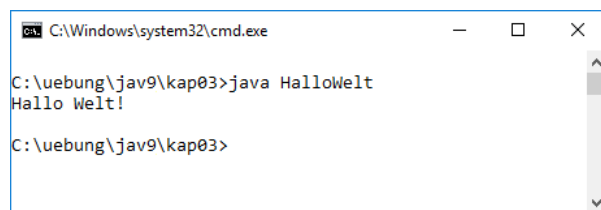
Vom Compiler übersetzte *.class*-Dateien sind noch nicht lauffähig. Die Dateien müssen vom Java-Interpreter **java** ausgeführt werden. Der Interpreter übernimmt für den Bytecode (*.class*-Datei) die Aufgabe, einen virtuellen Computer (virtuelle Maschine – Virtual Machine – VM) zu simulieren, der immer die gleichen Eigenschaften hat. Der Interpreter ist dann dafür verantwortlich, die Instruktionen im Bytecode in Maschinenanweisungen oder Anweisungen an das Betriebssystem auf dem betreffenden Rechner umzusetzen.

- ✓ Der Interpreter sucht nach einer Java-Klasse. Er unterscheidet dabei auch die Groß- und Kleinschreibung in den Dateinamen. Beispielsweise sucht der Interpreter beim Starten der Beispieldatei *HalloWelt.class* nach einer Klasse `HalloWelt`, die sich in der Datei *HalloWelt.class* befinden muss.
- ✓ Solange die Anwendung ausgeführt wird, kehrt der Interpreter nicht zurück, d. h., Sie erhalten keinen Eingabeprompt.
- ✓ Die beim Aufruf des Interpreters angegebene **.class*-Datei muss eine Methode `public static void main(String[] args)` besitzen, damit sie der Interpreter starten kann.

Beispielanwendung *HalloWelt* ausführen

Wenn Sie ein Java-Programm über die Eingabeaufforderung (Windows) bzw. die Konsole (Linux) übersetzen möchten, geben Sie den dazu notwendigen Befehl in die Kommandozeile ein.


- ▶ Um das Beispielprogramm *Hallo Welt* zu starten, geben Sie in die Kommandozeile der Eingabeaufforderung den Befehl `java HalloWelt` ein.
Die Anwendung gibt den Text "*Hallo Welt!*" in der folgenden Zeile aus.



```
C:\Windows\system32\cmd.exe
C:\uebung\jav9\kap03>java HalloWelt
Hallo Welt!
C:\uebung\jav9\kap03>
```

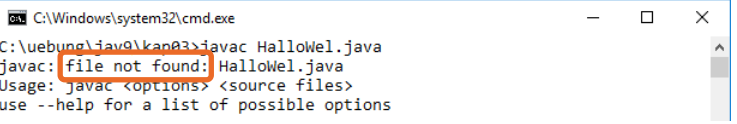
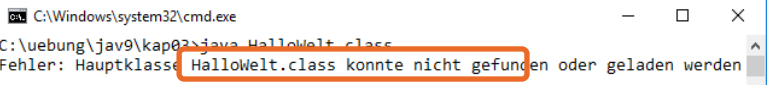
Beispielanwendung *HalloWelt* mit TextPad ausführen

Im Texteditor TextPad ist der Befehl zur Ausführung des Programms integriert:

- ▶ Um das Programm *HalloWelt* auszuführen, rufen Sie den Menüpunkt *Extras - Benutzerprogramme - Java-Programm starten* auf.
Alternative: `Strg` `2`
Es spielt dabei keine Rolle, in welchem Verzeichnis Sie sich befinden. Durch die `PATH`-Variable wird der Interpreter gefunden und durch die `CLASSPATH`-Variable die **.class*-Datei der Anwendung.
Die Anwendung gibt den Text "*Hallo Welt!*" im Eingabefenster unter Windows aus.
- ▶ Um das eingeblendete Fenster wieder zu schließen, betätigen Sie eine beliebige Taste oder das Schließfeld .


3.5 Problembehandlung

Beim Kompilieren oder beim Ausführen von Java-Programmen sowohl über die Eingabeaufforderung als auch über einen Texteditor mit integrierter Ausführungsfunktion können z. B. folgende Fehlermeldungen erscheinen:

Fehlermeldung	Bedeutung
<i>invalid flag</i>	 <p>Dieser Fehler tritt auf, wenn beim Compileraufruf eine ungültige Compileroption angegeben wird. Compileroptionen werden in diesem Buch nicht weiter erläutert.</p>
<i>file not found</i>	 <p>Diese Fehlermeldung erhalten Sie nach einem Compileraufruf, wenn die zu kompilierende Datei nicht vorhanden ist.</p>
<i>Fehler: Hauptklasse <Name der Klasse> konnte nicht gefunden oder geladen werden</i>	<p>Möglichkeit 1: Wenn keine entsprechende *.class-Datei vorhanden ist, wird beim Versuch, das Programm zu starten, der Fehler erzeugt.</p> <p>Möglichkeit 2:</p>  <p>Wenn Sie den Befehl <code>java HalloWelt.class</code> eingeben, erhalten Sie ebenfalls diese Fehlermeldung. Der Interpreter erwartet beim Aufruf über die Eingabeaufforderung keine Dateinamenerweiterung und sucht somit nach einer Datei mit dem Namen <code>HalloWelt.class.class</code>.</p>
<i>cannot find symbol</i>	 <p>Das Java-Programm ist syntaktisch fehlerhaft. Das Zeichen ^ kennzeichnet eine Position in der Zeile, die einen Fehler enthält. Im Beispiel fehlt bei der Methode <code>System.out.println</code> ein Punkt.</p>
<i>Fehler: Hauptmethode in Klasse <Name der Klasse> nicht gefunden</i>	 <p>Das Java-Programm besitzt keine <code>main</code>-Methode.</p>

3.6 Übung

Ausgabe erzeugen

Level		Zeit	ca. 10 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Programme erstellen ✓ Programme kompilieren und ausführen ✓ Verwenden des Texteditors und der Konsole 		
Übungsdatei	--		
Ergebnisdatei	<i>ErsteAusgabe.java</i>		

1. Erstellen Sie ein Programm, das folgende Ausgabe erzeugt: *Java-Programme bestehen aus Klassen*.
2. Speichern Sie das Programm unter dem Namen *ErsteAusgabe.java*.
3. Kompilieren Sie das Programm aus TextPad heraus.
4. Wenn das Kompilieren erfolgreich war, starten Sie das Programm aus TextPad.
5. Bauen Sie Fehler in das Programm ein. Schreiben Sie z. B. den Klassennamen anders als den Dateinamen und umgekehrt.
6. Kompilieren und starten Sie das Programm erneut. Verwenden Sie diesmal die Eingabeaufforderung. Löschen Sie gegebenenfalls fehlerhafte oder nicht benötigte **.java*- und **.class*-Dateien.

4

Grundlegende Sprachelemente

4.1 Was ist die Syntax?

Für jede Programmiersprache gelten bestimmte Regeln, die festlegen,

- ✓ welche Schreibweise für die Erstellung des Quelltextes erlaubt ist,
- ✓ welche Formulierungen dazu verwendet werden dürfen (Kombination aus Text, Zeichen, Symbolen).

Dieses Regelwerk wird als **Syntax** bezeichnet.

Die Syntax von Java legt somit fest, wie Sie den Quelltext formulieren und welche Sprachmittel die Programmiersprache Java dazu bereitstellt.

Hinweis zur Schreibweise der Syntax in diesem Buch

- ✓ Schlüsselwörter werden fett hervorgehoben.
- ✓ Optionale Angaben stehen in eckigen Klammern `[]`.
- ✓ Drei Punkte (...) kennzeichnen, dass weitere Angaben folgen können.
- ✓ Sofern eckige Klammern oder drei Punkte (...) als Bestandteil des Quelltextes erforderlich sind, wird darauf in der Erläuterung explizit hingewiesen.

4.2 Bezeichner und reservierte Wörter

Bezeichner festlegen

Bezeichner (Identifizier) sind alle frei wählbare Namen, mit denen Sie in Ihren Java-Programmen beispielsweise Variablen, Konstanten und Methoden benennen. Bei der Programmierung können Sie über einen Bezeichner auf diese Elemente zugreifen. Für den Aufbau eines Bezeichners gelten folgende Regeln:

- ✓ Bezeichner müssen mit 'a' ... 'z', 'A' ... 'Z', '_' oder '\$' beginnen und können dann beliebig fortgesetzt werden. Im Gegensatz zu früheren Java-Versionen ist in Java 9 die Verwendung von '_' als eigenständigen Bezeichnernamen nicht mehr möglich.

- ✓ Bezeichner beinhalten keine Leerzeichen und keine Sonderzeichen.
- ✓ Java verwendet den Unicode-Zeichensatz (1 Zeichen wird durch 2 Byte gespeichert). Deshalb können auch Buchstaben aus anderen Alphabeten genutzt werden. Dies ist allerdings nicht zu empfehlen. Schon im Sinne der besseren Lesbarkeit sollten Sie nur die Zeichen 'a' ... 'z', 'A' ... 'Z', '_' und '0' ... '9' verwenden.
- ✓ Java unterscheidet zwischen Groß- und Kleinschreibung. Name, NAME und name sind drei unterschiedliche Bezeichner für Elemente in Java.
- ✓ Java verwendet den Bezeichner in seiner kompletten Länge (alle Stellen sind signifikant).

Die hier aufgeführten Regeln sind vorgeschrieben. Für spezielle Elemente gelten Empfehlungen, an die sich Java-Programmierer üblicherweise halten. So ist der Quelltext leichter von anderen Programmierern zu verstehen. Diese Empfehlungen werden, sofern erforderlich, in dem jeweiligen Zusammenhang erläutert.

Reservierte Wörter

Java besitzt reservierte Wörter, die zum Schreiben eines Programms verwendet werden. Da jedes reservierte Wort eine feste, definierte Bedeutung besitzt, dürfen Sie diese nicht für eigene sogenannte **Bezeichner** einsetzen. Über Bezeichner (Identifier) können Sie die Programmteile in Java benennen, die Sie selbst erzeugen.

Die folgende Übersicht enthält alle reservierten Wörter von Java.

abstract	const	final	int	public	throw
assert	continue	finally	interface	return	throws
boolean	default	float	long	short	transient
break	do	for	native	static	true *)
byte	double	goto	new	strictfp	try
case	else	if	null *)	super	void
catch	enum	implements	package	switch	volatile
char	extends	import	private	synchronized	while
class	false *)	instanceof	protected	this	

Die drei mit *) gekennzeichneten reservierten Wörter stellen konstante vordefinierte Werte dar und werden als **Literale** bezeichnet. Alle anderen hier aufgeführten reservierten Wörter sind sogenannte **Schlüsselwörter**.

Die in der Version 9 von Java neu eingeführte Technik der Module zur Strukturierung von Anwendungen erfordert für ein Modul eine Deklarationsdatei. Innerhalb dieser gibt es folgende reservierte Schlüsselwörter:

exports	module	open	opens	provides	requires
uses	with	to	transitive		

Außerhalb einer Moduldefinition können diese Worte als Bezeichner verwendet werden. Schlüsselwörter werden in der Darstellung von Quelltextauszügen in diesem Buch fett hervorgehoben. Dies erleichtert Ihnen das Lesen der Quelltexte.

4.3 Quelltext dokumentieren

Mit Kommentaren den Überblick bewahren

Bei der Entwicklung eines Programms, insbesondere bei umfangreichen Programmen, sollten Sie den Quelltext mithilfe von Kommentaren detailliert erläutern:

- ✓ Kommentare erleichtern es Ihnen, zu einem späteren Zeitpunkt die Funktionsweise einzelner Teile wieder nachzuvollziehen.
- ✓ Mithilfe von Suchfunktionen können Sie die Quelltextdateien schnell nach Begriffen durchsuchen, die Sie im Kommentar verwendet haben.
- ✓ Sofern andere Personen das Programm weiterbearbeiten sollen, können sich diese leichter in den Quelltext einarbeiten.

Kommentare nehmen Sie direkt in den Quelltext Ihres Programms auf. Da die Kommentare aber nicht vom Compiler ausgewertet, sondern überlesen werden sollen, müssen Sie Ihre Erläuterungen im Quelltext speziell als Kommentare kennzeichnen.

Kommentare erstellen

In Java existieren zwei grundlegende Kommentartypen:

Einzeilige Kommentare	//	Kommentar bis zum Ende der Zeile Alle Zeichen der Zeile hinter // werden vom Compiler überlesen.
(Mehrzeiliger) Kommentarblock	/* */	Kommentar über mehrere Zeilen Ab der Zeichenkombination /* werden alle Zeichen im Quelltext überlesen, bis die Zeichenkombination */ auftritt.

Kommentare schachteln

- ✓ Ein Kommentarblock darf mit // gekennzeichnete einzeilige Kommentare beinhalten.
- ✓ Ein Kommentarblock darf **keinen** weiteren Kommentarblock enthalten.

Diese folgende Schreibweise ist beispielsweise fehlerhaft:

```
/* Beginn des Kommentarblocks
  /* Beginn eines zweiten Kommentarblocks
    Diese Zeile gehoert noch zum Kommentar
  */
  Diese Zeile wuerde als Programmzeile ausgewertet werden und Fehler
  hervorrufen
*/
```

Welchen Kommentartyp sollten Sie verwenden?

- ✓ Verwenden Sie für die Erläuterung Ihres Quelltextes einzeilige Kommentare.
- ✓ Verwenden Sie Kommentarblöcke, um zu Testzwecken Teile des Quelltextes als Kommentar zu kennzeichnen. Da dieser Abschnitt vom Compiler als Kommentar angesehen wird, wird er nicht berücksichtigt.

Ein Kommentarblock kann einzelilige Kommentare beinhalten. Dadurch können Sie einen Quelltextbereich als Kommentar kennzeichnen, ohne dass Sie Ihre Erläuterungen entfernen müssen.

Beispiele für Kommentare: *Comment.java*

Das folgende Beispiel zeigt die beiden Kommentierungsarten.

<pre> class Comment { public static void main(String[] args) { System.out.println("Hallo Welt!"); /* zu Testzwecken als Kommentar: System.out.println("Hallo Europa!"); */ } } </pre>	<pre> //Definition der Klasse //Comment (Kommentar) //Mit der Anweisung //System.out.println //koennen Sie einen Text //ausgeben //Mit der Anweisung //System.out.println //gekennzeichnet und wird //daher nicht ausgefuehrt </pre>
---	--

Quelltext kommentieren

Dokumentation mit javadoc

Das zusammen mit der Installation von Java installierte Programm javadoc können Sie dazu verwenden, automatisch eine Dokumentation für Ihren Programmcode zusammenzustellen. Dazu werden die Kommentare ausgewertet, die wie ein Kommentarblock gekennzeichnet sind, jedoch mit `/**` beginnen. Der erste Satz innerhalb dieses Kommentars wird als Beschreibung für die Dokumentation verwendet. Mithilfe spezieller Steuerzeichen können Sie beeinflussen, wie die Informationen in Ihren Kommentaren für die Dokumentation aufbereitet werden. Beispielsweise können Sie sich selbst mit `@Autor meinName` als Ersteller des Quellcodes bekannt geben. Eine Beschreibung, wie Sie javadoc verwenden, finden Sie in der Dokumentation zu Java.

4.4 Anweisungen in Java erstellen

Was sind Anweisungen?

Vom Programmierer geschriebene Anweisungen dienen zur Lösung einer Aufgabe. Ein Programm besteht aus einer Folge von Anweisungen, die in einer bestimmten Reihenfolge ausgeführt werden.

Syntax für Anweisungen

Die Syntax von Anweisungen ist durch folgende Regeln festgelegt:

- ✓ Eine Anweisung (Statement) besteht aus einer einfachen Anweisung oder aus einem Anweisungsblock.
- ✓ Eine einfache Anweisung wird mit einem Semikolon abgeschlossen.
- ✓ Ein Anweisungsblock fasst mehrere einfache Anweisungen in geschweiften Klammern `{ }` zusammen.
- ✓ Die Klammern `{ }` müssen immer paarweise auftreten.
- ✓ Anweisungsblöcke können geschachtelt werden.

```
statement;                //Eine einfache Anweisung

//oder

{                          //Beginn eines Anweisungsblocks
    statement1;
    statement2;
    ...                    //weitere Anweisungen
}                          //Ende des Anweisungsblocks
```

4.5 Primitive Datentypen

Datentypen

In den Anweisungen eines Programms arbeiten Sie mit Daten, die sich in ihrer Art unterscheiden:

- ✓ Zahlen (numerische Daten)
- ✓ Zeichen (alphanumerische Daten)
- ✓ Boolesche (logische) Daten

Damit genau festgelegt ist, welche Daten jeweils zulässig sind, besitzt Java einfache Datentypen, sogenannte primitive Datentypen. Sie unterscheiden sich in der Art der Daten, in dem zulässigen Wertebereich und in der Größe des dafür benötigten Speichers. Des Weiteren bestimmt der Datentyp die Operationen und Funktionen, die für diesen Datentyp angewendet werden können. Für jeden primitiven Datentyp ist der jeweils benötigte Speicherplatz immer gleich.

(1) Numerische Datentypen

Die numerischen Datentypen werden in **Integer**- und **Gleitkomma**-Typen unterteilt. Sie werden dann benötigt, wenn im Programm mit Zahlenwerten gearbeitet werden soll.

Integer-Datentyp

Integer-Datentypen stellen ganze Zahlen (ohne Nachkommastellen) mit Vorzeichen dar. Sie treten in vier verschiedenen Datentypvarianten auf, die sich in ihrem Wertebereich und der Größe des belegten Speicherplatzes unterscheiden.

Datentyp	Wertebereich	Speichergröße
byte	-128 ... 127	1 Byte
short	-32768 ... 32767	2 Byte
int	-2.147.483.648 ... 2.147.483.647	4 Byte
long	-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807	8 Byte

- ✓ Integer-Datentypen werden im Computer immer genau dargestellt.
- ✓ Es treten keine Rundungsfehler bei der Darstellung der Zahlen auf.

Üblicherweise werden Sie für ganzzahlige Werte den Datentyp `int` verwenden, denn er bietet für die meisten Anwendungsfälle einen ausreichenden Wertebereich.

Gleitkomma-Datentypen

Für **Fließkommazahlen** (Dezimalzahlen) werden Gleitkomma-Datentypen mit Vorzeichen verwendet. Der Computer kann jedoch nicht jede Zahl genau darstellen. Dies führt auch bei einfachen Rechnungen zu Rundungsfehlern. Je nach verwendetem Typ lassen sich nur Zahlen mit einer bestimmten Genauigkeit abbilden. Für eine höhere Genauigkeit wird aber auch mehr Speicherplatz benötigt.

Datentyp	Genauigkeit	Speichergröße
float	7 Stellen	4 Byte
double	15 Stellen	8 Byte

Üblicherweise werden Sie für Dezimalzahlen den Datentyp `double` verwenden, denn die Computer verfügen zumeist über ausreichenden Speicherplatz und beim Datentyp `float` machen sich Rundungsfehler deutlich bemerkbar.

(2) Zeichen-Datentyp

Für die Darstellung **alphanumerischer Zeichen** wird der primitive Zeichen-Datentyp `char` verwendet. Er kann ein beliebiges Unicode-Zeichen enthalten. Dadurch ist die Darstellung der Zeichen nicht nur auf Zahlen und Buchstaben begrenzt, sondern sie können auch Sonderzeichen wie `!""$$%&/` sowie Buchstaben anderer Alphabete enthalten. Sie können immer nur ein einziges Zeichen speichern.

Datentyp	Wertebereich	Speichergröße
<code>char</code>	Alle Unicode-Zeichen	2 Byte

(3) Boolescher (logischer) Datentyp

Java besitzt zur Repräsentation von Wahrheitswerten (wahr bzw. falsch) den Datentyp `boolean`. Ein Wahrheitswert kann nur `true` oder `false` als Wert annehmen.

Datentyp	Wertebereich	Speichergröße
<code>boolean</code>	<code>true</code> , <code>false</code>	1 Byte

`true` und `false` sind reservierte Wörter, die als Bezeichnung für die Wahrheitswerte **wahr** und **falsch** in Java festgelegt sind und als **boolesche Literale** bezeichnet werden. Die Literale `true` und `false` entsprechen nicht einer Zahl (beispielsweise 0 oder 1) wie in anderen Programmiersprachen.

4.6 Literale für primitive Datentypen

Werte, die Sie im Quelltext eingeben (z. B. Zahlen), werden als **Literale** bezeichnet. Für deren Schreibweise gelten entsprechende Regeln, damit der Datentyp ersichtlich ist.

Numerische Datentypen

- ✓ Für numerische Werte des Datentyps `int` können Sie beispielsweise die Vorzeichen `+` bzw. `-` und die Ziffern `0` ... `9` verwenden (das Vorzeichen `+` kann entfallen).
- ✓ Als Dezimaltrennzeichen bei Fließkommawerten verwenden Sie einen Punkt `.`.
- ✓ Für Fließkommawerte können Sie die Exponentialschreibweise verwenden. Den Wert $4,56 \cdot 10^3$ können Sie im Quelltext beispielsweise folgendermaßen eingeben: `4.56e3`. Die Schreibweise `4.56E3` ist ebenfalls zulässig, die Verwendung des Kleinbuchstabens erleichtert jedoch die Lesbarkeit.
- ✓ Wenn Sie explizit den Datentyp `long` benötigen, müssen Sie das Suffix `L` verwenden (z. B. `126L`).
- ✓ Wenn Sie explizit den Datentyp `float` benötigen, müssen Sie das Suffix `F` verwenden (z. B. `100.0F`).

Boolescher Datentyp

Für die Eingabe eines logischen Wertes existieren lediglich die beiden Literale `true` (wahr) und `false` (falsch).

Alphanumerischer Datentyp `char`

- ✓ Einzelne Zeichen werden bei der Wertzuweisung in Apostrophe `'` eingeschlossen.
- ✓ Sie können ein Zeichen auch als Unicode-Escape-Sequenz darstellen. Die Unicode-Repräsentation ist dann in Apostrophe `'` zu setzen. Escape-Sequenzen beginnen mit einem Backslash `\`. Die nebenstehende Tabelle zeigt eine Übersicht über die Escape-Sequenzen.
- ✓ Mit der Escape-Sequenz `\u` können Sie den Unicode für das gewünschte Zeichen direkt angeben.

Escape-Sequenz	Bedeutung
<code>\u...</code> Beispiel: <code>\u0045</code>	Ein spezielles Zeichen (im Beispiel das Zeichen E mit dem Code 0045)
<code>\b</code>	Backspace
<code>\t</code>	Tabulator
<code>\n</code>	line feed
<code>\f</code>	form feed
<code>\r</code>	carriage return
<code>\"</code>	Anführungszeichen
<code>\'</code>	Hochkomma
<code>\\</code>	Backslash

4.7 Mit lokalen Variablen arbeiten

Was sind Variablen?

Die Anweisungen eines Programms arbeiten mit Daten, die beispielsweise als Zwischenergebnis bei Berechnungen immer wieder verändert werden und somit variable Werte darstellen. Dazu werden sogenannte **Variablen** verwendet, für die entsprechender Speicherplatz im Arbeitsspeicher Ihres Computers reserviert wird. Im Programm greifen Sie auf diesen Speicherplatz über den **Variablennamen** zu. Variablen werden mit einem Namen und einem Datentyp definiert.

Voraussetzung für die Nutzung von Variablen

- ✓ Eine Variable muss definiert sein, bevor sie Daten aufnehmen kann.
- ✓ Eine Variable muss einen Wert enthalten, bevor sie verwendet werden kann.

Gültigkeitsbereich lokaler Variablen

Java kennt verschiedene Arten von Variablen. Variablen, die innerhalb eines Anweisungsblocks einer Methode definiert werden, werden als **lokale Variablen** bezeichnet und sind auch nur innerhalb des Blocks gültig. Dieser Bereich wird **lokaler Gültigkeitsbereich** genannt. Die lokale Variable verliert mit der schließenden Klammer `}` des Blocks ihre Gültigkeit. Ein Bezeichner in Java muss eindeutig sein. Das bedeutet, dass es nicht mehrere Variablen mit dem gleichen Namen geben kann, die an derselben Stelle im Quelltext gültig sind.

Die Eindeutigkeit einer Variablen bedeutet, dass sich der Gültigkeitsbereich von Variablen mit gleichem Namen nicht überschneiden darf. Sofern die Eindeutigkeit einer Variablen nicht gewährleistet ist, lehnt der Compiler die Definition der Variablen mit einem Fehlerhinweis ab. Die folgende Gegenüberstellung zeigt Variablendefinitionen, die aufgrund der Eindeutigkeit und der Gültigkeitsbereiche zulässig bzw. unzulässig sind.

Zulässige Variablendefinition	Unzulässige Variablendefinition
<pre> { int number; ① ... } //die Variable number ist hier //nicht mehr gueltig ... { int number; ② /* jetzt existiert wieder eine lokale Variable mit dem Namen number */ ... } </pre>	<pre> { int number; ③ ... ④ { int number; ⑤ //unzulaessig! ... } } /* diese Variablendefinition ist nicht zulaessig, da die Variable number aus dem uebergeordneten Block noch gueltig ist. */ </pre>

Zwei Beispiele für den Gültigkeitsbereich einer Variablen

- ✓ Nach der schließenden Klammer `}` ist die im Block definierte Variable ① nicht mehr gültig. Es kann eine neue Variable mit dem gleichen Namen definiert werden ②.
- ✓ Da Blöcke geschachtelt werden können, ist die Variable ③ auch im untergeordneten Block ④ gültig. Eine Variablendefinition mit demselben Namen ⑤ ist hier also **unzulässig**.

Syntax der Variablendefinition

- ✓ Die Variablendefinition besteht aus einem Datentyp ① und dem Namen ② der Variablen.

①
②
③
`type identifier[, identifier1...];`
- ✓ Die Definition einer Variablen ist eine Anweisung und wird mit einem Semikolon abgeschlossen.
- ✓ Die Namen der Variablen werden direkt nach dem Datentyp, getrennt durch ein oder mehrere Leerzeichen, angegeben.
- ✓ Mehrere Variablen desselben Typs können Sie in einer Anweisung definieren. Die Variablennamen werden dabei durch Kommata getrennt ③.
- ✓ Die Namen der Variablen halten sich an die Vorgaben für Bezeichner.

Benennung und Definition von Variablen

- ✓ Variablennamen beginnen üblicherweise mit einem Kleinbuchstaben.
- ✓ Möchten Sie einen Namen aus mehreren Wörtern zusammensetzen, schreiben Sie direkt hinter das erste kleingeschriebene Wort **ohne ein Trennzeichen** (z. B. einen Unterstrich) die weiteren Wörter. Beginnen Sie diese Wörter zur Abgrenzung jeweils mit Großbuchstaben. Beispiele für Variablennamen sind `time`, `maxTextLength`, `timeToEnd`.

- ✓ Definieren Sie nur eine Variable pro Zeile. Kommentare lassen sich übersichtlich einfügen. Die Lesbarkeit des Quelltextes wird verbessert, was auch eine mögliche Überarbeitung erleichtert.

Beispiele für Variablendefinitionen: *VariableDefinition.java*

```
//--richtige Definition -----
int number;           //definiert eine Variable number vom Typ int (Integer)
double price, size;   //definiert zwei Variablen vom Typ double
char c;               //definiert eine Variable c vom Typ char (Character)
//--fehlerhafte Definition -----
int &count;           //Bezeichner von Variablen dürfen kein & enthalten
double a b c;         //mehrere Variable müssen durch Kommata getrennt
                      //werden
```

4.8 Werte zuweisen

Wertzuweisung und Initialisierung von Variablen

Mit der Definition einer lokalen Variablen ist durch den Datentyp lediglich festgelegt, welche Daten in der Variablen gespeichert werden können. Der Wert ist noch unbestimmt, das heißt: Die Variable ist noch nicht **initialisiert**.

Bevor Sie auf den Wert einer Variablen zugreifen können, müssen Sie dieser Variablen einen Wert zuweisen. Bei der Definition einer lokalen Variablen geschieht dies **nicht** automatisch. Beachten Sie außerdem, dass einer Variablen nur Werte zugewiesen werden dürfen, die dem vereinbarten Typ entsprechen.

Die erste Wertzuweisung nach der Definition einer Variablen wird **Initialisierung** genannt. Aber auch im weiteren Verlauf des Programms kann der Wert einer Variablen geändert werden, es wird ein neuer Wert zugewiesen. Die Wertzuweisung an Variablen erfolgt nach der folgenden Syntax:

Syntax für Wertzuweisungen an Variablen

- ✓ Sie beginnen die Wertzuweisung mit dem Bezeichner (dem Namen) der Variablen. `identifizier = expression;`
- ✓ Anschließend folgt als **Zuweisungsoperator** ein Gleichheitszeichen (=) und der Ausdruck (Expression), der zugewiesen werden soll.
- ✓ Häufig ist Expression ein Wert, den Sie direkt beispielsweise als Zahl eingeben (Literal). Expression kann aber auch eine andere Variable sein, sofern diese bereits einen Wert besitzt, oder ein komplexer Ausdruck. Komplexe Ausdrücke werden Sie im weiteren Verlauf dieses Kapitels kennenlernen.
- ✓ Die Wertzuweisung wird als Anweisung (Statement) mit einem Semikolon (;) abgeschlossen.

Tipps und Hinweise zur Schreibweise

Sie können die Definition und die Initialisierung einer Variablen in einer einzigen Anweisung vornehmen. Dabei fügen Sie die Wertzuweisung direkt an die Definition an.

```
type identifier = value;
```

Syntax für Definition und Initialisierung in einer Anweisung

Es lassen sich auch mehrere Variablen eines Typs in einer Anweisung definieren und initialisieren. Diese Schreibweise ist nicht sehr übersichtlich, wird in diesem Buch in den Beispielen aus Platzgründen aber gelegentlich verwendet.

```
int result = 0, number = 0, counter = 1;
```

Beispiel für eine mögliche, aber unübersichtliche Schreibweise

Beispiele für Wertzuweisungen: *Assignment.java*

```
int i;           //Definitionen
int k;
char c;
double d = 1.7;  //zusätzlich zur Definition kann eine Variable
                //initialisiert werden
//--richtige Wertzuweisungen -----
i = 20;          //der Variablen i wird der Wert 20 zugewiesen
k = i;           //der Wert der Variablen i wird ausgelesen und
                //der Variablen k zugewiesen
c = 'A';         //der Variablen c wird das Textzeichen 'A' zugewiesen
//--fehlerhafte Wertzuweisungen -----
i = 0.15;        //0.15 ist kein gueltiger int-Wert
c = "Test";      //"Test" ist kein char-Wert
v = 7;           //hier ist keine Variable mit dem Namen v definiert
```

4.9 Typkompatibilität und Typkonversion

Typkompatibilität

Sie können einer Variablen nur den Wert des Datentyps zuweisen, den sie selbst besitzt oder den sie aufgrund ihres Wertebereichs umfasst. Zum Beispiel können Sie einer Variablen vom Typ `double` den Wert einer Variablen vom Typ `int` zuweisen. Der Datentyp `int` ist **kompatibel** zum Datentyp `double`. Der Datentyp `double` ist aber **nicht kompatibel** zum Datentyp `int`.

Typkonversion

Bei kompatiblen Typen führt Java automatisch eine **implizite Typkonversion** durch. So können Sie beispielsweise eine Ganzzahl (Typ `int`) problemlos einer Variablen vom Typ `double` zuweisen.

Sollten die Typen nicht kompatibel sein, so können Sie eine **explizite Typkonversion** durchführen. Logische Werte (`boolean`) können nicht umgewandelt werden.

Syntax für die explizite Typkonversion

```
(type) expression
```

Beispiele für Typkonversion: *TypeCast.java*

Die nachfolgenden Beispiele zeigen die Typumwandlung bei einfachen Wertzuweisungen:

```
① int position = 100; //korrekt
  double size = 1.45; //korrekt
② double weight = 80; //korrekt
③ int number = 1.23; //Compiler liefert eine Fehlermeldung
④ int num = (int)5.67; //korrekt durch die erzwungene
                        //Datentypkonversion. num erhält den
                        //Wert 5; die Dezimalstellen entfallen
```

- ① Die ersten beiden Zuweisungen sind korrekt, denn die Zahl 100 ist ganzzahlig und passt daher zu der Variablen vom Typ `int`, bzw. die Zahl 1.45 passt zur Variablen vom Typ `double`.
- ② Die Zuweisung ist ebenfalls korrekt, denn die Zahl 80 ist zwar ganzzahlig, sie ist aber Bestandteil der rationalen Zahlen und passt daher zu der Variablen vom Typ `double`. Diese (implizite) Datentypkonvertierung wird von Java automatisch durchgeführt.
- ③ Die Zahl 1.23 ist eine Zahl mit Nachkommastellen und gehört daher nicht zu den ganzen Zahlen. Java nimmt keine (implizite) Datentypkonversion wie bei der Zuweisung ② vor. Der Compiler liefert eine entsprechende Fehlermeldung.
- ④ In der Anweisung wird eine explizite Datentypkonvertierung (type cast) vorgenommen.

! Java führt, sofern es möglich ist, die explizite Typkonversion durch, ohne den Sinn des Ergebnisses zu berücksichtigen. Die Verantwortung für den sinnvollen Einsatz der erzwungenen Typumwandlung obliegt dem Programmierer.

Implizite Datentypkonversion vermeiden

Auch bei unterschiedlichen kompatiblen Datentypen sollten Sie eine **explizite Datentypkonversion** angeben. Als Programmierer kennzeichnen Sie dadurch, dass Sie sich darüber bewusst sind, dass es sich um verschiedene Datentypen handelt.

```
...
double value = 0.0;
int number = 5;
...
value = (int)number;
...
```


4.10 Konstanten – unveränderliche Variablen

Konstanten nutzen

In einem Programm werden häufig Werte benötigt, die sich nicht mehr ändern, wie beispielsweise die Schallgeschwindigkeit, ein Mehrwertsteuersatz oder der Umrechnungsfaktor von mm in Inch. Diese Konstanten werden durch spezielle unveränderliche Variablen dargestellt. Sobald während der Programmausführung eine Wertzuweisung erfolgt ist, ist diese endgültig (*final*). Sie können also keine zweite Wertzuweisung an eine Konstante vornehmen, nachdem sie initialisiert wurde.

Konstanten vereinfachen die Lesbarkeit und vermeiden Fehler: Der Wert einer Konstanten wird nur einmal eingegeben. Im weiteren Verlauf des Programms arbeiten Sie nur noch mit dem Namen der Konstanten. Sofern der Wert der Konstanten korrigiert werden muss, ist diese Änderung nur an einer einzigen Stelle im Quelltext vorzunehmen.

Wie für eine lokale Variable wird auch für jede Konstante Speicherplatz im Arbeitsspeicher Ihres Computers reserviert. Im Programm greifen Sie auf diesen Bereich über den Namen der Konstanten zu.

Wie eine Variable ist eine Konstante auch nur in dem Anweisungsblock gültig, in dem sie definiert wurde. Jede Konstante, die Sie in Ihrem Programm verwenden, muss vorher definiert werden.

Syntax der Konstantendefinition und -initialisierung

- ✓ Die Konstantendefinition wird mit dem Schlüsselwort `final` eingeleitet.
- ✓ Es folgen der Datentyp und der Name der Konstanten.
- ✓ Mehrere Konstanten können Sie, durch Kommata getrennt, in einer Anweisung definieren.
- ✓ Wie lokale Variablen können Sie Konstanten zusammen mit der Definition auch initialisieren oder später in **einer** separaten Anweisung.
- ✓ Die Namen der Konstanten halten sich an die Vorgaben für Bezeichner.

```
final type identifier1[,  
    identifier2...];  
identifier1 = expression1;  
...
```

Konstanten definieren und initialisieren

Die Schreibweise für Konstanten weicht üblicherweise von der Schreibweise für Variablen ab: Zur Kennzeichnung einer Konstanten werden nur Großbuchstaben verwendet. So können Konstanten sofort als solche erkannt werden und der Programmierer weiß, dass beispielsweise eine weitere Wertzuweisung nicht möglich ist. Sofern sich der Name einer Konstanten aus mehreren Wörtern zusammensetzt, wird der Unterstrich `_` zur Trennung dieser Wörter verwendet. Konstanten könnten beispielsweise `SONIC_SPEED`, `MAX_USERS` oder `MWST` heißen. Ab der Version Java 7 dürfen Namen für numerische Konstanten außer am Anfang und am Ende an beliebigen Stellen einen oder mehrere Unterstriche `_` enthalten. Dies erhöht die Übersichtlichkeit der Darstellung.

Konstanten verwenden

Sie schreiben beispielsweise ein Programm, in dem an vielen Stellen Berechnungen durchgeführt werden, die die aktuelle Mehrwertsteuer verwenden. Durch die Einführung einer Konstanten wird der Programmcode besser lesbar und leichter änderbar. Wenn sich die Höhe der Mehrwertsteuer verändert, brauchen Sie bei Verwendung einer Konstanten nur einmal den Wert der Konstanten zu ändern, statt im gesamten Programmquelltext die Zahl 0.19.

```
// Definition und Initialisierung der Konstanten MWST
final double MWST = 0.19;
...
// Verwendung der Konstanten MWST in Berechnungen
```

4.11 Arithmetische Operatoren und Vorzeichenoperatoren

Ausdrücke mit Operatoren bilden

Häufig werden Berechnungen benötigt, deren Ergebnisse anschließend in Variablen gespeichert werden. Mithilfe von **Operatoren** können Sie Ausdrücke bilden, die z. B. eine Formel zur Berechnung darstellen.

Ein **Ausdruck** (Expression) ist allgemein eine Kombination aus Werten, Variablen, Konstanten und Operatoren.

- ✓ Der Ausdruck wird ausgewertet.
- ✓ Der letztendlich ermittelte Wert wird der Variablen zugewiesen.

```
identifizier = expression;
```

Einige Operatoren wie die zur Addition (+), Subtraktion (-), Multiplikation (*) und Division (/) kennen Sie als Grundrechenarten. Für die Ausführungsreihenfolge in einem Ausdruck werden Operatoren nach Prioritäten eingeteilt. Die Ausführungsreihenfolge kann jedoch auch durch das Setzen von runden Klammern () festgelegt werden, die die höchste Priorität besitzen. Der Inhalt der Klammern wird immer zuerst ausgewertet.

Folgende Operatoren können Sie in Java verwenden:

Arithmetische Operatoren

Zu den arithmetischen Operatoren gehören in Java die Grundrechenarten (+, -, *, /) und der Modulo-Operator (%). Diese Operatoren benötigen jeweils **zwei Operanden**, die mit dem Operator verknüpft werden, und werden daher als **binäre Operatoren** bezeichnet.

Die Grundrechenarten können auf Integer- und auf Gleitkommawerte angewendet werden. Bei der Anwendung der Operatoren gilt wie in der Mathematik die Regel „Punktrechnung geht vor Strichrechnung“. Das bedeutet, dass die Operatoren (*) und (/) eine höhere Priorität besitzen als die Operatoren (+) und (-). Neben der Priorität ist die sogenannte **Assoziativität (Bindung)** für die Auswertung entscheidend. Sie legt fest, in welcher Richtung die Auswertung erfolgt.

Alle Operatoren mit der gleichen Priorität besitzen auch die gleiche Assoziativität. Die arithmetischen (binären) Operatoren besitzen eine Links-Assoziativität, d. h., sie sind **linksbindend**.

Beispiel

①	$3 + 4 * 5 + 6 + 7$	①	Nach der Priorität wird zunächst der Ausdruck $4*5$ ausgewertet.
②	$= 3 + 20 + 6 + 7$	②	Alle Operatoren haben jetzt die gleiche Priorität und sind alle linksbindend.
③	$= (3 + 20) + 6 + 7$	③	Die Auswertung erfolgt daher von links nach rechts. (Die Klammern sind hier nicht erforderlich, sondern dienen nur zur Veranschaulichung.)
	$= (23 + 6) + 7$		
	$= 29 + 7$		
	$= 36$		

Auswertung eines Ausdrucks

Beispiele für den Einsatz der Grundrechenarten

- ✓ Geklammerte Ausdrücke werden immer zuerst ausgewertet und „Punktrechnung geht vor Strichrechnung“.

```
int i = 3 + 4 * 5; //die Variable i erhaelt hier den Wert 23,
                  //da der Ausdruck 4 * 5 zuerst ausgewertet wird
```

```
int i = (3 + 4) * 5; //dieser Ausdruck weist der Variablen i den Wert 35
                    //zu, da geklammerte Ausdruecke zuerst ausgewertet
                    //werden
```

- ✓ Wenn sich bei einer Berechnung mit Integer-Werten Kommastellen ergeben, lässt Java diese einfach wegfallen. Java wendet das Verfahren Division mit Rest an (ein Divisionsrest wird in Kauf genommen), der verbleibende Rest bleibt dann unberücksichtigt.

```
int i = 14 / 5; //i erhaelt den Wert 2, Kommastellen entfallen
```

- ✓ Der `%`-Operator, auch Modulo-Operator genannt, bestimmt den Rest bei einer Division von Integer- oder Double-Zahlen.

```
int i = 14 % 5 //i erhaelt den Wert 4, denn 14 / 5 = 2, Rest = 4
```

Verkürzte Schreibweisen verwenden

Für die arithmetischen Operatoren mit Wertzuweisungen gibt es in Java die verkürzten Schreibweisen. Häufig treten Anweisungen auf, mit denen ein Wert einer Variablen ausgelesen, dann verändert und anschließend wieder in derselben Variablen gespeichert wird.

```
variable = variable operator expression;
```

Häufige Anweisung

Der folgende Quelltextauszug zeigt anhand einiger Beispiele, wie Sie die Schreibweise solcher Anweisungen verkürzen können:

Beispiel: *ReducedNotation.java*

```

int i = 15;
int k = 3;

i += 5;      //ausfuehrlich: i = i + 5; -> i erhaelt den Wert 20
i += k;      //ausfuehrlich: i = i + k; -> i erhaelt den Wert 23
i -= 7;      //ausfuehrlich: i = i - 7; -> i erhaelt den Wert 16
i /= k;      //ausfuehrlich: i = i / k; -> i erhaelt den Wert 5
i *= 7;      //ausfuehrlich: i = i * 7; -> i erhaelt den Wert 35
i %= 4;      //ausfuehrlich: i = i % 4; -> i erhaelt den Wert 3

```

Inkrementierung und Dekrementierung

Die Operatoren ++ und -- sind Sonderfälle der Addition bzw. Subtraktion und heißen Inkrement- und Dekrement-Operator. Sie können nur auf Variablen angewendet werden. Der Inkrement-Operator erhöht den Wert der Variablen um 1, der Dekrement-Operator erniedrigt den Wert der Variablen um 1.

Beispiel: *IncrementDecrement.java*

```

int i = 3;
double d = 1.5;

i++;        //ausfuehrlich: i = i + 1 -> i erhaelt den Wert 4
d--;        //ausfuehrlich: d = d - 1.0 -> d erhaelt den Wert 0.5

```

Postfix- und Präfix-Notation

Zusätzlich wird bei den Operatoren ++ und -- zwischen Postfix- und Präfix-Notation unterschieden. Die Operatoren stehen in diesem Fall hinter bzw. vor der Variablen. Bei der Postfix-Notation wird die Variable nach dem Auswerten des Ausdrucks inkrementiert bzw. dekrementiert, bei der Präfix-Notation davor.

Solange die Anweisung lediglich die Inkrementierung bzw. Dekrementierung ausführt, haben Postfix- und Präfix-Notation die gleiche Wirkung.

Erst bei der Verwendung innerhalb eines Ausdrucks wird der Unterschied deutlich:

```

int i = 5, j = 5;
i++; //entspricht ++i;
    //i erhaelt den Wert
    6;
j--; //entspricht --j;
    //j erhaelt den Wert
    4;

```

Beispiel: *PrefixPostfix.java*

```

int i = 10, j = 10;
① int result1 = 2 * ++i;    //result1 erhält den Wert 22
② int result2 = 2 * j++;    //result2 erhält den Wert 20

```

Präfix- und Postfix-Notation

- ① Die Berechnung erfolgt in zwei Schritten:
 1. Schritt: `i = i + 1;`
 2. Schritt: `result1 = 2 * i;`
`result1` erhält den Wert 22.
- ② Berechnung erfolgt auch hier in zwei Schritten:
 1. Schritt: `result2 = 2 * j;`
 2. Schritt: `j = j + 1;`
`result2` erhält den Wert 20.

Um Fehlinterpretationen zu vermeiden, sollten Sie nicht beide Notationen in einer Anweisung verwenden.

Vorzeichenoperatoren

Bei negativen Zahlen kennen Sie die Kennzeichnung mit einem vorangestellten `-`. Bei positiven Zahlen könnten Sie ein `+` voranstellen, das aber entfallen kann. Diese Vorzeichenoperatoren werden als **unäre Operatoren** bezeichnet, denn sie besitzen nur einen Operanden.

- ✓ Die unären Vorzeichenoperatoren können nicht nur auf Zahlen, sondern auch auf Variablen bzw. auf ganze Ausdrücke angewendet werden.
- ✓ Die unären Vorzeichenoperatoren besitzen eine höhere Priorität als die binären (arithmetischen) Operatoren.
- ✓ Die Auswertung der unären Vorzeichenoperatoren erfolgt von rechts nach links. Das heißt, unäre Vorzeichenoperatoren besitzen eine **Rechts-Assoziativität** (Rechts-Bindung).
- ✓ Die folgenden Beispiele zeigen die Verwendung der unären Vorzeichenoperatoren:

Beispiel: *Sign.java*

```

int i = 3, k = 5;
int result = 0;
① result = -i;                //result erhält den Wert -3
② result = -(i - 5);          //result erhält den Wert 2
③ result = -(-3);             //result erhält den Wert 3

```

Vorzeichen-Operatoren anwenden

- ① `result` erhält den Wert -3.
- ② Zuerst wird die Klammer ausgewertet und liefert das Ergebnis: -2.
 Anschließend wird durch den Vorzeichenoperator `-` das Vorzeichen gewechselt.
`result` erhält den Wert 2.
- ③ `result` erhält den Wert 3 (doppelte Vorzeichenumkehrung).



Die Schreibweise `--3` ist nicht zulässig, da `--` eine Dekrementierung bedeutet und diese nur für Variablen erlaubt ist. Jedoch ist die Formulierung `- -3` zulässig. Durch das Leerzeichen werden hier zwei Vorzeichenoperatoren und nicht die Dekrementierung erkannt.

4.12 Vergleichsoperatoren und logische Operatoren

Vergleichsoperatoren

Java besitzt für Wahrheitswerte den primitiven Datentyp `boolean`. Bisher haben Sie die Literale `true` und `false` kennengelernt, die Sie einer Variablen vom Typ `boolean` direkt zuordnen können.

```
boolean isValid = true;
boolean isOutOfRange = false;
```

Mithilfe von **Vergleichsoperatoren** können Sie Ausdrücke formulieren, die einen Wert vom Typ `boolean` liefern. Diese Ausdrücke lassen sich sprachlich entsprechend den folgenden Beispielen wie eine Behauptung formulieren, die entsprechend mit wahr (`true`) oder falsch (`false`) beurteilt werden kann:

- ✓ Ein Wert **gleich** einem anderen Wert!
- ✓ Ein Wert ist **größer als** ein anderer Wert!
- ✓ Der Wert eines Ausdrucks ist **kleiner als** der Wert eines anderen Ausdrucks!

In Java können alle mathematischen Vergleiche mithilfe einfacher Zeichen dargestellt werden. Vergleichsoperatoren können auf fast alle primären Datentypen angewendet werden. Ist ein Ausdruck nicht wahr, wird `false` geliefert, sonst `true`. Es werden 6 Vergleichsoperatoren unterschieden.

<code>==</code>	überprüft zwei Ausdrücke auf Gleichheit (alle primitiven Datentypen)
<code>!=</code>	überprüft zwei Ausdrücke auf Ungleichheit (alle primitiven Datentypen)
<code>></code>	liefert <code>true</code> , wenn der erste Operand größer als der zweite ist (alle außer <code>boolean</code>)
<code><</code>	liefert <code>true</code> , wenn der erste Operand kleiner als der zweite ist (alle außer <code>boolean</code>)
<code>>=</code>	liefert <code>true</code> , wenn der erste Operand größer oder gleich dem zweiten ist (alle außer <code>boolean</code>)
<code><=</code>	liefert <code>true</code> , wenn der erste Ausdruck kleiner oder gleich dem zweiten ist (alle außer <code>boolean</code>)

Beispiel: *CompareOperators.java*

```
int i = 10;
int j = 15;
boolean b = i > j; // b ist false
```

In Java ist es nicht möglich, den **Zuweisungsoperator** `=` mit dem **Vergleichsoperator** `==` zu verwechseln. Verwenden Sie in Ausdrücken versehentlich den Zuweisungsoperator, liefert der Compiler eine Fehlermeldung.

Logische Operatoren

Diese Operatoren dienen dazu, zwei Ausdrücke, die einen logischen Rückgabewert (`true`, `false`) liefern, entsprechend einer Vorschrift miteinander zu verknüpfen. Je nach Verknüpfungsvorschrift ist das Ergebnis ebenfalls `true` oder `false`. Es gibt vier logische Operatoren:

Verknüpfung	Syntax	Bedeutung
And (Und)	<code>expression1 && expression2</code>	Der UND-Operator <code>&&</code> ergibt nur dann <code>true</code> , wenn beide verknüpften Ausdrücke <code>true</code> liefern.
Or (Oder)	<code>expression1 expression2</code>	Der ODER-Operator <code> </code> ergibt <code>true</code> , wenn mindestens einer der verknüpften Ausdrücke <code>true</code> liefert.
Xor (exklusiv Oder)	<code>expression1 ^ expression2</code>	Der Exklusiv-ODER-Operator <code>^</code> ergibt <code>true</code> , wenn genau einer der verknüpften Ausdrücke <code>true</code> liefert.
Not (Negation)	<code>!expression</code>	Die Negation <code>!</code> wandelt das Ergebnis eines Ausdrucks in das Gegenteil um, z. B. <code>true</code> in <code>false</code> .

Die Operatoren `&&`, `||` und `^` sind links-bindend, während der Operator `!` wie der Vorzeichenoperator rechts-bindend ist. Das heißt, die Auswertung erfolgt von links nach rechts, sofern sie durch Klammern oder vorrangige Prioritäten der Operatoren nicht geändert wird.

Beispiel: *LogicalOperators.java*

```
int k = 0, i = 1;
boolean b = false;

b = (k > i) && (k >= 0); // b erhaelt den Wert false, da k nicht groesser
                        // als i ist
b = (k > i) || (i > k);  // b erhaelt den Wert true, da i groesser
                        // als k ist
b = !b;                // b erhaelt den Wert false, da b vorher true war
```



Beachten Sie, dass die Auswertung eines Ausdrucks, der logische Operatoren verwendet, abgebrochen wird, wenn sich am Gesamtergebnis nichts mehr ändern kann.

```
int k = 0, i = 1;
boolean b = false;

b = (k > i) && (k >= 0); //da k > i nicht zutrifft, kann auch der gesamte
                        //Ausdruck nicht wahr sein.
                        //Der zweite Ausdruck k >= 0 wird daher
                        // nicht mehr ausgewertet.
b = (i > k) || (k > i); //k > i wird nicht ausgewertet, da bereits i > k
                        // den Wert true liefert
```

4.13 Daten aus- und eingeben

Daten formatiert ausgeben

Eine einfache Datenausgabe erreichen Sie mit der Anweisung

```
System.out.println("...").
```

Im weiteren Verlauf dieses Buches wird zusätzlich die formatierte Ausgabe verwendet.

Syntax der formatierten Ausgabe

```
System.out.printf("text" [, expression1, expression2 ...]);
```

①

②

- ✓ Der in Anführungszeichen gesetzte Text `text` ① kann Formatierungszeichen enthalten. Die Tabelle zeigt eine Auswahl der möglichen Formatierungszeichen.
- ✓ Entsprechend der Anzahl der verwendeten Platzhalter `%d`, `%x` bzw. `%g` müssen entsprechende Variablen oder Werte ② (mit Komma abgetrennt) vorhanden sein.
- ✓ `%n` und `%%` sind keine Platzhalter, sondern dienen als Steuerzeichen.

<code>%d</code>	Platzhalter für einen ganzzahligen Wert
<code>%x</code>	Platzhalter für einen ganzzahligen Wert in hexadezimaler Schreibweise
<code>%g</code>	Platzhalter für einen Fließkomma-Wert
<code>%n</code>	bewirkt einen Zeilenumbruch
<code>%%</code>	gibt das Zeichen % selbst aus
Eine Auswahl der Formatierungszeichen	

Beispiel: *FormattedOutput.java*

```
int i = 213;
double x = 3.5;
System.out.printf("Eine Zahl: %d\nWert von i ist: %d\nWert von x ist: %g%n", 7, i, x);
```

- ✓ Zunächst wird der Text "Eine Zahl: " ausgegeben.
- ✓ Dann folgt der erste Ausdruck (7) als Ganzzahl (`%d`).
- ✓ Das Steuerzeichen `%n` veranlasst in der Ausgabe einen Zeilenumbruch.
- ✓ Auch in der zweiten Zeile werden Text und eine Ganzzahl (`i`) ausgegeben.
- ✓ In der dritten Zeile erfolgt die Ausgabe des Ausdrucks `x` als Fließkommazahl (`%g`). Standardmäßig erfolgt die Darstellung bei `%g` mit sechs Dezimalstellen (hier: eine Stelle vor und fünf Stellen hinter dem Dezimalpunkt).

```
Eine Zahl: 7
Wert von i ist: 213
Wert von x ist: 3.50000
```

Programmausgabe

Neben den hier beschriebenen stehen noch weitere Formatierungszeichen zur Verfügung. Außerdem lässt sich beispielsweise auch angeben, wie viele Dezimal- bzw. Nachkommastellen angezeigt werden sollen.

Daten als Programmparameter übergeben

Wenn Sie den Compiler starten, geben Sie als Parameter beispielsweise den Namen der Quelltextdatei an. Auf ähnliche Weise können Sie auch Ihrem Programm Parameter übergeben. Dies soll hier nur kurz erläutert werden, um Ihnen die Möglichkeit zu geben, ein Programm mit verschiedenen Werten testen zu können, ohne dass der Quelltext neu kompiliert werden muss.

Ein Parameter kann beispielsweise sein:

- ✓ ganzzahliger Wert vom Datentyp `int`
- ✓ Fließkommazahl vom Datentyp `double`

Einen Wert als Parameter übergeben

- ✓ Beim Starten des Programms geben Sie wie gewohnt den Interpreter `java` und anschließend den Programmnamen ein.
- ✓ Geben Sie nun hinter dem Programmnamen ein Leerzeichen und den gewünschten Wert ein. Verwenden Sie die bekannte Schreibweise für numerische Datentypen (Vorzeichen, Ziffern, Dezimalpunkt und gegebenenfalls die Exponentialschreibweise).



Beachten Sie, dass Sie bei Fließkommazahlen einen Dezimalpunkt und kein Komma verwenden.

Ein Texteditor wie beispielsweise TextPad, der speziell das Erstellen, Kompilieren und Ausführen von Java-Programmen unterstützt, bietet üblicherweise auch die Möglichkeit, Programmparameter über ein Dialogfenster einzugeben. Um TextPad so zu konfigurieren, rufen Sie den Menüpunkt KONFIGURATION - EINSTELLUNGEN auf und aktivieren Sie im Bereich *Extras - Java-Programm starten* das Kontrollfeld *Parameterabfrage*. Bei der Ausführung eines kompilierten Java-Programms wird dann automatisch ein Dialogfenster geöffnet. Tragen Sie den gewünschten Parameter hinter dem bereits bestehenden Eintrag ein. (Der bestehende Eintrag bezeichnet das auszuführende Java-Programm.)

Einen übergebenen Wert nutzen

Da Sie den Parameter hinter dem Programmnamen als Zeichen eingeben, muss der Parameter entsprechend umgewandelt werden, damit er als Wert vom Datentyp `int` bzw. `double` verwendet werden kann.

Auf die Schreibweise und die genaue Syntax soll hier nicht näher eingegangen werden, außerdem soll jeweils auch nur ein Parameter verwendet werden.

Parameter als ganze Zahl (int) verwenden	Syntax:	<code>identfier = Integer.parseInt(args[0]);</code>
	Beispiel: Die übergebene Zahl wird als Zahl vom Datentyp <code>int</code> ausgewertet ① und in der Variablen <code>number</code> gespeichert ②.	<pre> public static void main(String[] args) { int number; number = Integer.parseInt(args[0]); ... } </pre> <p>Diagramm: Ein Pfeil führt von ① zu <code>Integer.parseInt(args[0])</code>, ein weiterer von ② zu <code>number</code>.</p>
Parameter als Fließkommazahl (double) verwenden	Syntax:	<code>identfier = Double.parseDouble(args[0]);</code>
	Beispiel: Die übergebene Zahl wird als Zahl vom Datentyp <code>double</code> ausgewertet ③ und in der Variablen <code>height</code> gespeichert ④.	<pre> public static void main(String[] args) { double height; height = Double.parseDouble(args[0]); ... } </pre> <p>Diagramm: Ein Pfeil führt von ③ zu <code>Double.parseDouble(args[0])</code>, ein weiterer von ④ zu <code>height</code>.</p>

Dateneingaben im Programm entgegennehmen

Neben der Übergabe von Startparametern besteht die Möglichkeit, im Laufe der Programmabarbeitung Dateneingaben vom Anwender von der Konsole entgegenzunehmen. In den ersten Java-Versionen war dies etwas kompliziert. In den neueren Releases der Sprache stehen dafür spezielle Klassen bereit, die das Vorgehen vereinfachen:

- ✓ Die Klasse `Console` (seit Java 1.6) bietet eine einfache Möglichkeit der Ein- und Ausgabe über die Konsole. In Kapitel 15 wird diese Klasse vorgestellt.
- ✓ Die Klasse `Scanner` (seit Java 1.5) liest Daten nicht nur von der Konsole, sondern kann auch Zeichenketten und den Inhalt von Dateien auswerten. Zur Prüfung können dabei reguläre Ausdrücke verwendet werden. Ein Beispiel sehen Sie im Folgenden.

Der folgende Quelltext zur Demonstration der Eingabemöglichkeit mittels der Klasse `Scanner` enthält einige Komponenten, welche bisher noch nicht behandelt wurden. Den Umgang mit Klassen und Methoden, ihre Einbindung ins Programm und die Behandlung von Ausnahmen sowie die dafür notwendige Syntax lernen Sie in den folgenden Kapiteln.

Beispiel: *UseScanner.java*

```

① import java.util.*;

public class UseScanner
{
    public static void main(String args[])
    {
②        try
        {
③            Scanner sc = new Scanner(System.in);
④            System.out.print("Geben Sie einen ganzzahligen Wert ein: ");
⑤            int i = sc.nextInt();
        }
    }
}

```

```
⑥      System.out.printf("Es wurde die Zahl %d eingegeben", i);
      }
⑦      catch (InputMismatchException ex)
      {
          System.out.println
              ("Es wurde kein ganzzahliger Wert eingeben.");
      }
  }
}
```

Dateneingabe mit der Klasse Scanner

- ① Über die `import`-Vereinbarung importieren Sie die Klassen des Package `util`.
- ② Da das Programm die Eingabe einer ganzzahligen Zahl erwartet, würde bei der Eingabe eines Textes oder einer Fließkommazahl eine sogenannte Ausnahme (eine Exception) auftreten. Um diese abzufangen und zu behandeln, wird ein `try-catch`-Block verwendet.
- ③ Ein Objekt der Klasse `Scanner` wird erstellt. Als Quelle wird ihm die Konsoleneingabe zugeordnet.
- ④ Die Ausgabe der Eingabeaufforderung.
- ⑤ Mit der Methode `nextInt()` wird die Konsoleneingabe des Anwenders der Integer-Variablen `i` zugewiesen. Wurde kein ganzzahliger Wert eingegeben, löst diese Anweisung eine Ausnahme aus.
- ⑥ Der eingegebene ganzzahlige Wert wird mit etwas Begleittext auf der Konsole wieder ausgegeben.
- ⑦ Wurde eine Ausnahme ausgelöst, erfolgt hier deren Behandlung durch Ausgabe eines entsprechenden Hinweises.

```
C:\uebung\jav9\kap04>java UseScanner
Geben Sie einen ganzzahligen Wert ein: 12
Es wurde die Zahl 12 eingegeben
C:\uebung\jav9\kap04>java UseScanner
Geben Sie einen ganzzahligen Wert ein: 1.1
Es wurde kein ganzzahliger Wert eingeben.


C:\uebung\jav9\kap04>java UseScanner
Geben Sie einen ganzzahligen Wert ein: Test
Es wurde kein ganzzahliger Wert eingeben.

C:\uebung\jav9\kap04>
```

Ein- und Ausgaben im Programm UseScanner

4.14 Übung

Variable und Konstante nutzen

Level		Zeit	ca. 20 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Deklaration von Variablen ✓ Verwendung von Konstanten ✓ Nutzung von Inkrement und Dekrement ✓ Formatierte Ausgabe 		
Übungsdatei	--		
Ergebnisdateien	<i>Exercise1.java, Exercise2.java, Exercise3.java, Exercise4.java, Exercise5.java</i>		

1. Berichtigen Sie die Fehler in dem folgenden Programmausschnitt.

```
public static void main(String[] args)
{
    int a b;
    a = b = 10;
    System.out.println("Beide Zahlen haben jetzt den Wert 10);
}
```

2. Schreiben Sie ein Programm, welches den Konstanten `NUMBER1` und `NUMBER2` die Werte 12 und 4 zuordnet. Anschließend sollen die Summe, das Produkt, die Differenz und der Quotient dieser Konstanten berechnet und in geeigneten Variablen gespeichert werden. Geben Sie die jeweiligen Ergebnisse zur Kontrolle aus.
3. Geben Sie drei verschiedene Möglichkeiten an, den Wert 1 zu einer Variablen `x` vom Typ `int` zu addieren.
4. Welchen Wert liefern die folgenden Ausdrücke? Jeder Ausdruck übernimmt dabei die neuen Werte für `d` und `e`.

```
int d = 1, e = 2;
d *= e;
d += e++;
d -= 3 - 2 * e;
e /= (d + 1);
```

5. Schreiben Sie ein Programm, das die Anweisungen aus Aufgabenteil 4 enthält und nach jeder dieser Anweisungen die Namen und Werte der Variablen `d` und `e` ausgibt.

5

Kontrollstrukturen

5.1 Kontrollstrukturen einsetzen

Kontrollstrukturen im Überblick

Die Programme, die Sie bisher kennengelernt haben, bestanden aus einer Folge von Anweisungen, die genau einmal **sequenziell** (der Reihe nach) abgearbeitet wurden. Oft ist es jedoch auch erforderlich, dass Programmteile mehrmals oder gar nicht ausgeführt werden. Die Java-Sprachelemente, mit denen der Programmablauf gesteuert werden kann, werden **Kontrollstrukturen** genannt. Die Entscheidung, nach welchen Kriterien der Ablauf gesteuert wird, wird in **Bedingungen** (engl. conditions) formuliert.

Es werden zwei Gruppen von Kontrollstrukturen unterschieden:

Verzweigungen (engl. conditional statements)	Es werden alternative Programmteile angeboten, in die – abhängig von einer Bedingung – beim Programmablauf verzweigt wird.
Schleifen (engl. loops)	Ein Programmteil kann – abhängig von einer Bedingung – mehrmals durchlaufen werden.

Verzweigungen

Zwei Grundformen der Verzweigungen sind in Java enthalten:

if-Anweisung	Je nachdem, ob eine Bedingung erfüllt ist oder nicht, wird ein Programmteil ausgeführt oder übersprungen bzw. ein alternativer Programmteil ausgeführt.
switch-Anweisung	Es wird ein Ausdruck ausgewertet. Je nachdem, welchem Wert der Ausdruck entspricht, verzweigt das Programm fallweise (case) in einen entsprechenden Programmteil. Eine <code>switch</code> -Anweisung kann somit sehr viele Alternativen bereitstellen.

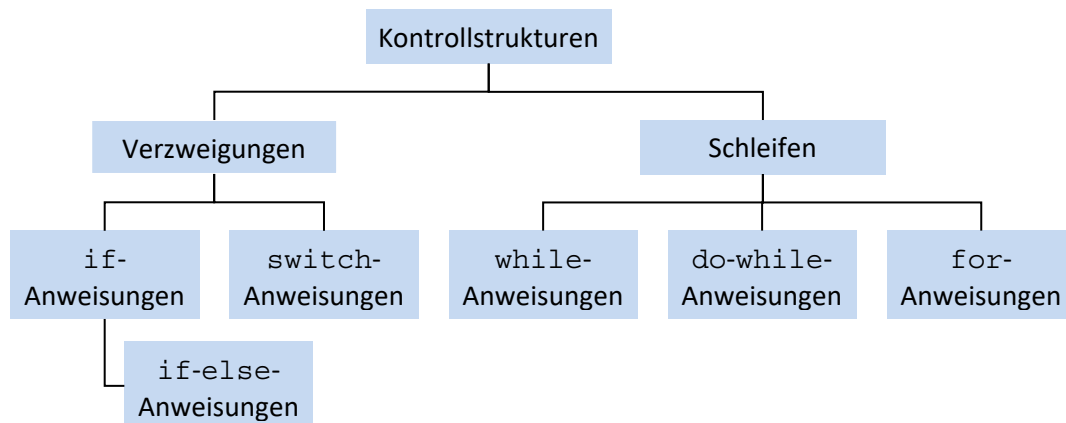
Schleifen

Über Schleifen können Anweisungen und Anweisungsblöcke mehrfach ausgeführt werden.

Drei Arten von Schleifenstrukturen sind in Java enthalten:

while-Anweisung	Solange eine Bedingung erfüllt ist, wird der in der Struktur eingeschlossene Programmteil ausgeführt. Anschließend wird das Programm hinter der <code>while</code> -Anweisung fortgesetzt. Die Überprüfung der Bedingung erfolgt vor der Ausführung des eingeschlossenen Programmteils.
do-while-Anweisung	Ein Programmteil wird ausgeführt. Anschließend wird die Bedingung geprüft. Ist die Bedingung erfüllt, wird der Programmteil erneut ausgeführt. Erst wenn die Bedingung nicht mehr erfüllt ist, wird das Programm hinter der <code>do-while</code> -Anweisung fortgesetzt.
for-Anweisung	Bei der <code>for</code> -Anweisung wird die Schleife über einen Zähler gesteuert. Die Überprüfung der Bedingung erfolgt vor der Ausführung des eingeschlossenen Programmteils. Mit der foreach -Schleife existiert für die einfache Auswertung von Arrays und Collections eine besondere Ausprägung der for -Schleife. Sie ermöglicht das Durchlaufen aller Elemente der Strukturen unter Anwendung einer vereinfachten Schreibweise.

Übersicht über die Kontrollstrukturen in Java

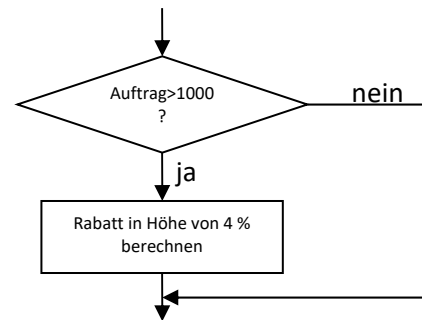


5.2 if-Anweisung

Bei vielen Problemstellungen ist die Verarbeitung von Anweisungen von Bedingungen abhängig. Nur wenn die Bedingung erfüllt ist, wird die betreffende Anweisung (oder der Anweisungsblock) ausgeführt. Andernfalls wird die Anweisung übersprungen. Für die Steuerung eines solchen Programmablaufs stellt Java die **einseitige** `if`-Anweisung zur Verfügung.

Beispiel für die Verwendung einer `if`-Anweisung

- ✓ Wenn (`if`) ein Kunde einen Auftrag über 1000,- EUR erteilt, bekommt er 4 % Rabatt. Bei Aufträgen bis 1000,- EUR wird kein Rabatt gewährt, d. h., die Berechnung des Rabatts wird nicht ausgeführt.
- ✓ Die Überprüfung der Bedingung kann nur die beiden Ergebnisse "ja" oder "nein" ergeben, in Java entsprechend `true` oder `false`.



Syntax der einseitigen `if`-Anweisung

- ✓ Die `if`-Anweisung beginnt mit dem Schlüsselwort `if`.
- ✓ Dahinter wird eine Bedingung (Condition) formuliert und in runde Klammern `()` eingeschlossen. Die Bedingung ist ein Ausdruck (Expression) und liefert als Ergebnis einen Wert vom Typ `boolean` zurück.
- ✓ Ergibt die Auswertung der Bedingung den Wert `true` (wahr), wird die folgende Anweisung ausgeführt. Liefert die Bedingung den Wert `false` (falsch), wird die Anweisung ① übersprungen.
- ✓ Statement kann eine einfache Anweisung oder ein Anweisungsblock sein.

```
if (condition)
    statement ①
```

Beispiel: *Discount.java*

Für einen Rechnungsbetrag (`invoiceAmount`) soll abhängig von seiner Höhe ein Rabatt berechnet werden. Anhand des Wertes der Variablen `invoiceAmount` wird entschieden, ob ein Rabatt gewährt wird. Dieser wird dann gegebenenfalls berechnet und vom Rechnungsbetrag subtrahiert. Anschließend wird der neue Rechnungsbetrag (abzüglich des Rabatts) ausgegeben. Die Währungsangaben werden hier nicht weiter berücksichtigt.

```

class Discount
{
    public static void main(String[] args)
    {
        ① double invoiceAmount = 0.0;           //Rechnungsbetrag
           //Programmparameter als Fließkommazahl auswerten und in
           //invoiceAmount speichern:
        ② invoiceAmount = Double.parseDouble(args[0]);
        ③ if (invoiceAmount > 1000)
        {
            ④ // Rabatt wird berechnet und vom Rechnungsbetrag abgezogen
               invoiceAmount -= invoiceAmount * 0.04;
               /*invoiceAmount *= 0.96; ist alternative Form */
               System.out.println("Es wurde ein Rabatt gewaehrt");
        }
        ⑤ System.out.printf
           ("Ihr Gesamtbetrag betraegt %g%n", invoiceAmount);
    }
}
    
```

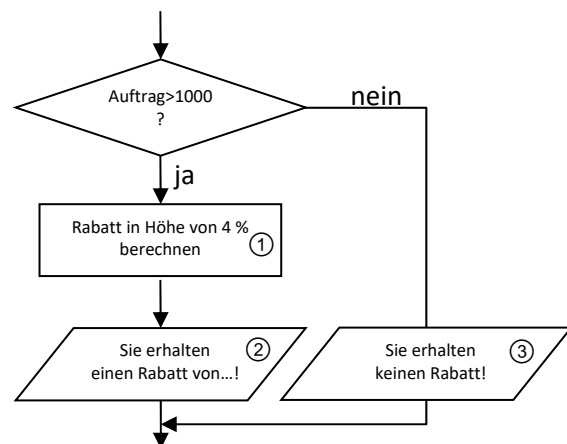
- ① Die Variable `invoiceAmount` wird definiert und initialisiert.
- ② Der beim Programmaufruf übergebene Wert wird als Fließkommazahl betrachtet, ausgewertet und als Rechnungsbetrag (`invoiceAmount`) verwendet.
- ③ Es wird überprüft, ob der Wert der Variablen `invoiceAmount` über 1000 liegt.
- ④ Ist der Wert von `invoiceAmount` größer als 1000, wird der Rabatt in Höhe von 4 % berechnet und ein entsprechender Text als Information ausgegeben.
- ⑤ Der neu berechnete Rechnungsbetrag wird ausgegeben. Wenn kein Rabatt gewährt wird, wird der Anweisungsblock ④ übersprungen und der Rechnungsbetrag unverändert ausgegeben ⑤.

5.3 `if-else`-Anweisung

Bei der `if-else`-Anweisung werden bestimmte Anweisungen durchgeführt, falls die Bedingung erfüllt ist. Falls die Bedingung nicht erfüllt ist, werden andere Anweisungen durchgeführt. Da hier bei beiden Fällen alternative Anwendungen ausgeführt werden, wird die `if-else`-Anweisung auch als **zweiseitige if-Anweisung** bezeichnet.

Beispiel für die Verwendung der `if-else`-Anweisung

- ✓ Wenn (`if`) ein Kunde einen Auftrag über 1000,- EUR erteilt, bekommt er 4 % Rabatt ①. Eine Meldung informiert darüber ②.
- ✓ Anderenfalls (`else`), d. h. bei Aufträgen bis 1000,- EUR, wird kein Rabatt gewährt und eine entsprechende Meldung ③ ausgegeben.



Syntax der `if-else`-Anweisung

- ✓ Nach der Bedingung in der `if`-Anweisung und der Anweisung ① folgt jetzt das Schlüsselwort `else`.
- ✓ Durch ein optionales `else if` mit einer weiteren Bedingung kann eine Verzweigungskette aufgebaut werden ②.
- ✓ Nach `else` schließt sich die Anweisung an, die ausgeführt wird, wenn die Bedingung in der `if`-Anweisung nicht zutrifft (Ausdruck liefert `false`) ③.
- ✓ Statement kann jeweils eine einfache Anweisung oder ein Anweisungsblock sein.

```

if (condition) ①
    statement1
else if (condition) ②
    statement2
else ③
    statement3
  
```


Geschachtelte if-Anweisungen

Zu welcher if-Anweisung gehört die **else**-Alternative?

<pre> if (condition) if (condition2) ① statement1 else ② statement2 </pre>	<pre> if (condition) ④ { if (condition2) statement1 } else ③ statement2 </pre>
<p>✓ Die mit else eingeleitete Alternative ② gehört zur jeweils vorherigen if-Anweisung ①.</p>	<p>✓ Sollte der else-Zweig ③ zu einer weiter entfernten if-Anweisung ④ gehören, müssen Sie dies durch geschweifte Klammern ⑤ angeben.</p>

Tipps für die Formulierung und Formatierung des Quelltextes

- ✓ Durch entsprechendes Einrücken gestalten Sie den Quelltext übersichtlicher.
- ✓ Zur Vermeidung von Fehlern empfiehlt es sich, geschweifte Klammern `{ }` einzufügen, auch wenn innerhalb der **if**-Anweisung nur eine einzige Anweisung ausgeführt wird. Die Klammern sind syntaktisch nicht erforderlich, verbessern aber die Lesbarkeit.

```

if (condition)
{
    statement1
}
...

```

Beispiel: *Discount2.java*

Für einen Rechnungsbetrag (`invoiceAmount`) soll abhängig von seiner Höhe ein Rabatt (`discountAmount`) berechnet werden. Wird ein Rabatt gewährt, wird dieser berechnet und der Rabattbetrag ausgegeben. Sofern kein Rabatt gewährt wird, erfolgt die Ausgabe einer entsprechenden Information.

```

class Discount2
{
    public static void main(String[] args)
    {
        ① double invoiceAmount = 0.0;           //Rechnungsbetrag
        final double DISCOUNT_RATE = 0.05;   //Rabatt: als Konstante
        ② double discountAmount = 0.0;         //Rabattbetrag
        ③ invoiceAmount = Double.parseDouble(args[0]);
        ④ if (invoiceAmount > 1000)
        {
            ⑤ discountAmount = invoiceAmount * DISCOUNT_RATE;
            System.out.printf
                ("Sie erhalten einen Rabatt von %g%n", discountAmount);
        }
    }
}

```

```

⑥      else
        {
            System.out.println
                ("Bei Werten ueber 1000 erhalten Sie Rabatt!");
        }
    }
}

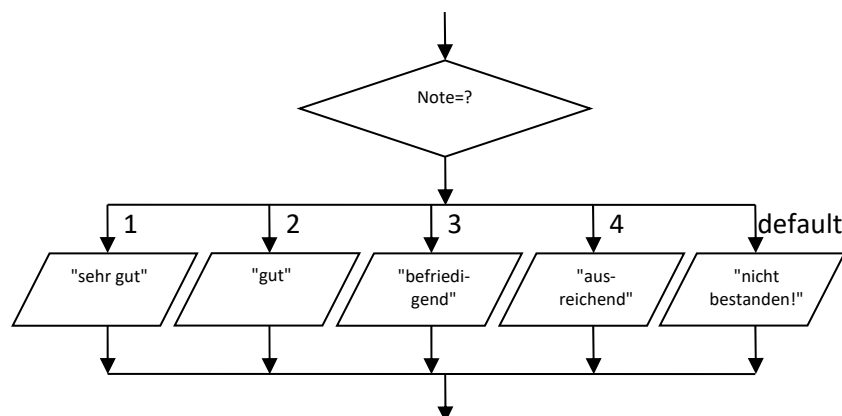
```

- ① Die Variablen (bzw. die Konstante) werden definiert und initialisiert.
- ② Es wird ein Rabatt bei einem Rechnungsbetrag über 1000 gewährt.
- ③ Der Programmparameter wird als Fließkommazahl ausgewertet und in `invoiceAmount` gespeichert.
- ④ In diesem Fall wird der Rabatt berechnet und ausgegeben.
- ⑤ Diese geschweiften Klammern könnten entfallen, da nur eine einzige Anweisung unter `else` ausgeführt wird.
- ⑥ Ist der eingegebene Wert kleiner oder gleich 1000, wird kein Rabatt gewährt. Sie erhalten jedoch eine Meldung, wann Sie Rabatt erhalten.

5.4 switch-Anweisung

Die `if-else`-Anwendung ermöglicht es, in Abhängigkeit von einer Bedingung lediglich zwei alternative Programmteile auszuführen, da die Bedingung als logischer Ausdruck nur die beiden Werte `true` oder `false` ergeben kann.

Mit einer `switch`-Anweisung testen Sie den Wert einer Variablen oder eines komplexen Ausdrucks. Diese Variable bzw. dieser Ausdruck wird **Selektor** genannt. In Abhängigkeit von dem Wert des Selektors können fallweise (`case`) verschiedene Anweisungen durchgeführt werden. Der Wert des Selektors muss einem der primitiven Datentypen `char`, `byte`, `short` und `int` oder dem Referenztyp `String` entsprechen. Die Verwendung von Zeichenketten (Referenztyp `String`) in `Switch`-Statements besteht seit der Version 7 von Java. Durch die Vielzahl der Werte, die diese Datentypen bereitstellen, kann in einer einzigen `switch`-Anweisung eine große Anzahl von Verzweigungen angeboten werden. Daher wird die `switch`-Anweisung auch als **mehrseitige** Auswahl bezeichnet.



Syntax der `switch`-Anweisung

- ✓ Das Schlüsselwort `switch` leitet die **mehrseitige** Auswahl (Verzweigung) ein.
- ✓ Danach folgt in Klammern `[]` der **Selektor**, der ein Ausdruck oder eine Variable sein kann und dessen Wert vom Datentyp `char`, `byte`, `short`, `int` oder dem Referenztyp `String` sein muss. Der Wert des Selektors bestimmt, welche Anweisung als Nächstes ausgeführt wird.
- ✓ Auf den Selektor folgt ein Anweisungsblock ①, der sogenannte `case`-Anweisungen einschließt.
- ✓ Für jeden zu betrachtenden Wert, den der Selektor annehmen kann, wird eine `case`-Anweisung formuliert. Eine `case`-Anweisung beginnt mit dem Schlüsselwort `case` (Fall), dem die Angabe des entsprechenden Wertes folgt.
Dieser Auswahlwert kann als Literal (beispielsweise als Zahlenwert, Zeichen oder Zeichenkette) oder als Konstante angegeben werden.
- ✓ Stimmt der Wert des Selektors mit einem der aufgeführten Auswahlwerte überein, dann wird das Programm mit der Anweisung unmittelbar hinter der entsprechenden `case`-Anweisung fortgesetzt. Mit einer `break`-Anweisung wird die `switch`-Anweisung verlassen und das Programm hinter der `switch`-Anweisung fortgesetzt.
- ✓ Die Auswahlwerte dürfen nicht mehrfach geprüft werden, d. h., es darf kein Wert mehr als einmal in den Auswahlwerten der `case`-Anweisung vorkommen.
- ✓ Stimmt der Wert des Selektors mit keinem Auswahlwert überein, werden die Anweisungen hinter der `default`-Anweisung ausgeführt. Wenn keine `default`-Anweisung vorhanden ist, wird das Programm hinter der `switch`-Anweisung (①) fortgesetzt.

```
switch (expression)
{
    case constantValue1: statement1;
                        break;
    case constantValue2: statement2; ①
                        break;
    ...
    [default: statement;]
}
```

Wird keine `break`-Anweisung angegeben, werden die Anweisungsblöcke von nachfolgenden `case`-Anweisungen ebenfalls ausgeführt, ohne dass eine erneute Prüfung des Selektors stattfindet.

Beispiel: `ControlSwitch.java`

Das Programm wird über den Wert der Variablen `number` gesteuert. Der Wert der Variablen wird in einer `switch`-Anweisung untersucht. Es wird die Teilbarkeit der Variablen `number` durch 7 geprüft. Ist die Zahl durch 7 teilbar, erfolgt eine entsprechende Ausgabe. Die Fälle, in denen der Divisionsrest 1 bzw. 2 ist, werden ebenfalls unterschieden. Alle anderen Fälle, d. h., der Divisionsrest ist größer als 2, werden zusammengefasst und nicht detailliert betrachtet.

```
class ControlSwitch
{
    public static void main(String[] args)
    {
        ①    int result = 0;
            int divisor = 7;
            int number = 0;
```

```

② number = Integer.parseInt(args[0]); //Zahl einlesen
  if (number > 0)
  {
③     result = number % divisor;           //ganzzahligen Divisions-
                                           // rest berechnen
④     switch(result)
      {
⑤         case 0:
            System.out.printf
              ("Die Zahl ist durch %d teilbar.%n", divisor);
            break;
            case 1:
            System.out.println("Der Divisionsrest betraegt 1.");
            break;
            case 2:
            System.out.println("Der Divisionsrest betraegt 2.");
            break;
⑥         default:
            System.out.println("Divisionsrest ist groesser als 2.");
      }
    }
  }
}

```

- ① Die Variablen werden definiert und initialisiert.
- ② Der übergebene Parameter wird als Zahl interpretiert und in der Variablen `number` gespeichert.
- ③ Aus der Zahl und dem vorgegebenen `divisor` wird der Divisionsrest berechnet und in der Variablen `result` gespeichert.
- ④ Über die `switch`-Anweisung wird der berechnete Wert (`result`) ausgewertet.
- ⑤ Bei dem Wert 0 ist die Zahl ohne Divisionsrest teilbar. Je nach Divisionsrest erfolgt eine entsprechende Ausgabe. Die `break`-Anweisungen verhindern jeweils, dass auch die nachfolgenden Ausgabeanweisungen ausgeführt werden.
- ⑥ Falls der Divisionsrest größer als 2 ist, wird die Anweisung nach `default` ausgeführt.

5.5 Schleifen

Schleifen verwenden

Um einen bestimmten Teil des Programms mehrfach auszuführen bzw. zu wiederholen, werden Schleifen verwendet:

- ✓ Sie können variabel (auch erst zur Laufzeit des Programms) festlegen, wie oft oder bis zum Eintreffen welcher Bedingung die Schleife durchlaufen werden soll.
- ✓ Durch die Verwendung von Schleifen sparen Sie Programmcode.

Aufbau einer Schleife

Eine Schleife besteht aus der **Schleifensteuerung** und dem **Schleifenrumpf**. Der Schleifenrumpf umfasst den Programmteil, der wiederholt werden soll. Die Schleifensteuerung dient dazu, festzulegen, wie oft die Anweisungen im Schleifenrumpf wiederholt werden sollen bzw. nach welchen Kriterien entschieden wird, ob eine Wiederholung erfolgen soll.

Es werden zwei Steuerungsarten unterschieden:

Kopfgesteuerte Schleife	Fußgesteuerte Schleife
<ul style="list-style-type: none"> ✓ Die Prüfung, ob die Anweisungen im Schleifenrumpf ausgeführt werden sollen, erfolgt gleich zu Beginn. ✓ Ist das Kriterium erfüllt, wird der Schleifenrumpf durchlaufen und anschließend erfolgt eine erneute Prüfung. ✓ Falls das Kriterium zu Beginn bereits nicht erfüllt ist, wird Schleifenrumpf gar nicht ausgeführt. 	<ul style="list-style-type: none"> ✓ Zuerst werden die Anweisungen im Schleifenrumpf ausgeführt. ✓ Dann erfolgt die Prüfung, ob ein weiterer Durchlauf erfolgen soll. ✓ Der Schleifenrumpf wird also immer mindestens einmal ausgeführt.

5.6 while-Anweisung

Bei der `while`-Anweisung (solange ...) handelt es sich um eine **kopfgesteuerte** Schleifenstruktur. Die Ausführung der Anweisungen im Schleifenrumpf ist von der Gültigkeit einer Bedingung abhängig, die gleich zu **Beginn** der Anweisung überprüft wird. Solange die Bedingung erfüllt ist, werden die folgenden Anweisungen (Schleifenrumpf) ausgeführt. Anschließend wird die Bedingung erneut geprüft. Ist die Bedingung nicht erfüllt, wird das Programm hinter der `while`-Anweisung fortgesetzt.

Beispiel

Solange eine Zahl kleiner als 100 ist, soll zu einer Zahl der Wert 10 addiert werden: Es wird geprüft, ob die Zahl kleiner als 100 ist. Ist die Zahl kleiner als 100, wird zu der Zahl der Wert 10 addiert. Anschließend erfolgt erneut die Prüfung der Bedingung.

Syntax der **while**-Anweisung

- ✓ Das Schlüsselwort **while** leitet die **while**-Anweisung ein.
- ✓ Es folgt in runden Klammern `()` die Formulierung der Bedingung, die einen Wert vom Typ `boolean` liefern muss.
- ✓ Anschließend folgt der Schleifenrumpf mit der Anweisung oder einem Anweisungsblock (`statement`).
- ✓ Ist die Bedingung erfüllt (`true`), wird die Anweisung (bzw. der Anweisungsblock) ausgeführt. Anschließend erfolgt eine erneute Prüfung die Bedingung. Solange die Bedingung erfüllt ist, wird die Ausführung der Anweisung(en) wiederholt.
- ✓ Ist die Bedingung nicht erfüllt (`false`), wird der Schleifenkörper übersprungen und das Programm nach der **while**-Anweisung fortgesetzt.

while (condition) statement

! Achten Sie bei der Arbeit mit Schleifen immer darauf, dass Sie das Abbruchkriterium so festlegen, dass es auch **mit Sicherheit eintritt**. Andernfalls wird der Schleifenkörper unendlich oft ausgeführt (**Endlosschleife**) oder so lange, bis ein Fehler bei der Zuweisung entsteht. Dies kann z. B. der Fall sein, wenn Sie zu einer Zahl einen konstanten Wert addieren. Bei unendlicher Ausführung wird der Wertebereich dieser Zahl überschritten.

Beispiel: *ControlWhile.java*

```
class ControlWhile
{
    public static void main(String[] args)
    {
        ① int counter = 1;
        ② while (counter < 10)
        {
            ④ System.out.println(counter); ③
            ⑤ counter += 2;
        }
    }
}
```

- ① Der Startwert der Schleife wird mit 1 festgelegt.
- ② Die Bedingung der Schleife legt fest, dass der Schleifenrumpf ③ ausgeführt wird, solange der Wert der Variablen `counter` kleiner als 10 ist.
- ④ Der Wert der Variablen `counter` wird ausgegeben.
- ⑤ Der Zähler (die Variable `counter`) wird um 2 erhöht, anschließend wird wieder Schritt ② ausgeführt.

5.7 do-while-Anweisung

Die do-while-Anweisung arbeitet **fußgesteuert**. Die Bedingung wird erst **nach** dem Durchlauf des Schleifenrumpfes ausgewertet. Dadurch wird der Schleifenkörper mindestens einmal ausgeführt. Ist die Bedingung erfüllt (`true`), wird der Schleifenrumpf erneut durchlaufen. Ist die Bedingung nicht erfüllt, wird die Ausführung der Schleife beendet und das Programm hinter der while-Schleife fortgesetzt.

Beispiel

Zu einer Zahl wird mindestens einmal der Wert 10 addiert. Ist die Zahl dann immer noch kleiner als 100, addieren Sie erneut den Wert 10. Brechen Sie ab, wenn die Zahl größer oder gleich 100 ist.

Syntax der do-while-Anweisung

- ✓ Das Schlüsselwort `do` leitet die do-while-Anweisung ein.
- ✓ Danach folgt der Schleifenrumpf (`statement`). Statement kann eine einzelne Anweisung oder ein Anweisungsblock sein.
- ✓ Diese werden im Gegensatz zur `while`-Anweisung mindestens einmal ausgeführt, da die Auswertung des Ausdrucks erst am Ende der Schleife erfolgt.
- ✓ Nach dem Schleifenrumpf folgt zur Einleitung der Schleifensteuerung das Schlüsselwort `while`.
- ✓ In runde Klammern `()` eingeschlossen folgt dann die Bedingung, die als Ausdruck einen Wert vom Typ `boolean` liefern muss.
- ✓ Ist die Bedingung erfüllt (`true`), wird der Schleifenkörper erneut ausgeführt. Ist die Bedingung nicht erfüllt (`false`), wird die do-while-Anweisung beendet und das Programm fortgesetzt.
- ✓ Die do-while-Anweisung wird mit einem Semikolon abgeschlossen.

```
do
    statement
while (condition);
```

Beispiel: *ControlDoWhile.java*

Das Programm berechnet die Anzahl der Jahre, die benötigt werden, um von einem Startkapital von 1000,- EUR nur über die Zinsen einen Endbetrag von 10000,- EUR zu erreichen.

```
class ControlDoWhile
{
    public static void main(String[] args)
    {
        ① double presentValue = 1000.0;
        double futureValue = 10000.0;
        ② final double INTEREST_RATE = 4.5; //Zinssatz in Prozent
        ③ int year = 0;

        do
        {
```

```

④    //Startwert wird um Zinsen erhoeht:
    presentValue = presentValue * (1.0 + INTEREST_RATE / 100);
⑤    year++;                                //Jahr um 1 erhoehen
    }
⑥    while                                //Das gewuenschte Kapital ist
    (presentValue < futureValue); //noch nicht erreicht, richtig?

⑦    System.out.printf("Die Dauer betraegt %d Jahre%n",year);
    }
}

```

- ① Diese Variablen enthalten die Werte für das Startkapital (`presentValue`) und das zu erreichende Kapital (`futureValue`). Da für die Zinsberechnung Kommastellen notwendig sind, werden Gleitkommazahlen (`double`) verwendet.
- ② Der Zinssatz wird als Konstante aus 4,5 % festgelegt.
- ③ Die Variable `year` enthält als Zähler die Anzahl der Jahre, die vergehen, bis das gewünschte Kapital erreicht ist.
- ④ In der `do-while`-Schleife werden nun die Zinsen berechnet und zum aktuellen Kapital addiert.
- ⑤ Der Jahres-Zähler (`year`) wird über den Inkrement-Operator (`++`) jeweils um 1 erhöht.
- ⑥ Solange das gewünschte Kapital nicht erreicht wurde, wird der Schleifenkörper erneut ausgeführt.
- ⑦ Ist das Kapital erreicht, wird die Anzahl der Jahre, die zum Erreichen des Endbetrags notwendig waren, mit `System.out.printf` ausgegeben.

5.8 for-Anweisung

Die `for`-Anweisung ist eine **kopfgesteuerte** Schleife ähnlich der `while`-Anweisung. Jede `for`-Anweisung lässt sich auch als `while`-Anweisung formulieren. Die `for`-Anweisung zeichnet sich durch eine kompakte Schreibweise aus und wird häufig verwendet, wenn die Anzahl der Schleifendurchläufe zuvor bereits bekannt ist. Der Schleifenrumpf wird mit einem Zähler in der gewünschten Anzahl wiederholt ausgeführt.

Viele Problemstellungen müssen durch eine wiederholte Durchführung von bestimmten Anweisungen gelöst werden. Dabei kann die Anzahl der Wiederholungen von Parametern oder Bedingungen abhängig gemacht werden.

Syntax der `for`-Anweisung

- ✓ Das Schlüsselwort `for` leitet die `for`-Anweisung ein.

`for (initStatement; condition; nextStatement)
 statement`
- ✓ Anschließend folgt die Schleifensteuerung, die in runde Klammern `()` eingeschlossen wird.

- ✓ Zur Schleifensteuerung gehören der Initialisierungsteil (`initStatement`), der Bedingungs-
ausdruck (`condition`) und der Aktualisierungsteil (`nextStatement`). Diese drei Teile
werden jeweils durch ein Semikolon getrennt.
- ✓ Im Initialisierungsteil (`initStatement`) wird eine Variable, die als Zähler dienen soll,
definiert und initialisiert.
- ✓ Im Bedingungsteil (`condition`) wird die Bedingung (ein Ausdruck vom Datentyp
`boolean`) entsprechend einer `while`-Anweisung formuliert.
- ✓ Solange die Bedingung den Wert `true` liefert, wird der Schleifenrumpf ausgeführt.
- ✓ Liefert die Bedingung den Wert `false`, wird die `for`-Anweisung beendet und das
Programm hinter der `for`-Anweisung fortgesetzt.
- ✓ Im Aktualisierungsteil kann der Wert von Variablen, meist der Wert der Zählvariablen,
geändert werden. Sie können dazu z. B. die Operatoren `++`, `--`, `+=`, `-=`, `*=` verwenden.
Die im Aktualisierungsteil aufgeführte Anweisung wird für jeden Schleifendurchlauf einmal
ausgeführt.

Statt im Aktualisierungsteil können Sie die Werte der Variablen auch in der Schleife selbst
ändern.



Eine Variable, die Sie beispielsweise im Initialisierungsteil oder im Aktualisierungsteil definie-
ren, ist nur im aktuellen Anweisungsblock, d. h. innerhalb der Schleifenstruktur, gültig.

Beispiel: *ControlFor.java*

Mit diesem Programm können Sie die Fakultät einer natürlichen Zahl im Bereich zwischen 1 und
15 berechnen. Zum Beispiel ist die Fakultät der Zahl 4 gleich 24 (Berechnung: $4! = 1 * 2 * 3 * 4$).

Achten Sie darauf, dass die Fakultät auch bei kleinen Zahlen sehr schnell wächst, sodass hier
die Berechnung der Fakultät auf Werte zwischen 1 und 15 beschränkt werden soll.

```
class ControlFor
{
    public static void main(String[] args)
    {
        ① int result = 1;
        ② int number = 0;

        ③ number = Integer.parseInt(args[0]); // Zahl einlesen

        ④ if ((number >= 1) && (number <= 15)) //zulaessiger Bereich 1-15
        {
            ⑤ for (int i = 1; i <= number; i++)
            {
                ⑥ result = result * i; //oder kurz: result *= i;
            }
        }
    }
}
```

```

⑦      System.out.printf
        ("Die Fakultaet von %d ist %d%n", number, result);
    }
    else
    {
⑧      System.out.println
        ("Zahl lag ausserhalb des zulaessigen Bereichs.");
    }
}
}

```

- ① Die Variable `result` speichert die berechnete Fakultät.
- ② Die Variable `number` speichert den Zahlenwert, für den die Fakultät berechnet werden soll.
- ③ Der übergebene Parameter wird als Zahl vom Datentyp `int` interpretiert und der Variablen `number` zugewiesen.
- ④ Für Zahlen kleiner als 1, für die die Fakultät nicht berechnet werden kann (per Definition ist die Fakultät von 0 1, wird im Beispiel aber nicht berücksichtigt), und für Zahlen größer als 15 erfolgt eine entsprechende Ausgabe ⑧ und das Programm wird beendet.
- ⑤ Innerhalb der `for`-Anweisung wird die Variable `i` definiert und mit dem Wert 1 initialisiert. Solange der Zähler `i` kleiner oder gleich der Zahl `number` ist, soll die Schleife ausgeführt werden. Die Variable `i` wird nach jedem Schleifendurchlauf um den Wert 1 inkrementiert (erhöht).
- ⑥ Die Fakultät wird berechnet. Durch die Wiederholungen des Schleifenrumpfs entsteht eine Formel der Art:

$$\text{result} = 1 * 2 * 3 * \dots * \text{number}$$
- ⑦ Das Ergebnis der Berechnung wird ausgegeben.

Tipps und häufige Verwendungsformen der `for`-Anweisung

- ✓ Dies ist eine typische Anwendung der `for`-Anweisung.

```
for (int i = 0; i < 10; i++)
```

Im ersten Schleifendurchlauf besitzt `i` den Wert 0 und wird in jedem Durchlauf um 1 erhöht. Im letzten Durchlauf trifft die Bedingung nicht mehr zu (bei `i = 9` ergibt `i < 10` nach der Erhöhung von `i` um 1 `false`). Die Schleife wird beendet.

- ✓ Sofern die Initialisierung von `i` bereits vor der `for`-Anweisung vorgenommen wurde, entfällt der Initialisierungsteil.

```
int i = 3;
for (; i < 10; i++)
{
    ...
}
```

- ✓ Sie können `for`-Anweisungen beliebig verschachteln. In diesem Beispiel wird die äußere Schleife 10-mal durchlaufen und in jedem dieser Durchläufe wird die innere Schleife 20-mal durchlaufen.

```
for (int i = 1; i < 11; i++)
{
    for (int j = 1; j < 21; j++)
    {
        ...
    }
}
```

✓ Beispiele für die Formulierung des Aktualisierungsteils:

`i++` `i` wird um 1 erhöht.
`i--` `i` wird um 1 erniedrigt.
`i -= 2` `i` wird um 2 erniedrigt.
`i *= 2` `i` wird mit 2 multipliziert.
`i /= 2` `i` wird durch 2 dividiert.
`i %= 3` `i` enthält den Rest bei der Division durch 3.

Die Besonderheiten der Verwendung der `foreach`-Schleife zur Behandlung von Strukturen werden in den thematisch zugehörigen Kapiteln 12 (Arrays) und 13 (Collections) behandelt.

5.9 Weitere Anweisungen in Kontrollstrukturen

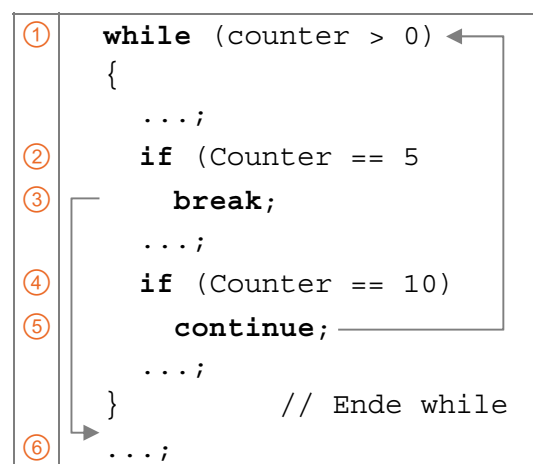
Schleifensteuerung mit **break** und **continue**

Zusätzlich zu der Schleifensteuerung haben Sie mit den beiden Anweisungen `break` und `continue` die Möglichkeit, den Ablauf des Programms zu beeinflussen. Die Anweisung `break` haben Sie bereits bei der Anwendung der `switch`-Anweisung kennengelernt.

- ✓ Die `break`-Anweisung kann innerhalb einer `while`-, `do-while`-, `for`- und `switch`-Anweisung angewendet werden und beendet die jeweilige Kontrollstruktur.
- ✓ Die Anweisung `continue` können Sie innerhalb einer `while`-, `do-while`- und `for`-Anweisung einsetzen. Bei `while` und `do-while` wird die Testbedingung neu ausgewertet. In der `for`-Anweisung wird der Aktualisierungsausdruck ausgeführt und danach die Testbedingung neu ausgewertet.

Beispiel für die Anwendung von **break** und **continue**

- ① Hier beginnt eine `while`-Anweisung.
- ② Innerhalb der `while`-Anweisung wird eine Bedingung durch eine `if`-Anweisung geprüft.
- ③ Wenn die Bedingung bei ② erfüllt ist, wird die `while`-Anweisung durch die Anweisung `break` verlassen. Die Anweisungen bei ⑥ werden als Nächstes ausgeführt.
- ④ Es wird eine weitere Bedingung durch eine `if`-Anweisung geprüft.
- ⑤ Wenn diese Bedingung erfüllt ist, wird die `continue`-Anweisung ausgeführt. Dies bewirkt, dass die Schleifenprüfung wieder bei ① beginnt und somit die restlichen Anweisungen nach der `continue`-Anweisung in diesem Schleifendurchlauf nicht mehr ausgeführt werden.




Schleifensteuerung mit `break` und `continue`

5.10 Java-Kontrollstrukturen im Überblick

Java-Anweisung	Zweck
if...	Es folgen Anweisungen, die abhängig von einer Bedingung einmal oder gar nicht ausgeführt werden sollen.
if... else...	Zwei alternative Anweisungsblöcke stehen zur Auswahl. In Abhängigkeit von einer Bedingung wird der <code>if</code> - oder <code>else</code> -Zweig mit seiner Anweisung oder seinem Anweisungsblock ausgeführt.
switch... case... default...	In Abhängigkeit von dem Wert eines Ausdrucks (Selektors) soll einer von mehreren Auswahlblöcken ausgeführt werden. Der Selektor muss vom Datentyp <code>char</code> , <code>byte</code> , <code>short</code> , <code>int</code> oder vom Referenztyp <code>String</code> sein. Stimmt keiner der <code>case</code> -Werte mit dem Wert des Selektors überein, werden die Anweisungen nach <code>default</code> ausgeführt.
for...	Eine oder mehrere Anweisungen werden in Abhängigkeit von einer Schleifenbedingung ausgeführt.
while...	Anweisungen sollen in Abhängigkeit von einer Bedingung einmal, mehrmals oder gar nicht ausgeführt werden. Die Bedingung wird am Anfang der Schleife geprüft.
do... while...	Anweisungen sollen in Abhängigkeit von einer Bedingung mindestens einmal oder mehrmals abgearbeitet werden. Die Bedingungsprüfung findet am Ende der Schleife statt.
break	In Abhängigkeit von einer Bedingung kann ein Anweisungsblock der <code>while</code> -, <code>do</code> -, <code>for</code> - und <code>switch</code> -Anweisung abgebrochen werden. Es wird die gesamte Anweisung beendet und die danach folgende Anweisung ausgeführt.
Continue	In Abhängigkeit von einer Bedingung kann in einer <code>while</code> -, <code>do-while</code> - oder <code>for</code> -Anweisung wieder zur Auswertung der Bedingung gesprungen werden.

5.11 Übung

Kontrollstrukturen verwenden

Level		Zeit	ca. 25 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Verwendung von Kontrollstrukturen ✓ Verarbeitung von Eingabeparametern ✓ Formatierte Ausgabe 		
Übungsdatei	--		
Ergebnisdateien	<i>TestResult.java, TestResult2.java, TestEvaluation.java, Thirty.java, Square.java</i>		

1. Erstellen Sie ein Programm **TestResult**, das die Bewertung eines Tests als Text ausgibt. In dem Test können maximal 10 Punkte erreicht werden. Sofern mindestens 7 Punkte erreicht wurden, gilt der Test als bestanden. Die Punktzahl soll als Parameter dem Programm übergeben werden. Die Textausgaben sollen folgendermaßen lauten:
 Mindestens 7 Punkte: "Der Test ist bestanden!"
 Anderenfalls: "Der Test ist leider nicht bestanden!"
2. Prüfen Sie zusätzlich, ob die eingegebene Punktzahl auch im zulässigen Bereich liegt. Anderenfalls soll folgender Text ausgegeben werden: "FEHLER: Ungültige Punktzahl"
3. Nehmen Sie in einem neuen Programm **TestEvaluation** eine detaillierte Auswertung vor. Die Ausgabe soll wieder als Text erfolgen.
 10 Punkte: "Ergebnis: Sehr gut"
 9 Punkte: "Ergebnis: Gut"
 8 Punkte: "Ergebnis: Befriedigend"
 7 Punkte: "Ergebnis: Ausreichend"
 weniger als 7 Punkte: "Ergebnis: Leider nicht genügend Punkte erreicht"
 Welche Verzweigungsstruktur eignet sich bei mehreren (mehr als zwei) Alternativen?
4. Geben Sie in einem Programm **Thirty** nacheinander alle ungeraden Zahlen zwischen 1 und 30 aus. Verwenden Sie hier zur Übung den Modulo-Operator (%).
5. Ein Programm **Square** soll, bei der Zahl 1 beginnend, in einer Schleife die Quadratzahlen (Zahl * Zahl) ausgeben. Die Schleife soll so lange durchlaufen werden, wie die Zahl, zu der das Quadrat berechnet wird, kleiner oder gleich 15 ist.

6

Klassen, Attribute, Methoden

6.1 Klassen

Was ist eine Klasse, was sind Objekte?

Eine **Klasse** beschreibt als Bauplan die Gemeinsamkeiten einer Menge von **Objekten**. Eine Klasse ist somit ein Modell, auf dessen Basis Objekte erstellt werden können. Die Klasse beinhaltet die vollständige Beschreibung dieses Modells. So vereint eine Klasse alle **Attribute** (auch Datenelemente oder Eigenschaften), die diese Klasse kennzeichnen, und **Methoden** zur Verwendung der Attribute und zur Beschreibung der Funktionalität.

Klasse
Attribute
Methoden

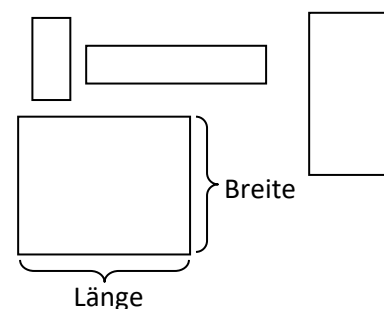
Objekte stellen konkrete Exemplare der Klasse dar. Auf der Basis einer Klasse können beliebig viele Objekte erzeugt (instanziiert) werden. In den Attributen werden die objektspezifischen Eigenschaften des Objekts gespeichert, während die Methoden Aktionen ausführen. So können Methoden Daten für die Klasse in Empfang nehmen, Eigenschaften (Daten) der Klasse ausgeben oder die Eigenschaften der Klasse bearbeiten.

Beispiel: Aus der Geometrie kennen Sie Formen, die vier Eckpunkte besitzen und deren benachbarte Kanten im rechten Winkel (90°) zueinander stehen.

Alle diese Formen gehören zur Gruppe (Klasse) der Rechtecke. Das Rechteck mit der Breite 3 cm und der Länge 5 cm ist ein bestimmtes Exemplar (Objekt) dieser Klasse.

Rechtecke können verschieden lang sein, aber sie besitzen alle das Attribut Länge. Auf diese Weise lassen sich gemeinsame Attribute finden, die die Klasse Rechteck auszeichnen:

- ✓ Breite
- ✓ Länge



Klassen – individuelle Datentypen erstellen

Die bisher verwendeten elementaren Datentypen konnten lediglich einen Wert speichern. Mit einer Klasse wie beispielsweise der Klasse Rechteck erzeugen Sie einen neuen Datentyp, der sich durch verschiedene Attribute und eine spezielle Funktionalität auszeichnet.

Java ist eine rein objektorientierte Programmiersprache, die auf diesem Konzept aufbaut. Sämtlicher Programmcode befindet sich innerhalb von Klassenbeschreibungen. Variablen und Methoden sind immer innerhalb einer Klasse beschrieben.

Syntax für die grundlegende Klassendefinition in Java

- ✓ Die Definition einer Klasse beginnt mit dem Schlüsselwort `class` (Klasse).
- ✓ Dem Schlüsselwort `class` folgt der Name der Klasse, der sich an die Konventionen von Bezeichnern halten muss.
- ✓ Alle weiteren Beschreibungen innerhalb der Klasse werden in geschweifte Klammern `{ }` eingeschlossen.
- ✓ Die Klassendefinition sollte in einer eigenständigen Textdatei gespeichert werden. Der Name dieser Datei entspricht dem Namen der Klasse und erhält die Erweiterung `.java`.
- ✓ Der Klassendefinition können Sie sogenannte Modifizierer (modifiers) voranstellen. Mit Modifizierern haben Sie die Möglichkeit, beispielsweise die Zugriffsrechte auf die Klasse festzulegen.

```
[modifiers] class identifier
{
    ...
}
```

Der Name einer Klasse sollte immer mit **Großbuchstaben** beginnen. Es wird empfohlen, Substantive zu verwenden, die dann jeweils ohne Leerzeichen angehängt werden und ebenfalls mit einem Großbuchstaben beginnen. Sinnvolle Klassennamen sind beispielsweise *ProductList* oder *CustomerData*.

Beispiel für die Definition einer Klasse: *Rectangle.java*

- ✓ Der nebenstehende Quelltextauszug zeigt die Definition einer Klasse Rechteck (Rectangle).
- ✓ Diese Klassendefinition schreiben Sie in eine neue Datei, deren Name dem Klassennamen entspricht (*Rectangle.java*).

```
class Rectangle
{
    ...
}
```

Empfehlungen zum Formatieren des Quelltextes

- ✓ **Geschweifte Klammern:** Bezüglich der Syntax ist es unerheblich, ob die geschweifte Klammer `{ }` bereits hinter dem Klassennamen oder aber in einer neuen Zeile beginnt. Die Schreibweise sollte allerdings einheitlich sein.
- ✓ **Einrücken:** Alle Anweisungen innerhalb geschweiften Klammern `{ }` (eines Blocks) sollten zwei Zeichen eingerückt werden, damit der Quelltext besser lesbar ist.

6.2 Die Attribute einer Klasse

Attribute

Lokale Variablen werden **innerhalb** einer Methode in einem Anweisungsblock definiert und sind auch nur in diesem Bereich gültig.

Neben lokalen Variablen kennt Java sogenannte **Attribute**. Attribute werden **außerhalb** einer Methode in der Klassenbeschreibung definiert.

Klasse
Attribute
Methoden

Die Klasse `Rectangle` enthält beispielsweise die Attribute `width` (Breite) und `length` (Länge). Für ein spezielles Rechteck, das auf Basis dieses neuen Datentyps erzeugt wird, stellen die Attribute die Eigenschaften dar. Ein spezielles Rechteck hat dann beispielsweise folgende Eigenschaften:

- ✓ Es ist 4 cm breit.
- ✓ Es ist 6 cm lang.

So können die Werte für diese Variablen individuell für jedes Objekt festgelegt werden. Die Variablen gehören somit zu dem Objekt und werden **Instanzvariablen** genannt.

Attribute definieren

Je nachdem, welche Art von Daten die Attribute beschreiben, werden unterschiedliche Datentypen verwendet. Ein einfacher Datentyp zur Darstellung von numerischen Werten wie beispielsweise für die Breite und die Länge eines Rechtecks ist der Datentyp `int`.

Syntax für die Definition von Attributen

Ein Attribut einer Klasse wird wie eine lokale Variable definiert.

- ✓ Die Definition eines Attributs beginnt mit dem Schlüsselwort für den Datentyp. `[modifiers] type identifier[, identifier1...];`
- ✓ Hinter dem Datentyp folgt der Name des Attributs (`identifier`), der den Konventionen von Bezeichnern entspricht.
- ✓ Um dem Attribut einen definierten Wert zu geben, sollten Sie diesen mit einer Wertzuweisung festlegen.
- ✓ Die Definition eines Attributs ist eine Anweisung und wird daher mit einem Semikolon abgeschlossen.
- ✓ Der Definition können Modifizierer vorangestellt werden, über die Sie wie bei der Definition einer Klasse beispielsweise die Zugriffsrechte festlegen können.
- ✓ In einer Anweisung können Sie mehrere Attribute des gleichen Typs definieren. Dazu geben Sie, mit Kommata getrennt, die Namen der Attribute an.

Die Benennung von Attributen muss den bereits bekannten Konventionen entsprechen. Der Name eines Attributs sollte immer mit einem Kleinbuchstaben beginnen. Sinnvolle Attributnamen sind beispielsweise auch *numberArticles* oder *maxTextLength*.

Das Beispiel zeigt eine Klasse Rechteck (*Rectangle*), die zwei Attribute vom Datentyp *int* enthält.

```
class Rectangle
{
    int width = 0;
    int length = 0;
    ...
}
```

6.3 Objekte erzeugen

Neue Objekte mit **new** erzeugen

Nach der Definition einer Klasse als allgemeine Beschreibung von Objekten lassen sich beliebig viele Objekte (Exemplare) von ihr erzeugen. Die Objekte einer Klasse unterscheiden sich in den Daten (Werte der Attribute). Die Erzeugung (Instanziierung) eines Objektes auf der Basis einer Klasse ähnelt der Erzeugung einer lokalen Variablen.

Syntax von **new**

Ein Objekt einer Klasse wird in zwei Schritten erstellt:

- ① Zunächst definieren Sie eine Variable vom Typ der gewünschten Klasse.
Die Variable wird als Referenzvariable bezeichnet und ist später nur eine Referenz (Verweis) auf das eigentliche Objekt. Es wird nur Speicher für die Referenz angelegt. Das Objekt selbst existiert noch nicht.

```
① ClassName identifier;
② identifier = new ClassName();
```

- ② Anschließend erzeugen Sie mit dem Operator **new** ein Objekt der Klasse. Nach **new** geben Sie den Klassennamen und runde Klammern `()` ein.

Beispiel für die Erzeugung eines Objekts einer Klasse

```
class Rectangle ② //Neuer Datentyp
{
    int width = 0; //Attribut Breite
    int length = 0; //Attribut Länge
    ...
}
```

```
class Geometry
{
    public static void main(String[] args)
    {
        // ein Rechteck definieren
    }
}
```

①	<code>Rectangle specialRect;</code>
	<code>// das Rechteck erzeugen</code>
③	<code>specialRect = new Rectangle();</code>
	<code>...</code>
	<code>}</code>
	<code>...</code>
	<code>}</code>

- ① Die Variable `specialRect` vom Typ (der Klasse) `Rectangle` wird deklariert.
- ② Die Klasse `Rectangle` wird definiert.
- ③ Mit dem Operator `new` wird ein Objekt der Klasse `Rectangle` erzeugt und die Referenz auf dieses Objekt der Referenzvariablen `specialRect` zugewiesen.

Die Klassen und `Geometry` müssen sich hier im gleichen Verzeichnis befinden.

Die Erzeugung einer Variablen eines primitiven Datentyps und eines Objekts im Vergleich

Die folgende Gegenüberstellung der Erzeugung einer lokalen Variablen des primitiven Datentyps `int` und der Erzeugung eines Objektes einer Klasse soll Ihnen diese Vorgehensweise und die Auswirkung auf die Speicherbelegung verdeutlichen:

Erzeugung einer Variablen eines primitiven Datentyps:

...	<div></div>	number
① <code>int number;</code>		
② <code>number = 4;</code>	<div>4</div>	number
...		

- ① Bei der Definition einer Variablen vom Datentyp `int` wird so viel Speicherplatz bereitgestellt, der für die Aufnahme von ganzen Zahlen des Datentyps `int` erforderlich ist.
- ② Bei der Wertzuweisung wird der Wert an der entsprechenden Speicherstelle abgelegt.

Erzeugung eines Objekts einer Klasse:

...	<div></div>	
① <code>Rectangle specialRect;</code>	<div>specialRect</div>	
② <code>specialRect = new Rectangle();</code>	<div>●</div> <div>specialRect</div>	<div> <div>④</div> <div>→</div> <div> <div>③</div> <div>Rectangle</div> <div>...</div> <div>width <div>0</div> ⑤</div> <div>length <div>0</div></div> <div>...</div> </div> </div>

- ① Bei der Definition einer Variablen vom Typ der Klasse `Rectangle` wird eine sogenannte **Referenzvariable** erzeugt, die später eine Referenz (einen Verweis) auf das eigentliche Objekt darstellt. Das Objekt selbst existiert zunächst noch nicht. Speicherplatz wird auch nur für die Aufnahme der Referenz belegt.
- ② Erst mit dem Operator `new` wird ein Objekt ③ der Klasse erzeugt. Dabei wird auch entsprechend Speicherplatz für das Objekt bereitgestellt.
- ④ Die Referenzvariable verweist nun auf dieses neue Objekt.
- ⑤ Sofern den Attributen in der Klassendefinition spezielle Werte zugewiesen wurden, werden diese Werte als Eigenschaften der Objekte vorgegeben ⑤. Anderenfalls wird beispielsweise bei Attributen vom Typ `int` automatisch der Wert 0 verwendet.

Objekte in einem Schritt definieren und erzeugen

Sie können die Definition und die Initialisierung einzeln vornehmen oder die beiden Schritte zusammenfassen. So können Sie die Objekte sofort bei der Definition erzeugen oder erst in einer dazu vorgesehenen Methode.

```
...  
// ein Rechteck definieren und erzeugen  
Rectangle anotherRect = new Rectangle();  
...
```

Auf die Eigenschaften (Attribute) eines Objekts zugreifen

Die mit `new` erzeugten Objekte besitzen die Attribute der entsprechenden Klasse. Sie können auf die Attribute der Objekte zugreifen, indem Sie den Namen des Objektes mit einem Punkt abgetrennt vor den Attributnamen schreiben.

Beispiel: *Geometry.java*

Im folgenden Beispiel werden zwei Objekte vom Typ der Klasse `Rectangle` erzeugt. Beide Objekte erhalten unterschiedliche Eigenschaften, d. h. unterschiedliche Breite und Länge.

```
class Geometry  
{  
    public static void main(String[] args)  
    {  
        Rectangle specialRect = new Rectangle ();  
        Rectangle anotherRect = new Rectangle ();  
        ① specialRect.width = 5;  
           specialRect.length = 8;  
           anotherRect.width = 3;  
        ② anotherRect.length = 2 * anotherRect.width;  
           System.out.printf("1. Rechteck:\nBreite: %d\nLaenge: %d\n",  
                             specialRect.width, specialRect.length);  
           System.out.printf("2. Rechteck:\nBreite: %d\nLaenge: %d\n",  
                             anotherRect.width, anotherRect.length);  
    }  
}
```

Sie können für die Objekte `specialRect` und `anotherRect`

- ① die Attribute (`width` und `length`) verändern,
- ② die Attribute (`width` und `length`) auslesen.

```
1. Rechteck:
Breite:5
Laenge:8
2. Rechteck:
Breite:3
Laenge:6
```

Die Ausgabe des Programms

Zuweisungen bei Variablen primitiver Datentypen und bei Referenzvariablen im Vergleich

Zuweisungen bei Variablen eines primitiven Datentyps

Ausgangszustand	Zuweisung	Ergebnis
<div>3 number</div> <div>7 counter</div>	<div>3 number</div> <div>↓ Inhalt kopieren</div> <div>7 counter</div>	<div>3 number</div> <div>3 counter</div>
<pre>int number = 3; int counter = 7;</pre>	<pre>counter = number;</pre>	

Zuweisungen bei Referenzvariablen

Ausgangszustand	Zuweisung	Ergebnis
①	<code>rect2 = rect1;</code>	Beide Referenzvariablen verweisen auf dasselbe Objekt.

```

① Rectangle rect1;
   rect1 = new Rectangle();
   rect1.width = 2;
   rect1.length = 7;
   Rectangle rect2;
   rect2 = new Rectangle();
   rect2.width = 3;
   rect2.length = 4;

```

6.4 Methoden – die Funktionalität der Klassen

Funktionalität mit Methoden beschreiben

Durch Attribute legen Sie die Eigenschaften von Objekten einer Klasse fest. Neben diesen Attributen können auch Methoden zu einer Klasse gehören. Mit Hilfe von Methoden drücken Sie die Funktionalität der Objekte einer Klasse aus.

Klasse
Attribute
Methoden

- ✓ Methoden werden wie die Attribute innerhalb einer Klasse beschrieben.
- ✓ Methoden gehören immer zu der Klasse.
- ✓ Eine Klasse kann mehrere Methoden enthalten, wobei die Methoden in beliebiger Reihenfolge beschrieben werden können.
- ✓ Mit Methoden führen Sie eine Aktion aus, verändern Daten oder geben Daten aus.
- ✓ Methoden können nicht geschachtelt werden.

Methoden der Klasse `Rechteck` sind beispielsweise:

- ✓ das Berechnen und Ausgeben des Flächeninhalts,
- ✓ das Verändern der Länge bzw. der Breite.

Methoden erstellen

Syntax für die Beschreibung einer einfachen Methode vom Typ `void`

- ✓ Die Beschreibung einer Methode erfolgt durch den Methodenkopf ① und dem Methodenrumpf ②.

```

[modifiers] void identifier() ①
{
    ...                          ②
}

```

- ✓ Der Methodenkopf besteht aus dem Namen der Methode, der sich an die Konventionen von Bezeichnern hält, und runden Klammern. Das Schlüsselwort `void` kennzeichnet eine Methode, die keinen Rückgabewert liefert.
- ✓ Anschließend folgt ein Block mit Anweisungen, der in geschweifte Klammern `{ }` gesetzt wird.
- ✓ Der Methodendefinition können Sie sogenannte Modifizierer (`modifiers`) voranstellen. Mit Modifizierern haben Sie die Möglichkeit, beispielsweise die Zugriffsrechte auf die Methode festzulegen.

Methodennamen beginnen gewöhnlich mit einem Kleinbuchstaben. Sinnvolle Methodennamen sind z. B. `getPageNO` oder `printBill`.

Beispiel für das Erstellen einer einfachen Methode: *Rectangle.java*

In dem Beispiel der Klasse Rechteck (*Rectangle*) möchten Sie mithilfe einer Methode die Länge und die Breite eines Rechtecks vertauschen.

```
class Rectangle
{
    int width = 5;
    int length = 7;
    ...
    void swapWidthLength()           //einfache Methode: Laenge und
                                    //Breite vertauschen
    {
        ①    int temp = width;         //lokale Variable
        ②    width = length;          ④
        ③    length = temp;
    }
}
```

Eine einfache Methode

- ① Die lokale Variable `temp` wird innerhalb des Methodenrumpfs, also eines Blocks, definiert und ist somit auch nur innerhalb dieses Blocks ④ gültig. Mithilfe dieser Variablen „merkt“ sich die Methode den Wert des Attributs `width`.
- ② Das Attribut `width` erhält nun den bisherigen Wert des Attributs `length`.
- ③ Abschließend wird der bisherige (zwischengespeicherte) Wert des Attributs `width` als neuer Wert für das Attribut `length` festgelegt.

Methoden anwenden

Um eine Methode anzuwenden, benötigt der Compiler folgende Informationen:

```
objectName.methodName();
```

- ✓ Name des Objektes, dessen Methode ausgeführt werden soll,
- ✓ Name der Methode.

Beispiel: *SimpleMethods.java*

```
class SimpleMethods
{
    public static void main(String[] args)
    {
        Rectangle specialRect = new Rectangle();
    }
}
```

```

① specialRect.width = 5;
   specialRect.length = 8;
   System.out.printf("Urspruenglich:\nBreite: %d\nLaenge: %d\n",
                     specialRect.width, specialRect.length);
                                   ③
② specialRect.swapWidthLength();
   System.out.printf("Vertauscht:\nBreite: %d\nLaenge: %d\n",
                     specialRect.width, specialRect.length);
}
}

```

- ① Die Eigenschaft des Objekts wird hier außerhalb der Klasse Rechteck (Rectangle) verändert. Daher muss das Objekt genannt werden, dessen Eigenschaft verändert werden soll. Objektname und Attributname werden durch einen Punkt getrennt.
- ② Der Aufruf der Methode (swapWidthLength) erfolgt ebenfalls außerhalb der Klasse (Rectangle). Daher wird auch hier vor dem Methodennamen mit einem Punkt als Trennzeichen der Name des Objekts aufgeführt.
- ③ Im Gegensatz zum Zugriff auf ein Attribut wird der Zugriff auf eine Methode durch runde Klammern gekennzeichnet.

```

Urspruenglich:
Breite: 5
Laenge: 8
Vertauscht:
Breite: 8
Laenge: 5

```

Die Ausgabe des Programms

6.5 Methoden mit Parametern erstellen

Häufig benötigen Methoden weitere Informationen zur Ausführung, beispielsweise um neue Werte für Attribute festzulegen.

Syntax für die Beschreibung einer Methode vom Typ **void** mit Parametern

```

[modifiers] void identifier(type identifier1[, type identifier2,...])
{
    ...
}

```

①

- ✓ Die runden Klammern der Methode beinhalten die Beschreibung der Parameter.
- ✓ Die Namen der Parameter halten sich an die Konventionen von Bezeichnern.
- ✓ Mehrere Parameter werden durch Kommata getrennt.
- ✓ Wie bei einer Variablendefinition werden für jeden Parameter der Datentyp und der Name angegeben.
- ✓ Der Datentyp kann ein primitiver Datentyp oder ein Referenztyp (eine Klasse) sein.
- ✓ Der Teil des Methodenkopfs, der den Methodennamen und die in Klammern eingeschlossenen Parameter umfasst, wird **Signatur** ① der Methode genannt.

Die Namen der Parameter beginnen wie Variablennamen mit einem Kleinbuchstaben.

Eine Methode mit einem Parameter erstellen

In diesem Beispiel soll eine Methode `buildSquare` das Rechteck so verändern, dass beide Kantenlängen (`width` und `length`) auf einen bestimmten Wert (`sideLength`) gesetzt werden. Methoden können zu diesem Zweck Informationen (Werte) von dem aufrufenden Programm entgegennehmen. Diese Übergabewerte werden **Parameter** genannt.

Beispiel: *Rectangle.java*

Folgendermaßen könnte die Methode in der Klasse `Rechteck` formuliert werden:

```
class Rectangle
{
    ...
    ① void buildSquare(int sideLength)
    {
    ②     width = sideLength;
        length = sideLength;
    }
}
```

- ① In diesem Beispiel benötigt die Methode einen Parameter vom Datentyp `int`.
- ② Die Anweisungen können den Parameter wie eine Variable nutzen.

Eine Methode mit einem Parameter aufrufen

Für den Aufruf der Methode verwenden Sie den Methodennamen und in Klammern die Parameter. Die Parameter können Werte (Literele), Variablen oder Ausdrücke sein, die einen Wert des entsprechenden Datentyps liefern.

Beispiel: *MethodsWithArguments.java*

```
class MethodsWithArguments
{
    public static void main(String[] args)
    {
        Rectangle specialRect = new Rectangle();
        specialRect.width = 5;
        specialRect.length = 8;
        System.out.printf("Ein Rechteck:\nBreite: %d\nLaenge: %d\n",
                           specialRect.width, specialRect.length);
    ① specialRect.buildSquare(7);
        System.out.printf("Ein Quadrat:\nBreite: %d\nLaenge: %d\n",
                           specialRect.width, specialRect.length);
    }
}
```

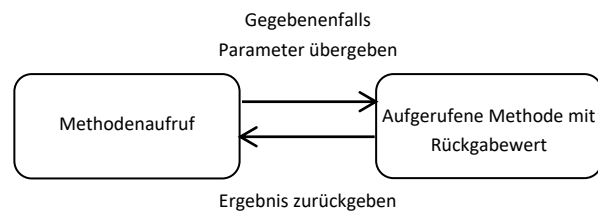

- ① Innerhalb der runden Klammern der Methode werden beim Aufruf der Methode die Parameter aufgeführt.

```
Ein Rechteck:
Breite: 5
Laenge: 8
Ein Quadrat:
Breite: 7
Laenge: 7
```

Die Ausgabe des Programms

6.6 Methoden mit Rückgabewert definieren

In vielen Fällen soll mit einer Methode eine Funktionalität beschrieben werden, die als Antwort Informationen an die ausführende Methode zurückliefert. Um dies in Java zu ermöglichen, können Sie für eine Methode einen Rückgabewert festlegen.



Syntax für Methoden mit Rückgabewert

```
[modifiers] type identifier([type identifier1,...])
{
    ...
    return expression;
}
```

- ✓ Vor dem Namen der Methode geben Sie den Datentyp an, den der zurückgegebene Wert der Methode erhalten soll. Dies kann ein primitiver Datentyp sein oder aber auch ein Referenztyp (eine Klasse).
- ✓ Mit dem Schlüsselwort `return` wird das Ergebnis zurückgegeben und die Methode beendet.

Das Schlüsselwort `void`

Bei der Definition der vorherigen Methoden haben Sie das Schlüsselwort `void` statt eines Rückgabewertes vor den Methodennamen gesetzt. Das Schlüsselwort `void` (engl. für „leer“) bedeutet, dass die Methode **keinen Ergebniswert** zurückgibt.

Wenn eine Methode keinen Wert zurückgibt (Kennzeichnung durch `void`), dann entfällt bei dieser Methode üblicherweise die `return`-Anweisung. Sie **darf** jedoch angegeben werden (allerdings ohne einen entsprechenden Rückgabewert). `return` bewirkt, dass die Methode verlassen wird.

Beispiel für eine Methode mit einem Rückgabewert: *Rectangle.java* und *MethodsWithReturn.java*

Im folgenden Beispiel wird die Klasse Rechteck um eine Methode erweitert, mit der der Flächeninhalt des Rechtecks berechnet wird. Ein Parameter ist nicht erforderlich, da hier ausschließlich die Attribute `laenge` und `breite` für die Berechnung benötigt werden.

- ① In diesem Beispiel soll der Ergebniswert vom Datentyp `int` sein.
- ② Die Fläche wird berechnet und in der Variablen `result` gespeichert.
- ③ Über den Aufruf des reservierten Wortes `return` wird die Methode verlassen und das Ergebnis als Wert zurückgegeben.

```
class Rectangle
{
    ...
    ① int getArea()
    {
        ② int result = width * length;
        ③ return result;
    }
}
```

Die Schreibweise der Anweisungen ② und ③ lässt sich verkürzen, indem die Berechnung direkt hinter das Schlüsselwort `return` gestellt wird.

```
...
int getArea()
{
    return width * length;
}
```



Bei einer Methode mit Rückgabewert müssen Sie sicherstellen, dass in jedem Fall eine `return`-Anweisung ausgeführt wird.

6.7 Methoden überladen

Innerhalb einer Klasse können mehrere Methoden mit demselben Namen existieren. Jedoch müssen sie sich hinsichtlich der Anzahl oder des Datentyps der Parameter unterscheiden. Der Methodenname in Verbindung mit den Parametern wird **Signatur** genannt. Die Signatur muss innerhalb der Klasse eindeutig sein. Die Programmierung mehrerer Methoden innerhalb einer Klasse mit gleichem Namen und unterschiedlicher Anzahl von Parametern wird **Überladen (Overloading)** genannt.

Beispiel: *Rectangle.java* und *Overloading.java*

```
class Rectangle
{
    ...
    ① void resize(int newWidth, int newLength)
    {
        width = newWidth;
        length = newLength;
    }
    void resize(double factor)
```

```
② {
    setWidth(((int) width * factor));
    setLength(((int) length * factor));
    // auch moeglich:
    // resize(((int) width * factor), ((int) length * factor));
}
```

- ① Die Klasse `Rectangle.java` besitzt eine Methode `resize(int width, int length)` zur Festlegung der Rechtecksgröße. Die beiden Parameter legen die künftige Breite und die künftige Länge fest.
- ② Außerdem existiert eine zweite gleichnamige Methode, mit der das Rechteck proportional über einen Faktor vergrößert bzw. verkleinert werden kann. Diese Methode benötigt dann nur ein Argument: `resize(double factor)`.

Anhand der Signatur (Methodennamen und Anzahl bzw. Datentyp der Argumente) entscheidet der Compiler, welche der beiden Methoden verwendet wird, beispielsweise `resize(3, 7)` ① oder `resize(5.0)` ②. Die Vielgestaltigkeit, in der sich eine Methode zeigen kann, ist eine Art der sogenannten **Polymorphie**, die ein wichtiges Merkmal objektorientierter Sprachen ist.

6.8 Statische Variablen und Methoden

Was sind statische Variablen und statische Methoden?

Statische Variablen und Methoden sind nicht an die Existenz eines Objekts einer Klasse gebunden. Objektvariablen gehören zu einem Objekt vom Datentyp der Klasse und stellen die Eigenschaften des Objekts dar. Sie existieren somit erst dann, wenn das Objekt existiert, d. h. nach der Initialisierung der Instanzvariablen. Demgegenüber gehören statische Variablen zu der Klasse selbst. Sie werden daher auch als **Klassenvariablen** bezeichnet. Klassenvariablen existieren genau einmal, und es ist nicht erforderlich, dass Objekte vom Datentyp der Klasse existieren.

Ebenso ist die Festlegung von statischen Methoden möglich. Statische Methoden arbeiten unabhängig von den Attributen der Klasse.

Einsatzgebiete von statischen Variablen und Methoden sind Klassen, die bestimmte Funktionen bereitstellen, die unabhängig von Objekten operieren. Beispielsweise sind alle Methoden der Klasse `java.lang.Math` statisch und stellen mathematische Grundfunktionen bereit. Dafür ist keine Erzeugung eines Objekts notwendig. Da sich in Java alle Methoden in Klassen befinden müssen, ist dies die einzige Möglichkeit, derartige Methoden zu erzeugen.

Statische Variablen und Methoden verwenden

Sie können jederzeit über den Klassennamen (ohne dass ein Objekt der Klasse existiert) auf statische Variablen und Methoden der Klasse zugreifen.

Syntax von statischen Variablen und Methoden

```
[modifiers] static type identifier;  
  
[modifiers] static type identifier1([type identifier2,...])  
{  
    return expression;  
}
```

- ① Um eine statische Variable oder Methode zu erzeugen, beginnen Sie die Definition mit dem Schlüsselwort `static`.
- ② Statische Variablen können Sie wie andere Variablen sofort initialisieren.

Statische Methoden können nur mit statischen Variablen und Methoden arbeiten. Diese sind unabhängig von einem Objekt der Klasse immer vorhanden.

Beispiele mit statischen Variablen und Methoden: *StaticElements.java* und *UseStatic.java*

Das folgende Beispiel zeigt eine Klasse `StaticElements`, in der eine statische Variable und eine statische Methode definiert sind.

```
class StaticElements  
{  
①    static int DotsPerInch = 300;  
②    static double calcSquare(double wert)  
    {  
        return wert * wert;  
    }  
}
```

Die Klasse *StaticElements*

- ① Eine statische Variable speichert den Wert für eine Druckerauflösung.
- ② Außerdem existiert eine statische Methode `calcSquare`, mit der für eine beliebige Zahl das Quadrat berechnet wird.

In der folgenden Klasse `UseStatic` wird die Klasse `StaticElements` verwendet.

```

class UseStatic
{
    public static void main(String[] args)
    {
        System.out.printf("Druckerauflösung: %d dpi%n",
                           StaticElements.DotsPerInch);

        double inputValue = Double.parseDouble(args[0]);
        double result = StaticElements.calcSquare(inputValue);
        System.out.printf("Das Quadrat von %g ist %g%n",
                           inputValue, result);
    }
}

```

Die Klasse *UseStatic*

```

Druckerauflösung: 300 dpi
Das Quadrat von 4.50000 ist 20.2500

```

Die Ausgabe des Programms

Die statische Methode **main**

Um eine Anwendung zu schreiben, benötigen Sie mindestens eine Klasse, in der Sie die Methode `main`, das sogenannte Hauptprogramm, schreiben. Von der Klasse `UseStatic` wird kein Objekt erzeugt. Daher muss die Methode `main` statisch sein, damit sie dennoch vom Interpreter ausgeführt werden kann.

Syntax der Methode **main**

- ✓ Benennen Sie die Klasse mit dem Programmnamen ①.
- ✓ Speichern Sie die Klasse in einer gleichnamigen Datei mit der Erweiterung *.java*.
- ✓ Innerhalb dieser Klasse schreiben Sie eine Methode `main`, die beim Programmstart ausgeführt wird.
- ✓ Der Anweisungsblock der Methode `main` enthält Anweisungen, d. h. die Definition von Variablen, Wertzuweisungen und Methodenaufrufe.

```


class ProgramName ①
{
    ...
    /**
     * die Methode main, das Hauptprogramm
     */
    public static void main(String[] args)
    {
        //statements
        // - Variablendefinitionen
        // - Zuweisungen
        // - Methodenaufrufe
    }
    ...
}

```

Die Methode `main` verwendet Argumente. Auf diese Weise lassen sich beim Programmstart Parameter übergeben.

6.9 Übung

Klassen erstellen und verwenden

Level		Zeit	ca. 25 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Erstellung von Klassen ✓ Definition von Methoden ✓ Statische Methoden 		
Übungsdatei	--		
Ergebnisdateien	<i>SomeMaths.java, Circle.java, CircleExercise.java</i>		

1. Erzeugen Sie eine Klasse `SomeMaths`, die die Definition der Konstanten `pi` ($\pi = 3,14159$) und eine Funktion `getSquare` zum Quadrieren von Zahlen des Datentyps `double` bereitstellt. Das Ergebnis soll ebenfalls vom Typ `double` sein.
2. Erzeugen Sie eine Klasse `Circle`, die Sie durch folgende Attribute und Funktionalität beschreiben:

Attribute

- ✓ `radius` (Typ `double`)

Funktionalität

- ✓ **`getCircumference`**: Berechnung und Rückgabe des Kreisumfangs (Berechnung: $\text{Umfang} = 2 * \pi * \text{Radius}$)
 - ✓ **`getArea`**: Berechnung und Rückgabe der Kreisfläche (Berechnung: $\text{Fläche} = \pi * \text{Radius}^2$)
3. Erstellen Sie eine Klasse `CircleExercise`, in der Sie das Hauptprogramm definieren.
 4. Lassen Sie im Hauptprogramm drei Kreise erzeugen.
 5. Verändern Sie die Eigenschaften der Kreise so, dass alle Kreise unterschiedlich groß sind (`radius`).
 6. Lassen Sie für jeden Kreis den Radius, den Umfang und den Flächeninhalt in der Konsole ausgeben.

```

1. Kreis:
Radius: 3.00000
Umfang: 18.8495
Flaeche: 28.2743

2. Kreis:
Radius: 8.00000
Umfang: 50.2654
Flaeche: 201.062

3. Kreis:
Radius: 5.00000
Umfang: 31.4159
Flaeche: 78.5398

```

Die Ausgabe des Programms



Wissenstests: *Java 9 – Grundlagen_1 und – Grundlagen_2*

7

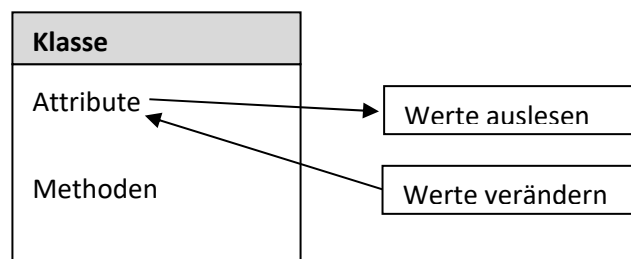
Kapselung und Konstruktoren

7.1 Kapselung

Zugriff auf die Attribute kontrollieren

Eine Klasse vereinigt die Daten (Attribute) und die Funktionalität (Methoden) in einer einzigen Struktur.

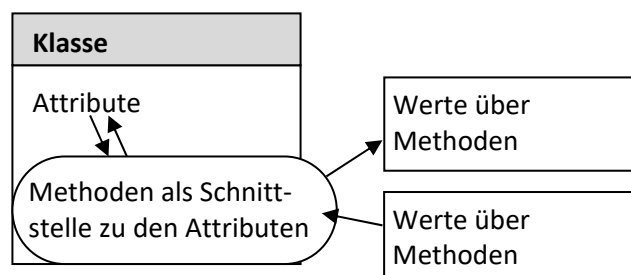
In den bisherigen Beispielen konnten Methoden anderer Klassen (beispielsweise die Methode `main`) auf die Attribute direkt zugreifen und die Werte auslesen und verändern.



Attribute im direkten Zugriff von „außen“

Was ist Kapselung?

Bei der Kapselung werden die Attribute wie in einer Kapsel in der Klasse eingeschlossen und „nach außen abgeschirmt“. Nur die Methoden der Klasse selbst haben Zugriff auf die Attribute. Auf diese Weise lässt sich der Zugriff auf die Attribute kontrollieren.



Kontrollierter Zugriff auf gekapselte Attribute

- ✓ Nur Methoden der Klasse können Werte nach „draußen“ weitergeben.
- ✓ Nur Methoden der Klasse können Werte in den Attributen speichern.
- ✓ Methoden können Werte prüfen und verändern, bevor sie in den Attributen gespeichert werden.
- ✓ Methoden können Werte aufbereiten und verändern, bevor sie weitergegeben werden.

Was sind Zugriffsmodifizierer?

Zugriffsmodifizierer sind Schlüsselwörter, mit denen die Zugriffsrechte für die Elemente einer Klasse (Attribute und Methoden) festgelegt werden. Zugriffsmodifizierer schreiben Sie bei der Definition des Attributs bzw. der Methode vor den Datentyp.

```
[modifiers] type identifier[, ...];

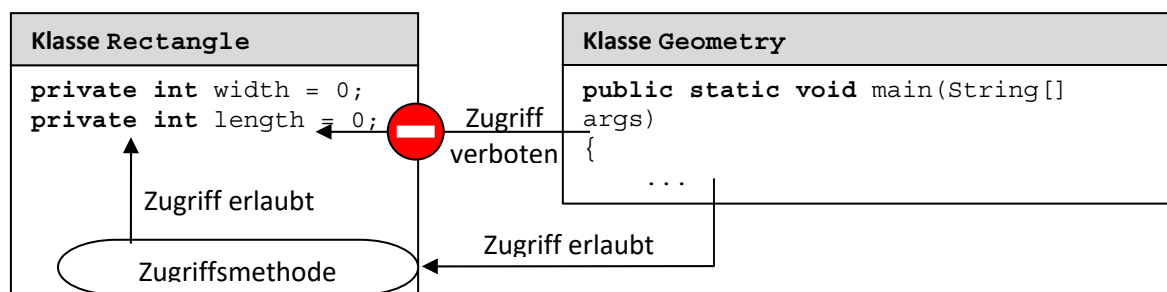
[modifiers] type identifier([, ...])
{
    ...
}
```

Bei den bisherigen Beispielen haben Sie keine Zugriffsmodifizierer angegeben, das bedeutet, dass Sie aus allen anderen Klassen, die im selben Ordner (dem sogenannten selben Package) gespeichert sind, auf die Attribute der Klasse zugreifen und diese verändern und auslesen können.

Über die Zugriffsmodifizierer lassen sich die Zugriffsrechte ausweiten (auf Klassen in anderen Ordnern) oder einschränken (beispielsweise auf die Klasse selbst).

Attribute mit dem Zugriffsmodifizierer **private** kapseln

Um eine Kapselung der Attribute zu erreichen, „privatisieren“ Sie die Attribute mit dem Zugriffsmodifizierer **private**. Die so gekennzeichneten Attribute sind nur innerhalb der Klasse selbst sichtbar, und der Zugriff ist so weit eingeschränkt, dass nur Methoden der Klasse selbst darauf zugreifen können. Um die Attribute aus anderen Klassen heraus verändern oder auslesen zu können, definieren Sie als Schnittstelle spezielle Zugriffsmethoden, die im folgenden Abschnitt erläutert werden.



7.2 Zugriffsmethoden

Mit Getter-Methoden Eigenschaften lesen

Durch die Kapselung eines Attributs mit dem Zugriffsmodifizierer **private** wird der direkte Zugriff von außen auf das Attribut verhindert. Um den Wert einer gekapselten Eigenschaft auslesen zu können, benötigen Sie eine spezielle Methode, die die gewünschten Informationen liefert. Diese Methode wird als **Getter-Methode** bzw. als Accessor (Access = Zugriff) bezeichnet.

```
class Rectangle
{
    private int width = 0;
```



```
private int length = 0;
...
int getWidth()
{
    return width;
}
int getLength()
{
    return length;
}
...
```

- ✓ Der Name der Methode wird gewöhnlich aus dem Wort `get` („get“ = erhalten) und dem Namen des Attributs gebildet, der dann mit einem Großbuchstaben beginnt.
- ✓ Eine Ausnahme bilden Attribute des Datentyps `boolean`. Den Namen für Getter-Methoden auf diese Attribute stellen Sie das Wort „is“ voran. Die Methode heißt beispielsweise `isFilled`.

Mit Setter-Methoden Eigenschaften ändern

Soll der Wert des Attributs eines Objekts der Klasse verändert werden können, definieren Sie dafür eine entsprechende Methode. Diese Methode wird als **Setter-Methode** bzw. als Mutator (Mutation = Veränderung) bezeichnet.

Ähnlich den Getter-Methoden setzt sich der Name einer Setter-Methode üblicherweise aus dem Wort `set` („set“ = setzen) und dem Namen des Attributs, der dann mit einem Großbuchstaben beginnt, zusammen.

Eine Setter-Methode besitzt einen Parameter, mit dem der künftige Wert des Attributs mitgeteilt wird. Als Name des Parameters wird gewöhnlich ebenfalls der Attributname verwendet. So heißt in der Methode `setLength` der Parameter ebenfalls `length`. Da die Methode aber zwischen den beiden Variablen mit dem Namen `length` unterscheiden muss, wird die Referenzvariable `this` verwendet. Sie besagt, dass das Attribut gemeint ist.

Die Referenzvariable `this`

Mit jedem Objekt wird vom Compiler automatisch eine Referenzvariable auf das eigene Objekt erzeugt, die den Namen `this` trägt. Diese Referenzvariable kann in allen Methoden des Objekts eingesetzt werden. `this` ist ein Schlüsselwort und kann deshalb nicht in Ihren eigenen Programmen als Bezeichner eingesetzt werden.

Die Referenzvariable `this` existiert (nur) für jedes erzeugte Objekt. Da statische Methoden nicht an Objekte gebunden sind, kann in statischen Methoden die Referenzvariable `this` nicht verwendet werden.

Setter-Methoden erstellen

```
class Rectangle
{
    private int width = 0;
    private int length = 0;
    ...
    ① void setWidth(int width)
    {
        this.width = width; ②
    }

    void setLength(int length)
    {
        this.length = length;
    }
    ...
}
```

- ✓ Im Beispiel bezeichnet der Ausdruck `this.width` das Attribut `width` der Klasse `Rectangle` ①.
- ✓ Der Ausdruck `width` (ohne `this`) entspricht der Variablen, die als Parameter übergeben wird ②.

Konsequent mit Getter-und Setter-Methoden arbeiten

Sie sollten immer die entsprechenden Setter- und Getter-Methoden verwenden, auch wenn Sie aus anderen Methoden der Klasse auf die Attribute zugreifen möchten. Dies hat folgende Vorteile:

- ✓ Sofern spezielle Gültigkeitsregeln für ein Attribut gelten, müssen Sie diese nur einmal in der Setter-Methode berücksichtigen.
- ✓ Eine eventuell erforderliche Bearbeitung eines Wertes vor dem Speichern oder nach dem Auslesen eines Attributs muss nur einmal in der jeweiligen Getter- bzw. Setter-Methode programmiert werden.
- ✓ Änderungen beim Schreiben (Speichern) und Auslesen von Attributen müssen ebenfalls nur in den Zugriffsmethoden vorgenommen werden. Zugriffsmethoden senken somit den Wartungsaufwand.
- ✓ Bei fehlerhafter Ausführung des Programms lassen sich zur Überwachung Ausgabeanweisungen in die Getter- und Setter-Methoden einfügen. So kann zur Fehlersuche der Zugriff auf die Attribute kontrolliert werden.

7.3 Konstruktoren

Was ist ein Konstruktor?

Ein Konstruktor ist eine spezielle Methode, die automatisch beim Erzeugen eines Objekts ausgeführt wird. Java hält standardmäßig für jede Klasse, die Sie definieren, einen Standardkonstruktor bereit. Dieser Konstruktor wird aufgerufen, wenn Sie mit `new` ein neues Objekt erstellen.

```
...  
Rectangle specialRect = new Rectangle();  
...  
①          ②          ③
```

Objekt erzeugen und zuweisen

- ✓ Der Konstruktoraufruf ② zeigt, dass der Konstruktor den gleichen Namen wie die Klasse ① besitzt.
- ✓ Wie eine Methode besitzt der Standardkonstruktor Klammern, es werden aber keine Parameter ③ übergeben.

Einen individuellen Konstruktor erstellen

In vielen Fällen ist es notwendig, dass beispielsweise Initialisierungen durchgeführt werden, wenn ein Objekt erzeugt wird. Dazu programmieren Sie einen individuellen Konstruktor, der als Anweisungen die gewünschten Initialisierungen enthält. Die Beschreibung eines Konstruktors orientiert sich an folgenden Regeln:

```
class Rectangle ①  
{  
    private int width = 0;  
    private int length = 0;  
    ...  
    Rectangle() ②  
    {  
        width = 1;  
        length = 1;  
    }  
    ...  
}
```

- ✓ Ein Konstruktor ist eine spezielle Methode ②, die den gleichen Namen wie die Klasse ① besitzt. Im Gegensatz zu den bisher bekannten Methoden beginnt der Konstruktorname wie der Klassenname mit einem Großbuchstaben.
- ✓ Er liefert **keinen** Wert zurück, es wird aber auch nicht das Schlüsselwort `void` verwendet.



Sobald Sie einen Konstruktor erstellen, existiert der Standardkonstruktor nicht mehr.

Ein Objekt mit dem neuen Konstruktor erzeugen

Der folgende Programmcode zeigt, wie Sie den neuen Konstruktor anwenden:

```
...
// ein Rechteck mit dem neuen Konstruktor erzeugen
Rectangle specialRect = new Rectangle();
...
```

Den Konstruktor anwenden

Ein Konstruktor kann nicht wie eine Methode aufgerufen werden, sondern dies geschieht automatisch beim Erzeugen eines Objekts der Klasse. Eine Ausnahme ist der im nachfolgenden Abschnitt beschriebene Konstruktoraufwurf innerhalb eines anderen Konstruktors.

Mehrere individuelle Konstruktoren bereitstellen

Wie andere Methoden können Sie auch Konstruktoren überladen. Sie haben so die Möglichkeit, verschiedene Konstruktoren bereitzustellen, mit denen Sie ein Objekt erzeugen und initialisieren können. Java unterscheidet die Konstruktoren anhand der Signatur, d. h. an dem Namen und der Anzahl und dem Datentyp der Parameter.

```
class Rectangle
{
    private int width = 0;
    private int length = 0;
    ...
    Rectangle()
    {
        setWidth(1);
        setLength(1);
    }
    Rectangle(①int width, int length)
    {
        setWidth(width);
        setLength(length); ②
    }
    ...
}
```

- ✓ Der Konstruktor kann in Klammern `()` Parameter enthalten ^①, die Sie bei der Erzeugung eines Objektes beispielsweise zur Initialisierung der Attribute verwenden können ^②.
- ✓ Für die Initialisierung von Attributen sollten Sie entsprechende Setter-Methoden nutzen, sofern Sie diese definiert haben ^②.

Die verschiedenen Konstruktoren anwenden

Beim Erzeugen eines Objekts mit `new` fügen Sie gegebenenfalls in Klammern die betreffenden Parameter ein.

```
class Geometry
{
    ...
    // ein Rechteck mit der Breite 1 und der Länge 1 erzeugen
    Rectangle specialRect = new Rectangle();

    // ein Rechteck mit der Breite 3 und der Länge 5 erzeugen
    Rectangle anotherRect = new Rectangle(3, 5);
    ...
}
```

Konstruktoren anwenden

- ✓ In diesem Beispiel können Sie den Konstruktor ohne Parameter aufrufen.
- ✓ Oder Sie verwenden den Konstruktor mit zwei Parametern, mit denen Sie automatisch die Breite und Länge mit den gewünschten Werten initialisieren können.



Deklarieren Sie immer einen parameterlosen Konstruktor, wenn Sie eigene Konstruktoren erstellen. Dies ist für die Vererbung von Klassen notwendig.

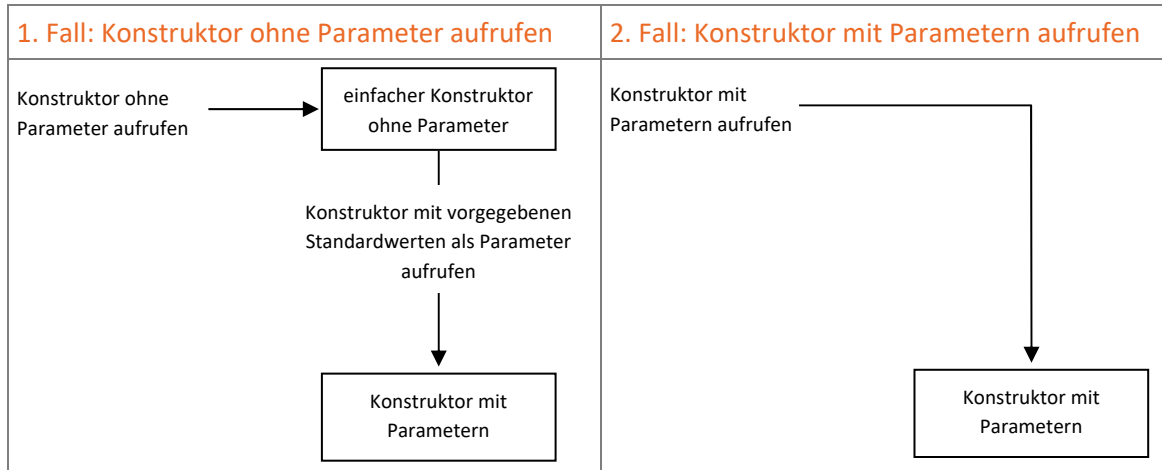
Ein Konstruktor mit einem Parameter ist in der Klasse `Rectangle` nicht definiert. Daher würde eine Anweisung wie beispielsweise `Rectangle thirdRect = new Rectangle(4);` eine Fehlermeldung hervorrufen.

Konstruktoren verketteten

Bei dem zuvor genannten Beispiel ist in den einzelnen Konstruktoren ähnlicher Programmcode mehrfach vorhanden. Häufig wird in Konstruktoren aber nicht nur die Initialisierung der Attribute, sondern werden beispielsweise auch Gültigkeitsprüfungen vorgenommen, sodass mehrfach aufgeführter Programmcode besonders bei umfangreichen Programmen einen erhöhten Wartungsaufwand bewirkt. Daher sollten Sie bestrebt sein, Programmcode wiederzuverwenden.

Durch die sogenannte Verkettung von Methoden können Sie bei der Beschreibung eines Konstruktors einen anderen Konstruktor aufrufen und so dessen Programmcode übernehmen.

In dem folgenden Beispiel mit zwei Konstruktoren dient der einfache Konstruktor lediglich dazu, den zweiten Konstruktor mit Parametern (Standardwerte) aufzurufen.



Dies bedeutet, dass der überwiegende Anteil des Programmcodes lediglich in dem Konstruktor vorhanden ist, der mit Parametern aufgerufen wird.

Beispiel: *Rectangle.java* und *Geometry.java*

Das in Java umgesetzte Beispiel besitzt drei Konstruktoren.

```

class Rectangle
{
    private int width = 0;
    private int length = 0;
    ...
    ① Rectangle()
    {
        ② this(1, 1); //Konstruktoraufruf: Rechteck mit Breite=1 und
                     //Länge=1
    }
    ③ Rectangle(int width) //Quadrat: Breite und Länge sind gleich
    {
        this(width, width); //Konstruktoraufruf: Breite=Länge
    }
    ④ Rectangle(int width, int length) //allgemeines Rechteck
    {
        setWidth(width);
        setLength(length);
        //weitere Anweisungen
    }
}

```

Konstruktoren verketteten

- ① Der einfache Konstruktor ohne Parameter legt Standardwerte fest und verwendet diese als Parameter für den Aufruf des allgemeinen Konstruktors ④.
- ② Der Aufruf eines anderen Konstruktors muss als **erste** Anweisung stehen. Der Aufruf erfolgt über das Schlüsselwort **this** und die entsprechende Parameterliste des Konstruktors.
- ③ Weitere Konstruktoren wie hier mit einem Parameter können für die komfortable Erstellung von Objekten der Klasse nützlich sein.

```
class Geometry
{
    public static void main(String[] args)
    {
        Rectangle specialRect = new Rectangle();
        Rectangle anotherRect = new Rectangle(3, 5);
        System.out.printf("Das Rechteck (%d x %d)%n",
                           specialRect.getWidth(), specialRect.getLength());
        System.out.printf("hat den Flächeninhalt %d%n",
                           specialRect.getArea());
        System.out.printf("Das Rechteck (%d x %d)%n",
                           anotherRect.getWidth(), anotherRect.getLength());
        System.out.printf("hat den Flächeninhalt %d%n",
                           anotherRect.getArea());
    }
}
```


Konstruktoren anwenden

```
Das Rechteck (1 x 1)
hat den Flächeninhalt 1
Das Rechteck (3 x 5)
hat den Flächeninhalt 15
```

Die Ausgabe des Programms

7.4 Übung

Getter- und Setter-Methoden erstellen, Konstruktoren deklarieren

Level		Zeit	ca. 30 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Kapseln von Attributen ✓ Erstellen und verwenden von Getter- und Setter-Methoden ✓ Deklarieren von Konstruktoren ✓ Verwenden unterschiedlicher Konstruktoren 		
Übungsdateien	<i>SomeMaths.java, Circle.java, CircleExercise.java (alle aus ..\Kap06)</i>		
Ergebnisdateien	<i>SomeMaths.java, Circle.java, CircleExercise.java</i>		

1. Öffnen Sie die in Kapitel 6 erstellte Quelltextdatei mit der Klasse `Circle` zur Bearbeitung im Editor. (Die Klasse `Circle` verwendet die Klasse `SomeMaths.java`, die ebenfalls in Kapitel 6 erstellt wurde.)
2. Ergänzen Sie ein Attribut `filled` mit logischem Datentyp.
3. Kapseln Sie die Attribute.
4. Erstellen Sie für alle Attribute der Klasse `Circle` Setter- und Getter-Methoden, mit denen Sie einen Zugriff auf die Attribute aus einer anderen Klasse heraus ermöglichen (lesen und ändern).
5. Formulieren Sie in der Klasse `Circle` drei verschiedene Konstruktoren zur Erzeugung von Objekten auf Basis dieser Klasse:

Sofern als Parameter keine Werte übergeben werden, soll der Radius standardmäßig den Wert 1 erhalten und der Kreis soll nicht gefüllt sein:

- ✓ Konstruktor ohne Parameter;
- ✓ Konstruktor, bei dem der Radius angegeben wird;
- ✓ Konstruktor, der die Angaben des Radius und die Information, ob der Kreis gefüllt sein soll, entgegennimmt.

Verwenden Sie dabei, soweit möglich, die Getter- und Setter-Methoden.

6. Öffnen Sie im Editor die in Kapitel 6 erstellte Klasse `CircleExercise` zur Bearbeitung.
7. Verändern Sie den Quelltext so, dass die neuen Konstruktoren verwendet werden.
8. Korrigieren Sie die Ausgabeanweisungen.
9. Ergänzen Sie für jedes Kreisobjekt eine Ausgabezeile, in der Sie die Eigenschaft `filled` mit einer `if`-Struktur auswerten und einen entsprechenden Text ausgeben, beispielsweise für den ersten Kreis:

1. Kreis ist gefuehlt.

bzw. 1. Kreis ist nicht gefuehlt.

```

1. Kreis:
Radius: 3.00000
Umfang: 18.8495
Flaeche: 28.2743
1. Kreis ist gefuehlt.

2.Kreis:
Radius: 8.00000
Umfang: 50.2654
Flaeche: 0.00000
2. Kreis ist nicht
gefuehlt.

3. Kreis:
Radius: 5.00000
Umfang: 31.4159
Flaeche: 78.5398
3. Kreis ist gefuehlt.

```

Die Ausgabe des Programms

8

Vererbung

8.1 Grundlagen zur Vererbung

Was ist Vererbung?

Die **Vererbung** (engl. inheritance), auch **Ableitung** genannt, ist neben der Kapselung ein weiteres wichtiges Merkmal objektorientierter Programmiersprachen.

- ✓ Über Vererbung erzeugen Sie eine neue Klasse auf Basis einer bereits vorhandenen Klasse.
- ✓ Die neue **abgeleitete Klasse** übernimmt dabei automatisch die Eigenschaften und Methoden der **übergeordneten Klasse**, der sogenannten **Basisklasse**.
- ✓ Sie können zu den geerbten Eigenschaften und Methoden weitere hinzufügen oder geerbte Methoden ändern. Die abgeleitete Klasse ist dann gegenüber der Basisklasse erweitert.

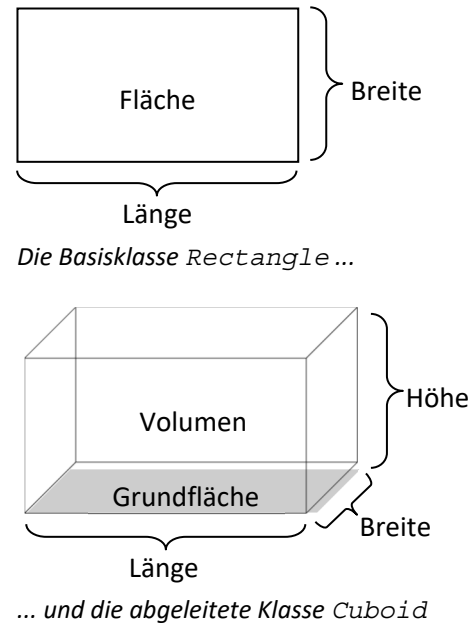
Für Basisklasse können auch die Begriffe **Superklasse** oder übergeordnete Klasse verwendet werden. Statt abgeleiteter Klasse können Sie die Begriffe **Subklasse**, untergeordnete Klasse oder Unterklasse verwenden.

Beispiel zur Vererbung

Im Beispiel speichert die Klasse `Rectangle` die Breite und Länge eines Rechtecks. Über die Methoden `getWidth` und `getLength` bzw. `setWidth` und `setLength` können die Breite und die Länge ermittelt bzw. geändert werden. Mit der Methode `getArea` berechnen Sie die Fläche des Rechtecks.

Von der Klasse `Rectangle` soll eine Klasse `Cuboid` (Quader) abgeleitet werden. Die Klasse `Rectangle` ist die Basisklasse. Die abgeleitete Klasse `Cuboid` enthält ein zusätzliches Attribut `height` (Höhe) und definiert weitere Methoden:

- ✓ Setter- und Getter-Methoden ermöglichen den Zugriff auf das Attribut `height`.
- ✓ Die Methode `getVolume` berechnet das Volumen, indem die Grundfläche mit der Höhe multipliziert wird.



8.2 Klassen ableiten und erweitern

Syntax für das Ableiten von Klassen

```
[modifiers] class identifier extends superClassName
{
    statements
}
```

① ②

Syntax für das Ableiten einer Klasse

- ✓ Sie beginnen die Klassendefinition mit dem Schlüsselwort `class` und dem Klassennamen.
- ✓ Über das Schlüsselwort `extends` ① geben Sie an, dass Sie die Klasse (`identifier`) von einer anderen Klasse ableiten wollen.
- ✓ Nach `extends` geben Sie den Klassennamen der Basisklasse an ②. Die abgeleitete Klasse erbt alle Elemente dieser Klasse, die nicht mit dem Zugriffsmodifizierer `private` als privat deklariert sind.

In Java gibt es keine Mehrfachvererbung: Es kann immer nur eine Klasse als Basisklasse dienen.

Beispiel: Klassen ableiten und erweitern: *Cuboid.java*

Der abgebildete Quelltext zeigt die Klasse *Cuboid*, die von der Klasse *Rectangle* abgeleitet wird.

- ✓ Die Klasse besitzt ein zusätzliches Attribut *height*, das wie die Attribute der Basisklasse mit *private* gekapselt ist. ①
- ✓ Die Getter-Methode *getHeight* und die Setter-Methode *setHeight* ermöglichen den Zugriff auf das Attribut *height*. ②
- ✓ Für die Berechnung des Volumens nutzt die Methode *getVolume* ③ die Methoden *getWidth* und *getLength* der Basisklasse und die Methode *getHeight* der abgeleiteten Klasse selbst.

```
class Cuboid extends Rectangle
{
    private int height;①

    int getHeight()
    {
        return height;
    }

    void setHeight(int height) ②
    {
        this.height = height;
    }

    int getVolume()③
    {
        return getWidth()
            * getLength()
            * getHeight();
    }
    ...
}
```

Die abgeleitete Klasse *Cuboid* (Quader)

In abgeleiteten Klassen auf gekapselte Attribute der Basisklasse zugreifen

Da die Attribute in der Basisklasse mit dem Zugriffsmodifizierer *private* versehen wurden, können Sie darauf in der abgeleiteten Klasse nicht direkt zugreifen.

So ist die Berechnung des Volumens mit dem folgenden Ausdruck nicht möglich:

```
width * length * height
```

Stattdessen müssen Sie die Berechnung mithilfe der entsprechenden Getter-Methoden ③ formulieren:

```
getWidth() * getLength() * getHeight()
```

Den Zugriffsmodifizierer *protected* einsetzen

Durch den Zugriffsmodifizierer *private* sind Attribute bzw. Methoden gekapselt, und ein Zugriff darauf ist nur innerhalb der Klasse erlaubt. Um den direkten Zugriff auf diese Attribute bzw. Methoden aus einer abgeleiteten Klasse zu ermöglichen, kann der Zugriffsschutz für die gekapselten Attribute gelockert werden, indem statt *private* der Zugriffsmodifizierer *protected* eingesetzt wird.

Alle Klassen, die sich im selben Ordner befinden, erhalten ebenfalls Zugriff auf Attribute und Methoden, die mit dem Modifizierer *protected* ausgestattet sind.

Viele Programmierer bevorzugen es, Attribute vollständig mit `private` zu kapseln und den Zugriff lediglich über Getter- und Setter-Methoden zu gestatten.

8.3 Konstruktoren aufrufen

Objekte von abgeleiteten Klassen erzeugen

Mit den individuellen Konstruktoren werden häufig umfangreiche Initialisierungen vorgenommen (im Beispiel ist dies lediglich die Initialisierung der Attribute `width` und `length`). Es muss sichergestellt werden, dass diese Initialisierungen auch erfolgen, wenn von der abgeleiteten Klasse Objekte erzeugt werden.

Über Vererbung werden nur Methoden und Attribute vererbt, aber keine Konstruktoren. Es steht somit zunächst nur der Standardkonstruktor zur Verfügung. Sie müssen in den abgeleiteten Klassen eigene Konstruktoren definieren, um die gewünschten Initialisierungen vorzunehmen.

Beispiel zur Nutzung des Standardkonstruktors der abgeleiteten Klasse:
Rectangle.java, Cuboid.java, CreateCuboids.java

Bisher besitzt die abgeleitete Klasse `Cuboid` nur den Standardkonstruktor. Um zu zeigen, welche Konstruktoren der Basisklasse bei der Erzeugung eines Objektes der Klasse `Cuboid` aufgerufen werden, wurden hier zwei Ausgabeanweisungen ① und ② in der Basisklasse eingefügt.

```
...
Rectangle()
{
    this(1, 1);
    System.out.println("ausfuehren: Rectangle()"); ①
}

Rectangle(int width, int length)
{
    ...
    System.out.println("ausfuehren: Rectangle(a, b)"); ②
}
...
```

Ausgabeanweisungen in den Konstruktoren der Klasse `Rectangle`

Ausgabeanweisungen eignen sich dazu, den Ablauf eines Programms zu verfolgen oder durch die Ausgabe von Variableninhalten Fehler aufzuspüren.

```
class Cuboid extends Rectangle
{
    private int height;
    ...
}
```

Die abgeleitete Klasse `Cuboid`

```
class CreateCuboids
{
    public static void main(String[] args)
    {
        Cuboid firstCuboid = new Cuboid();
    }
}
```

Die Klasse *CreateCuboid* mit der *main*-Methode

Da die Klasse *Cuboid* bisher keinen speziellen Konstruktor enthält, wird automatisch der Standard-Konstruktor aufgerufen, der wiederum den ohne Parameter definierten Konstruktor der Basisklasse ausführt ②. Durch die Weiterleitung der Konstrukturen in der Basisklasse *Rectangle* wird zuvor der Konstruktor mit zwei Parametern ausgeführt ①.

```
ausfuehren: Rectangle(a, b) ①
ausfuehren: Rectangle()      ②
```

Die Ausgabe des Programms

Konstrukturen der Basisklasse werden nicht vererbt. So führt die folgende Anweisung zu einer Fehlermeldung, denn die Klasse *Cuboid* definiert bisher keine Konstrukturen:

```
Cuboid FirstCuboid = new Cuboid(3, 4)
```

Konstrukturen für abgeleitete Klassen erstellen

Die Konstrukturen in abgeleiteten Klassen werden dazu genutzt, zusätzlich hinzugefügte Attribute zu initialisieren und spezielle Festlegungen für die abgeleitete Klasse zu treffen. Die Konstrukturen der Basisklasse können dabei verwendet werden. Der Aufruf eines Konstruktors der Basisklasse erfolgt über das Schlüsselwort *super*.

Syntax zur Verwendung eines Konstruktors der Basisklasse

- ✓ Über das Schlüsselwort *super* können Sie den Konstruktor der Basisklasse aufrufen. Auf diese Weise können Sie jeden beliebigen Konstruktor der Basisklasse verwenden.
- ✓ Der Aufruf von *super* muss dazu als erste Anweisung im Konstruktor stehen.
- ✓ Wenn Sie keinen Konstruktor der Basisklasse explizit mit dem Schlüsselwort *super* aufrufen, wird automatisch der Standardkonstruktor der Basisklasse als erste „unsichtbare“ Anweisung aufgerufen.
- ✓ Sie können nur den Konstruktor der direkten Basisklasse aufrufen. Konstrukte der Form *super.super()* sind **nicht zulässig**.

```
super ( [argument, ...] );
```

Um sicherzustellen, dass die geerbten Attribute richtig initialisiert sind, sollten Sie immer explizit einen geeigneten Konstruktor der Basisklasse aufrufen.

Beispiel zur Verwendung eines Konstruktors der Basisklasse: *Cuboid.java*

```

class Cuboid extends Rectangle
{
    private int height;
    ...
① Cuboid()                                //einfacher Konstruktor
    {
②     this(1, 1, 1);                      //Konstruktorweilerschaltung
    }
③ Cuboid(int width, int length, int height) //Konstruktor mit
                                           //Parametern
    {
④     super(width, length); //Konstruktor der Basisklasse nutzen
⑤     setHeight(height);
    }
    ...
}

```

Die abgeleitete Klasse *Cuboid*

- ① Der einfache Konstruktor (ohne Parameter) ruft den Konstruktor mit Parametern auf und verwendet die Zahl 1 als Standardwert für die Breite, die Länge und die Höhe ②.
- ③ Der Konstruktor mit Parametern ruft den Konstruktor der Basisklasse auf ④ und initialisiert anschließend das Attribut *height* (Höhe) ⑤, da dies in der Basisklasse nicht enthalten ist.

8.4 Geerbte Methoden überschreiben

Geerbte Methoden verwenden

Methoden, die nicht als `private` deklariert wurden, werden bei der Ableitung vererbt. Die Klasse *Cuboid* wurde von der Klasse *Rectangle* abgeleitet. Sie erbt daher auch deren Methoden, wie beispielsweise die Methoden `getWidth` und `getLength`. Sie können die Methoden auch in der Klasse *Cuboid* ① bzw. für Objekte der Klasse ② verwenden.

```

class Cuboid extends Rectangle
{
    ...
    int getVolume()
    {
        return getWidth() * getLength() * getHeight();
    }
    ①
}

```

Die geerbten Methoden in der Klasse *Cuboid* verwenden

```

class UsingCuboid
{
    public static void main(String[] args)
    {
        Cuboid oneCuboid = new Cuboid(3, 5, 7);
        System.out.printf("Quaderbreite: %d%n", oneCuboid.getWidth());
    }
}

```

②

Mit Objekten der Klasse *Cuboid* arbeiten

Overriding – geerbte Methoden überschreiben

Wenn Sie eine Methode einer Basisklasse in einer abgeleiteten Klasse redefinieren, überschreiben (verdecken) Sie die von der Basisklasse geerbte Methode. Die Methode bleibt für die Basisklasse aber in der ursprünglichen Form erhalten.

Die neue Methode muss die gleiche Signatur haben, d. h., sie muss ...

- ✓ denselben Namen besitzen,
- ✓ die gleichen Parameter verwenden,
- ✓ den gleichen (oder einen davon abgeleiteten) Datentyp als Rückgabewert verwenden.

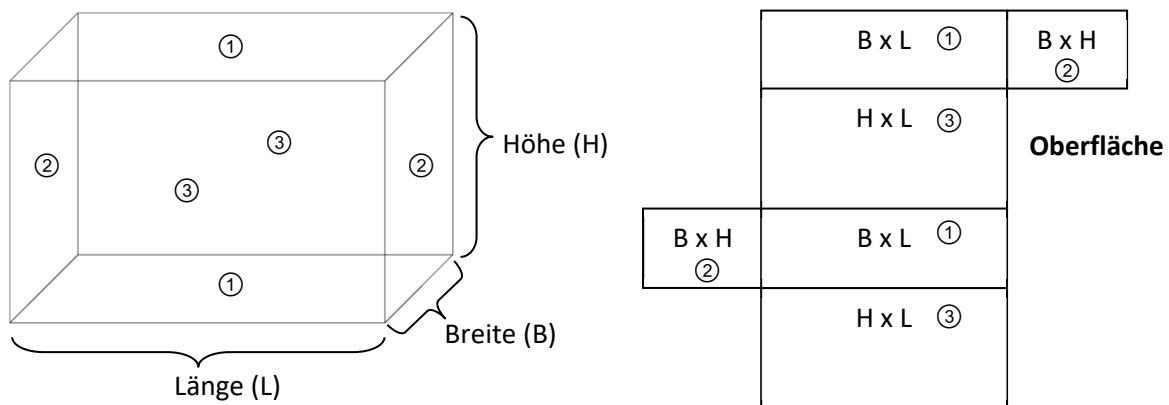
Die Zugriffsrechte dürfen Sie beim Überschreiben einer Methode erweitern. Beispielsweise können Sie eine Methode, die in der Basisklasse als `protected` gekennzeichnet wurde, beim Überschreiben mit dem Zugriffsmodifizierer `public` als öffentlich kennzeichnen. Eine Einschränkung der Zugriffsrechte ist nicht erlaubt.

Beispiel für das Überschreiben geerbter Methoden

Die Klasse *Cuboid* hat die Methode `getArea` von der Klasse *Rectangle* geerbt. Diese Methode berechnet den Flächeninhalt des Rechtecks mit der Formel:

$$\text{Flächeninhalt} = \text{Breite} \times \text{Länge}$$

Sie möchten auch in der Klasse *Cuboid* die Fläche berechnen – die Oberfläche des Quaders. Die Oberfläche eines Quaders setzt sich aus den sechs Seitenflächen zusammen:



Die Oberfläche eines Quaders ermitteln

Die Formel für die Berechnung der Oberfläche lautet daher:

$$\text{Oberfläche} = 2 \times (\underbrace{\text{Breite} \times \text{Länge}}_{\textcircled{1}} + \underbrace{\text{Breite} \times \text{Höhe}}_{\textcircled{2}} + \underbrace{\text{Höhe} \times \text{Länge}}_{\textcircled{3}})$$

Die Methode `getArea` wird entsprechend dieser Formel in der Klasse `Cuboid` neu formuliert.

```
class Cuboid extends Rectangle
{
    ...
    int getArea()
    {
        return 2 * (getWidth() * getLength() ①
                    + getWidth() * getHeight() ②
                    + getHeight() * getLength()); ③
    }
    ...
}
```

Die abgeleitete Klasse Cuboid (Quader)

Private, finale und statische Methoden einer Basisklasse können nicht überschrieben werden. Private Methoden sind in der abgeleiteten Klasse nicht sichtbar (es ist kein Aufruf von `super.methode()` möglich). Finale und statische Methoden dürfen in abgeleiteten Klassen nicht überschrieben werden. Sie erhalten in diesem Fall einen Compiler-Fehler.

Die Methoden der Basisklasse verwenden

Die Basisklasse von `Cuboid` beschreibt das Rechteck der Grundfläche. Die von der Basisklasse `Rectangle` geerbte Methode `getArea` ermittelt daher den Flächeninhalt **der Grundfläche**.

Die Klasse `Cuboid` redefiniert die Methode `getArea`. Über das Schlüsselwort `super` haben Sie jedoch weiterhin die Möglichkeit, die entsprechende Methode der Basisklasse aufzurufen. Der Compiler kann so zwischen der Methode der Basisklasse und der Methode der Klasse selbst unterscheiden.

```
class Cuboid extends Rectangle
{
    ...
    int getBase()
    {
        return super.getArea(); ①
    }
}
```

Methoden der Basisklasse verwenden

In der Klasse `Cuboid` möchten Sie hier eine spezielle Methode `getBase` erstellen, mit der die Grundfläche des Quaders ermittelt wird. Sie nutzen die Methode der Basisklasse und verwenden dazu ebenfalls das Schlüsselwort `super` ①.

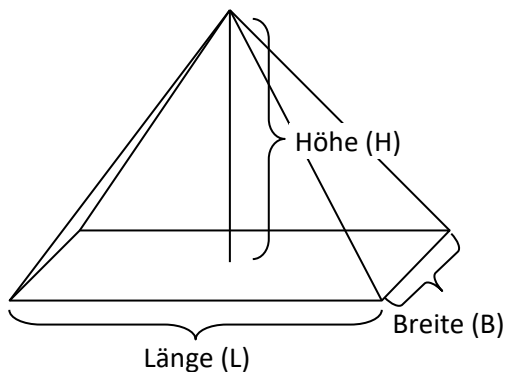
Durch das Überschreiben von Methoden können abgeleitete Klassen ein spezielles Verhalten implementieren und haben dennoch Zugriff auf die Methode der Basisklasse. Benötigt die abgeleitete Klasse bei einer Methode keine spezielle Funktionalität, wird die Methode der Basisklasse verwendet.

```
class UsingCuboid
{
    public static void main(String[] args)
    {
        Cuboid oneCuboid = new Cuboid(3, 5, 7);
        System.out.printf("Oberflaeche: %d%n",
                           oneCuboid.getArea());
        System.out.printf("Grundflaeche: %d%n",
                           oneCuboid.getBase());
    }
}
```

Die Klasse Cuboid anwenden

Beispiel für eine weitere abgeleitete Klasse: *Pyramid.java*

Ähnlich der Klasse Cuboid lässt sich von der Klasse Rectangle eine weitere Klasse ableiten – die Klasse Pyramid.



Eine Pyramide

- ① Die Klasse Pyramid erhält ein zusätzliches Attribut height.
- ② Die Berechnung der Oberfläche einer Pyramide ist nicht ganz einfach und soll hier nicht erfolgen. Die Methode getArea soll zur Kennzeichnung den Wert -1 liefern ③.
- ④ Die Berechnung des Volumens erfolgt nach der Formel:

$$\text{Volumen} = 1/3 * \text{Grundfläche} * \text{Höhe}$$

```
class Pyramid extends Rectangle
{
    ① private int height;
    ...
    ② int getArea()
    {
        ③ return -1;
    }

    int getVolume()
    ④ {
        return getWidth() * getLength()
           * getHeight() / 3;
    }
}
```

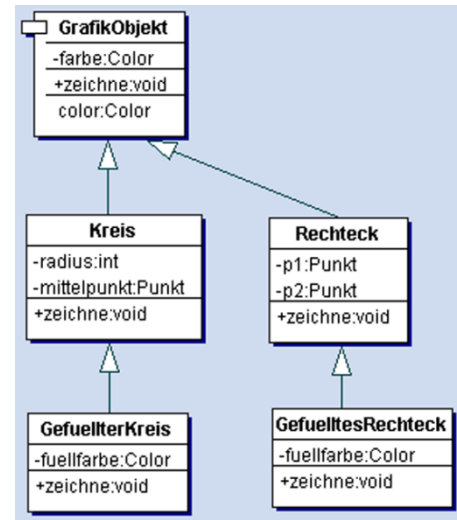
8.5 Vererbungsketten und Zuweisungskompatibilität

Eine Klassenhierarchie bilden

Von einer Klasse können Klassen abgeleitet werden, von denen wiederum Klassen abgeleitet werden. So entstehen Vererbungsketten, die eine Klassenhierarchie bilden.

Die nebenstehende Grafik zeigt eine mögliche Klassenhierarchie für Grafikobjekte eines Zeichenprogramms.

- ✓ Die Klassen `Kreis` und `GefuellterKreis` erben beispielsweise von `GrafikObjekt` das Attribut `farbe`.
- ✓ Die Klassen `Kreis` und `GefuellterKreis` definieren aber jeweils selbst eine eigene Methode `zeichne`.
- ✓ Außerdem fügen sie neue Attribute wie `radius` und `mittelpunkt` hinzu.



In einer Vererbungshierarchie besteht zwischen den einzelnen Ebenen immer zwischen der abgeleiteten und der Superklasse eine 'is a' ('ist ein') Beziehung. In der vorherigen Abbildung ist beispielsweise ein `Kreis` (Klasse `Kreis`) ein grafisches Objekt (Klasse `GrafikObjekt`) und ein gefülltes Rechteck (Klasse `GefuelltesRechteck`) ein Rechteck (Klasse `Rechteck`). Die 'is a' Beziehung beschränkt sich dabei nicht auf eine Ebene. So ist auch ein gefülltes Rechteck (Klasse `GefuelltesRechteck`) ein grafisches Objekt (Klasse `GrafikObjekt`). Zwischen den einzelnen Vererbungslinien besteht diese Beziehung jedoch nicht. So ist ein gefüllter Kreis (Klasse `GefuellterKreis`) kein Rechteck (Klasse `Rechteck`), da zwischen diesen Klassen keine Vererbungsbeziehung besteht.

Im Unterschied zur 'is a' Beziehung steht die 'has a' ('hat ein') Beziehung für eine Beziehung zwischen zwei Klassen, welche nicht auf Vererbung basiert. Bei einer 'has a' Beziehung besitzt eine Klasse ein Instanz-Attribut, welches auf eine andere Klasse typisiert ist und somit Referenzen auf diese verwalten kann. Im obigen Beispiel besitzen die Klassen `Kreis` und `Rechteck` Attribute, welche auf die Klasse `Punkt` (nicht in der Abbildung dargestellt) typisiert sind. Sie besitzen (haben) einen oder mehrere Punkte, zum Beispiel hat ein `Kreis` einen Mittelpunkt. Es besteht jedoch keine Vererbungsbeziehung zwischen den Klassen `Kreis` und `Punkt` bzw. `Rechteck` und `Punkt`.

Kompatibilität von Objekten einer Klasse und Objekten der Basisklasse

Innerhalb der Klassenhierarchie sind Objekte abgeleiteter Klassen immer zu Objektvariablen der Basisklassen zuweisungskompatibel. Der Grund liegt darin, dass diese Objekte immer mindestens die Methoden und Attribute der Basisklasse besitzen und somit vollständig als ein solches Objekt funktionstüchtig sind.

Beispiel

Einer Referenzvariablen vom Typ `Rectangle` wird ein Objekt vom Typ `Cuboid` zugewiesen. Da der Quader auch die Methoden des Rechtecks (`Rectangle`) besitzt, ist die Zuweisung erlaubt.

```
Rectangle_firstForm = new Cuboid();
```

Explizite Typkonversion

Wenn Sie einer Referenzvariablen einer abgeleiteten Klasse ein Objekt der Basisklasse zuweisen, fehlen die Methoden, die erst in der abgeleiteten Klasse definiert wurden.

```
class TypeCast
{
    public static void main(String[] args)
    {
        Rectangle firstForm = new Cuboid();
        Rectangle secondForm = new Rectangle();
        ① Cuboid oneCuboid = (Cuboid)firstForm;
        //kein Compilerfehler, kein Laufzeitfehler
        ② Cuboid anotherCuboid = (Cuboid)secondForm;
        //kein Compilerfehler, aber Laufzeitfehler
    }
}
```

- ① Durch die explizite Typkonversion (**cast**) ③ können Sie die Umwandlung des Objekts vom Typ `Rectangle` in einen Typ `Cuboid` **erzwingen**. Es entsteht kein Compilerfehler und auch kein Fehler während der Laufzeit, da die Referenzvariable `firstForm` auf ein Objekt vom Typ `Cuboid` verweist.
- ② Die Umwandlung des zweiten Objekts `secondForm` bewirkt aufgrund der expliziten Typkonversion ebenfalls keinen Compilerfehler. Es entsteht aber ein Laufzeitfehler, da die Referenzvariable nicht auf ein Objekt des Datentyps `Cuboid` verweist. Die beiden Datentypen sind nicht kompatibel.

```
Exception in thread "main" java.lang.ClassCastException: Rectangle
cannot be cast to Cuboid
    at TypeCast.main(TypeCast.java:8)
```

Die Ausgabe des Programms bei der Arbeit mit Java 9

Um Laufzeitfehler dieser Art zu vermeiden, sollten Sie eine Kompatibilitätsprüfung vornehmen.

Typkompatibilität eines Objekts prüfen

Mit dem Operator `instanceof` können Sie prüfen, ob ein Objekt vom Typ einer bestimmten Klasse (oder einer davon abgeleiteten Klasse) ist.

Beispiel: *TypeVerification.java*

```

class TypeVerification
{
    public static void main(String[] args)
    {
        Rectangle firstForm = new Cuboid();
        Rectangle secondForm = new Rectangle();
        ① if (firstForm instanceof Cuboid)
        {
            ② System.out.println("firstForm ist kompatibel zu Cuboid");
            Cuboid oneCuboid = (Cuboid)firstForm;
        }
        else
            System.out.println
                ("firstForm ist zu Cuboid nicht kompatibel");
        ③ if (secondForm instanceof Cuboid)
        {
            System.out.println("secondForm ist kompatibel zu Cuboid");
            Cuboid anotherCuboid = (Cuboid)secondForm;
        }
        else
            ④ System.out.println
                ("secondForm ist zu Cuboid nicht kompatibel");
    }
}

```

Überwachung der Typkonversion

- ① Die Referenzvariable `firstForm` verweist auf ein Objekt vom Typ `Cuboid`. Daher liefert die Bedingung mit `instanceof` den Wert `true`.
- ② Die Typumwandlung wird vorgenommen.
- ③ Die Referenzvariable `secondForm` verweist nicht auf ein Objekt vom Typ `Cuboid`. Die Bedingung mit `instanceof` liefert den Wert `false`.
- ④ Die Typumwandlung wird nicht vorgenommen, sondern es erfolgt die Ausgabe der Meldung.

```

firstForm ist kompatibel zu Cuboid
secondForm ist zu Cuboid nicht kompatibel

```

Die Ausgabe des Programms

8.6 Polymorphie in der Vererbung

Polymorphie bezieht sich neben der Vielgestaltigkeit von überladenen Methoden innerhalb einer Klasse auch auf die Vielgestaltigkeit von überschriebenen (überlagerten) Methoden in abgeleiteten Klassen. Der Compiler findet selbstständig die gewünschte Methode. Wird eine Methode eines Objekts aufgerufen, gibt es mehrere Möglichkeiten, wo sich die Methodendefinition befinden kann.

- ✓ Die Methode befindet sich in der Klasse, von der das Objekt erzeugt wurde, beispielsweise die Methode `getVolume` in der Klasse `Cuboid`.
- ✓ Die Methode befindet sich in der Basisklasse und wurde von der Klasse des Objekts geerbt. Beispielsweise erbt die Klasse `Cuboid` die Methode `getWidth` von der Basisklasse `Rectangle`.
- ✓ Die Methode befindet sich in der Basisklasse, wurde aber von der Klasse des Objekts überschrieben, beispielsweise die Methode `getArea`.

Beispiel für überschriebene Methoden in abgeleiteten Klassen: *Polymorphism.java*

Die Klasse `Rectangle` enthält eine Methode `getArea`, mit der der Flächeninhalt des Rechtecks berechnet wird. Auch die abgeleiteten Klassen `Cuboid` (Quader) und `Pyramid` (Pyramide) enthalten eine Methode `getArea`. Diese wurde aber überlagert, sodass sie die Oberfläche des Quaders berechnet bzw. für die Pyramide den Wert `-1` zurückgibt.

Das folgende Programm erzeugt entsprechend der zuvor beschriebenen Zuweisungskompatibilität mehrere Objekte der übergeordneten Superklasse `Rectangle`.

- ✓ Die erste Form ist ein Rechteck selbst.
- ✓ Die zweite Form ist ein Quader.
- ✓ Die dritte Form ist eine Pyramide.

Alle Formen basieren auf einem Rechteck und besitzen beispielsweise die Methoden `getWidth`, `getLength` und `getArea`, die in dem Programm aufgerufen werden.

```
class Polymorphism
{
    public static void main(String[] args)
    {
        ① Rectangle oneForm = new Rectangle(3, 5);
        ② Rectangle secondForm = new Cuboid(3, 5, 4);
        ③ Rectangle thirdForm = new Pyramid(3, 5, 4);

        ④ System.out.printf("Objektoberfläche %n(%d x %d): %d%n%n",
                           oneForm.getWidth(), oneForm.getLength(),
                           oneForm.getArea());
    }
}
```

```

⑤      System.out.printf("Objektoberfläche %n(%d x %d): %d%n%n",
                           secondForm.getWidth(),
                           secondForm.getLength(),
                           secondForm.getArea());
⑥      System.out.printf("Objektoberfläche %n(%d x %d): %d%n%n",
                           thirdForm.getWidth(),
                           thirdForm.getLength(),
                           thirdForm.getArea());

    }
}

```

Die Vielgestaltigkeit der Methode `getArea`

- ① Ein Rechteck wird erzeugt.
- ② Ein auf einem Rechteck aufbauender Quader wird erzeugt.
- ③ Eine auf einem Rechteck aufbauende Pyramide wird erzeugt.
- ④ Die Methoden `getWidth`, `getLength` und `getArea` der Klasse `Rectangle` werden ausgeführt.
- ⑤ Die Methoden `getWidth` und `getLength` der Klasse `Rectangle` und die überschriebene Methode `getArea` der Klasse `Cuboid` werden ausgeführt.
- ⑥ Die Methoden `getWidth` und `getLength` der Klasse `Rectangle` und die überschriebene Methode `getArea` der Klasse `Pyramid` werden ausgeführt.

```

Objektoberfläche
(3 x 5): 15      ④

Objektoberfläche
(3 x 5): 94      ⑤

Objektoberfläche
(3 x 5): -1      ⑥

```

Die Ausgabe des Programms

Für alle Objekte können die Methoden angewendet werden. Der Compiler entscheidet, welche Methode ausgeführt wird.

Die Objekte `secondForm` und `thirdForm` zeigen sich zwar in der Gestalt eines Quaders bzw. einer Pyramide. Methoden wie beispielsweise `getVolume` und `getBase` würden beim Kompilieren aber zu einer Fehlermeldung führen, denn sie sind in der Klasse `Rectangle` nicht definiert.

8.7 Die Superklasse `Object`

Jede Klasse, die Sie neu erzeugen, aber nicht von einer vorhandenen Klasse ableiten, besitzt „unsichtbar“ die Basisklasse `Object`. Diese Angabe müssen Sie bei der Klassendefinition nicht angeben.

```
class identifier [extends Object] ...
```

In Java ist die Klasse `Object` die direkte bzw. indirekte Basisklasse **aller Klassen**. Die in der Klasse `Object` definierten Methoden sind aufgrund der Vererbung in allen Klassen verfügbar.

Die Methode `getClass`

Eine Methode der Klasse `Object` ist die Methode `getClass`.

Mit der Methode `getClass` bestimmen Sie den Referenztyp eines Objekts.

```
class Cuboid
class Rectangle
```

Die Ausgabe des Programms

```
class WhichClass
{
    public static void main(String[] args)
    {
        Rectangle firstForm = new Cuboid();
        Rectangle secondForm = new Rectangle();
        System.out.println(firstForm.getClass());
        System.out.println(secondForm.getClass());
    }
}
```

Den Referenztyp von Objekten ermitteln: „WhichClass.java“

Weitere Methoden der Klasse `Object`

Die folgenden Methoden sind Methoden der Klasse `Object`. Sie besitzen jedoch nur eine Grundfunktionalität. In eigenen Klassen werden diese Methoden gewöhnlich überschrieben, um die gewünschte Funktionalität zu erhalten.

<code>public boolean equals(Object obj)</code>	Standardmäßig überprüft diese Methode, ob das Objekt selbst und das Objekt <code>obj</code> identisch sind. Die Methode liefert dann den Wert <code>true</code> , wenn die zu vergleichenden Referenzvariablen auf dasselbe Objekt verweisen. Sie müssen die Methode in abgeleiteten Klassen überschreiben, wenn Sie auf inhaltliche Gleichheit (Attribute) eines Objekts testen möchten.
<code>protected Object clone()</code>	Hiermit erzeugen Sie ein neues Objekt als exakte Kopie des aktuellen Objekts.
<code>public String toString()</code>	Die Methode <code>toString</code> liefert eine String-Repräsentation des Objekts.
<code>int hashCode()</code>	Gibt einen eindeutigen Wert vom Typ <code>Integer</code> zur Identifizierung des Objekts zurück

Die Methoden `clone` und `equals` anwenden

Die folgende Klasse `ObjectMethods` enthält Methoden zum Kopieren und Vergleichen von Objekten. Sie überschreibt die von der Klasse `Object` geerbten Methoden `clone` und `equals`.

Beispiel zum Überschreiben der Methoden `clone` und `equals`:
ObjectMethods.java

```
class ObjectMethods
{
    ① private int data = 0;
```

```

②    int getData()
    {
        return data;
    }
③    void setData(int data)
    {
        this.data = data;
    }
④    protected Object clone()
    {
⑤        ObjectMethods om = new ObjectMethods();
⑥        om.setData(data);
⑦        return om;
    }
⑧    public boolean equals(Object obj)
    {
⑨        if (this == obj)
            return true;
⑩        if ((obj == null) || (this.getClass() != obj.getClass()))
            return false;
⑪        return (data == ((ObjectMethods)obj).getData());
    }
}

```

- ① Das Attribut `data` der Klasse `ObjectMethods` repräsentiert den aktuellen Zustand des Objekts (es gibt keine weiteren Argumente). Es wird mit dem Wert 0 initialisiert.
- ②-③ Für den Zugriff auf das gekapselte Argument `data` werden die Getter-Methode `getData` und die Setter-Methode `setData` bereitgestellt.
- ④ **Die Methode `clone`** erzeugt eine Kopie des aktuellen Objekts.
- ⑤ Zuerst wird ein neues Objekt vom Typ `ObjectMethods` erzeugt.
- ⑥ Über die Methode `setData` wird der Wert des Attributs des aktuellen Objekts auf das Attribut des neuen Objekts übertragen (kopiert).
- ⑦ Das neue Objekt wird als Rückgabewert der Methode zurückgegeben.
- ⑧ **Die Methode `equals`** soll die inhaltliche Gleichheit von zwei Objekten des Typs `ObjectMethods` überprüfen.
- ⑨ Als Erstes wird geprüft, ob es sich bei dem übergebenen Objekt um das aktuelle Objekt selbst handelt. In diesem Fall wird die Gleichheit sofort festgestellt und `true` zurückgegeben.
- ⑩ Anschließend wird geprüft, ob das zu vergleichende Objekt `null` ist (nicht existiert) oder ob die Objekte von unterschiedlichem Typ sind. In beiden Fällen kann die inhaltliche Gleichheit nicht geprüft werden und der Wert `false` wird zurückgegeben.
- ⑪ Hier findet der eigentliche inhaltliche Vergleich statt. Da der Parameter `obj` vom Typ `Object` ist, wird eine Typumwandlung nach `ObjectMethods` benötigt. Für den Typ `ObjectMethods` steht die Methode `getData` zur Verfügung. Der Wert von `data` des Objekts `obj` kann mit dem Wert von `data` des aktuellen Objekts verglichen werden.

Beispiel zum Testen der Klasse `ObjectMethods`: `TestObjectMethods.java`

In der `main`-Methode der Klasse `TestObjectMethods` werden die neuen Methoden getestet.

```
class TestObjectMethods
{
    public static void main(String[] args)
    {
        ① ObjectMethods obj1 = new ObjectMethods();
        ObjectMethods obj2 = new ObjectMethods();
        ② obj1.setData(123);
        ③ if (obj1.equals(obj2))
            System.out.println("obj1 und obj2 sind gleich!");
        else
            System.out.println("obj1 und obj2 sind nicht gleich!");
        ④ ObjectMethods obj3 = (ObjectMethods)obj1.clone();
        ⑤ if (obj1.equals(obj3))
            System.out.println("obj1 und obj3 sind gleich!");
        else
            System.out.println("obj1 und obj3 sind nicht gleich!");
    }
}
```

- ① Es werden zwei neue Objekte `obj1` und `obj2` der Klasse `ObjectMethods` erzeugt.

obj1 und obj2 sind nicht gleich! ⑥
obj1 und obj3 sind gleich! ⑦

Die Ausgabe des Programms

- ② Über die Methode `setData` wird das Attribut `data` des Objekts `obj1` auf den Wert 123 geändert. Dadurch sind die Objekte `obj1` und `obj2` nicht mehr inhaltlich gleich.
- ③ Die Methode `equals` prüft, ob die Objekte `obj1` und `obj2` inhaltlich gleich sind. Da die Objekte nicht gleich sind, wird die entsprechende Ausgabe ⑥ durchgeführt.
- ④ Die Referenzvariable `obj3` wird definiert. `obj3` wird ein geklontes `obj1`-Objekt zugewiesen, d. h., das Attribut `data` von `obj3` hat ebenfalls den Wert 123.
- ⑤ Da die Objekte `obj1` und `obj3` inhaltlich gleich sind, erfolgt die entsprechende Ausgabe ⑦.

8.8 Finale Klassen

Wenn Sie verhindern möchten, dass von einer Klasse andere Klassen abgeleitet werden können, erzeugen Sie finale Klassen. Finale Klassen haben die folgenden Eigenschaften:

- ✓ Finale Klassen werden durch den Modifizierer `final` gekennzeichnet.
- ✓ Sie können keine weiteren Klassen von einer als `final` deklarierten Klasse ableiten. Beispielsweise haben Sie eine Klasse definiert, die Methoden zur Verschlüsselung von Daten bereitstellt.

Mit `final` verhindern Sie, dass die Klasse abgeleitet werden kann und der Verschlüsselungsalgorithmus möglicherweise durch Überschreiben von Methoden verändert wird.

- ✓ Der Aufruf von Methoden finaler Klassen ist schneller, da keine dynamische Bindung der Methodenaufrufe erfolgt (während der Laufzeit). Dies wird in vielen Hilfsklassen genutzt, die Methoden bereitstellen, ohne dass Objekte der Klassen benötigt werden. Die Methoden sind dann in der finalen Klasse als `static` definiert, damit sie ohne ein Objekt der Klasse aufgerufen werden können.

Beispiel für eine statische Methode in einer finalen Klasse: *Formula.java*

```
public final class Formula
{
    public static double getSqaare(double value)
    {
        return (value * value);
    }
}
```

Eine finale Klasse des JDK ist beispielsweise die Klasse `java.lang.Math`.

8.9 Abstrakte Klassen und abstrakte Methoden

Was sind abstrakte Klassen und abstrakte Methoden?

Abstrakte Klassen sind Klassen, die als Vorlage für andere Klassen dienen. Eine Klasse muss als abstrakt gekennzeichnet werden, wenn sie eine **abstrakte Methode** enthält.

- ✓ Abstrakte Methoden enthalten nur die Methodenköpfe: Es ist festgelegt, welche Parameter die Methode erfordert, ob die Methode einen Wert zurückgibt und welchen Datentyp der Rückgabewert gegebenenfalls hat.
- ✓ Abstrakte Methoden enthalten keinen Methodenrumpf: Es ist nicht beschrieben, welche Funktionalität die Methode bewirkt.

Da die Beschreibung der abstrakten Klasse nicht vollständig sein muss, werden von abstrakten Klassen keine Objekte gebildet. Abstrakte Klassen können Sie nur als Basisklasse für andere Klassen verwenden. Abstrakte Klassen werden daher auch als **abstrakte Basisklassen** bezeichnet. In einer abstrakten Klasse werden die Schnittstellen für die abgeleiteten Klassen in Form von abstrakten Methoden vorgegeben. Die Aufgabe der abgeleiteten Klassen ist, die abstrakten Methoden zu implementieren. Von der abstrakten Klasse wird so eine **konkrete Klasse** abgeleitet.

- ✓ Eine Klasse mit mindestens einer abstrakten Methode muss als abstrakt gekennzeichnet werden. Eine abstrakte Klasse muss jedoch nicht zwingend eine abstrakte Methode besitzen.
- ✓ Sie können von abstrakten Klassen keine Instanzen erzeugen.
- ✓ Es ist möglich, Referenzvariablen vom Typ einer abstrakten Klasse zu erzeugen. Diesen Variablen können Objekte der davon abgeleiteten Klassen zugewiesen werden.

In seltenen Fällen, in denen die Klasse nur die wesentliche Funktionalität bereitstellt und die Klasse ohne die Erweiterung durch eine abgeleitete Klasse nicht sinnvoll genutzt werden kann, wird diese Klasse ebenfalls als abstrakte Klasse definiert, obwohl sie keine abstrakten Methoden enthält.

Abstrakte Klassen anwenden

Abstrakte Klassen werden z. B. in den folgenden Situationen eingesetzt.

- ✓ Sie dienen als Vorlage für andere Klassen. Damit ist gewährleistet, dass die abgeleiteten Klassen die vorgegebenen abstrakten Methoden implementieren, im einfachsten Fall aber nur einen leeren Methodenrumpf besitzen.
- ✓ Sie definieren über normale Methoden eine Grundfunktionalität und durch abstrakte Methoden eine einheitliche Schnittstelle, die abgeleitete Klassen implementieren müssen.

Syntax für die Definition abstrakter Klassen und abstrakter Methoden

- ✓ Durch den Modifizierer `abstract` kennzeichnen Sie eine Klasse als abstrakte Klasse ①.
- ✓ Durch weitere Modifizierer (modifier) können Sie die Zugriffsrechte für die Klasse festlegen.
- ✓ Der Modifizierer `final` kann für eine abstrakte Klasse nicht verwendet werden, da die als `final` definierte Klasse nicht abgeleitet werden kann.
- ✓ Der Zugriffsmodifizierer `private` kann für eine abstrakte Methode nicht verwendet werden, da die als `private` definierte Methode in der abgeleiteten Klasse nicht sichtbar wäre.
- ✓ Abstrakte Methoden werden ebenfalls mit dem Modifizierer `abstract` gekennzeichnet und besitzen keinen Methodenrumpf ②. Sie müssen in abgeleiteten Klassen implementiert werden.

```
[modifier] abstract class identifier ①
{
    ...
    [modifier] abstract type identifier1(); ②
}
```

Beispiel für die Verwendung abstrakter Klassen: *Shape.java*, *Triangle.java*, *Circle.java*

Eine Klasse `Shape` soll als abstrakte Klasse als Basis für geometrische Formen dienen. Alle Klassen, die von der Klasse `Shape` abgeleitet werden, sollen eine Methode enthalten, mit der der Flächeninhalt für die jeweilige Form berechnet werden kann. Um sicherzustellen, dass die abgeleiteten Klassen eine entsprechende Methode `getArea` implementieren, enthält die Klasse `Shape` eine abstrakte Methode `getArea`.

Definition einer abstrakten Klasse `Shape`

```
① abstract class Shape
{
  ② private boolean filled = false;
```

```

③ boolean isFilled()
{
    return filled;
}
...
④ abstract double getArea();
}

```

- ① Die abstrakte Klasse `Shape` wird definiert. Da die Klasse eine abstrakte Methode enthält, muss auch die Klasse mit dem Schlüsselwort `abstract` als abstrakt gekennzeichnet werden.
- ②-③ Abstrakte Klassen können Variablen wie auch konkrete Methoden besitzen. Diese werden wie üblich an abgeleitete Klassen vererbt.
- ④ Die abstrakte Methode `getArea` muss in allen abgeleiteten Klassen implementiert werden, damit von diesen Objekte erzeugt werden können. Die Methode `getArea` soll den jeweiligen Flächeninhalt als Fließkommazahl (`double`) berechnen und zurückgeben.

Klassen von der abstrakten Klasse `Shape` ableiten

```

① class Triangle extends Shape
{
②     private double base;
    private double altitude;
③     double getBase()
    {
        return base;
    }
    double getAltitude()
    {
        return altitude;
    }
    ...
④     double getArea()
    {
        return getBase()
            * getAltitude() / 2;
    }
}

```

```


① class Circle extends Shape
{
②     private double radius;
③     double getRadius()
    {
        return radius;
    }
    ...
④     double getArea()
    {
        return 3.14159 * getRadius()
            * getRadius();
    }
}

```

- ① Die Klassen `Triangle` und `Circle` erweitern die abstrakte Klasse `Shape`.
- ②-③ Die Klassen enthalten entsprechende Attribute und Methoden wie beispielsweise Getter-Methoden und Setter-Methoden.
- ④ Durch die Implementation von `getArea` erzeugen Sie eine konkrete Klasse. Von den beiden Klassen `Triangle` und `Circle` können Objekte erzeugt werden.

8.10 Übung

Abstrakte Klassen und Vererbung anwenden

Level		Zeit	ca. 60 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Abstrakte Klassen und Methoden erstellen ✓ Vererbung anwenden ✓ Abstrakte Methoden implementieren ✓ Statische Methoden deklarieren und verwenden 		
Übungsdatei	--		
Ergebnisdateien	<i>Zeit.java, ZeitFormat.java, ZeitFormat24.java, ZeitFormat12.java, Uhrzeit.java, Uebung1.java, Uebung4.java, Uebung5.java, Uebung6.java</i>		

1. Definieren Sie eine Klasse `Zeit`, die geeignete Getter- und Setter-Methoden zum Ändern und Ermitteln einer gespeicherten Uhrzeit besitzt (Stunde, Minute).
2. Leiten Sie eine abstrakte Klasse `ZeitFormat` mit Konstruktor ab, die eine abstrakte Methode zur Ausgabe der Zeitangabe auf der Konsole besitzt.
3. Leiten Sie von der abstrakten Klasse `ZeitFormat` die Klassen `ZeitFormat24` und `ZeitFormat12` ab und implementieren Sie dort die abstrakten Methoden der Klasse `ZeitFormat`.
4. Definieren Sie sechs Referenzvariablen vom Typ `ZeitFormat` und erzeugen Sie für diese Variablen drei Objekte vom Typ `ZeitFormat12` und drei Objekte vom Typ `ZeitFormat24`. Initialisieren Sie die Objekte mit verschiedenen Uhrzeit-Angaben und rufen Sie die jeweilige Ausgabemethode auf.
5. Definieren Sie für die Klasse `Zeit` aus Übung ① die Methoden `clone` und `equals`. Verwenden Sie die Methoden in einem Testprogramm.
6. Entwerfen Sie eine finale Klasse `UhrZeit`, die eine statische Methode entspricht bereitstellt, um zwei Zeitangaben (Typ `Zeit`) zu vergleichen.

9

Packages und Module

9.1 Klassen in Packages organisieren

Was sind Packages?

Ein wichtiges Merkmal von Java ist die Wiederverwertbarkeit von Klassen. Sie möchten Klassen anderer Programmierer nutzen und umgekehrt. Die Vielzahl der einzelnen Klassen erfordert eine strukturierte Organisation dieser Klassen. Außerdem müssen die Klassen eindeutig unterschieden werden können. Dies wird dadurch erreicht, dass die Klassendateien in Paketen, sogenannten **Packages**, organisiert werden. So kann eine Klasse `Rectangle` zwar nur einmal in einem Package vorkommen, es kann aber in verschiedenen Packages eine Klasse mit diesem Namen existieren.

- ✓ Packages können geschachtelt werden, indem mehrere Packages wiederum in einem Package zusammengefasst werden.
- ✓ Die Package-Hierarchie wird auf Rechnern meist durch eine Ordnerstruktur abgebildet (es ist auch möglich, mehrere Klassen in Datenbanken oder Archiven zusammenzufassen).
- ✓ Jede Klasse gehört zu genau einem Package und befindet sich daher in einem bestimmten Ordner.

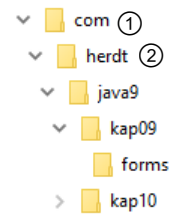
In den vorherigen Beispielen konnten Sie in der Klasse `Geometry` die Klassen `Rectangle`, `Cuboid` und `Pyramid` verwenden, ohne explizit angeben zu müssen, wo sich die Dateien mit den Klassen befinden.

Standardmäßig verwendet Java das aktuelle Verzeichnis, um nach Dateien mit den verwendeten Klassen zu suchen. Die Klassen `Rectangle`, `Cuboid` und `Pyramid` gehören bei den vorherigen Beispielen zum selben Package wie die Klasse `Geometry`.

Ein eigenes Package einrichten

Erzeugen Sie für Ihre selbst erstellten Klassen eine eigene Package-Struktur. Diese Package-Struktur bilden Sie als Ordnerstruktur ab. Als Ausgangsordner für die Struktur dient der aktuelle Ordner. Wenn Sie die Umgebungsvariable `CLASSPATH` festgelegt haben, ist der dort angegebene Ordner der Ausgangsordner.

- ✓ Es ist üblich, die Package-Struktur mit dem Domain-Namen (beispielsweise herdt.com) in umgekehrter Reihenfolge zu beginnen ①/②.
- ✓ Innerhalb dieses Packages können Sie beliebige weitere Packages, sogenannte **Subpackages** einrichten, mit denen Sie die Klassen beispielsweise thematisch ordnen können. Das bedeutet, dass Sie innerhalb eines Ordners entsprechend der Package-Struktur weitere Ordner anlegen.



Package-Struktur
als Ordnerstruktur

Für die Beispiele zu diesem Buch wurde eine Package-Struktur festgelegt, die in der Abbildung als Ordnerstruktur dargestellt ist.

Die Namen der Packages und somit der Ordner werden kleingeschrieben.



Die Namen müssen innerhalb des Packages eindeutig sein. So darf im Beispiel das Package `java9` keine Klasse mit dem Namen `kap09` enthalten, denn es existiert bereits ein Package mit dem Namen `kap09`.

Eine Klasse mit dem vollen Package-Namen identifizieren

Da Klassen mit einem bestimmten Namen in verschiedenen Packages vorhanden sein können, wird der volle Package-Name zur eindeutigen Identifikation einer Klasse in der Package-Hierarchie hinzugezogen. Der Zugriff auf eine Klasse erfolgt immer über den Package-Namen. Dazu wird folgende Schreibweise verwendet:

- ✓ An den Namen eines Packages wird, jeweils mit einem Punkt getrennt, der Name des entsprechenden Subpackages angehängt.
- ✓ Hinter dem vollständigen Package-Namen folgt mit einem Punkt abgetrennt der Name der gewünschten Klasse.

```
packageName [.subpackageName ...] .className
```

Der Bytecode der Klasse `Rectangle` ist in der Datei `Rectangle.class` im Verzeichnis `..\com\herdt\java9\kap09\forms` gespeichert. Der aus dem **vollständigen Package-Namen** und dem Namen der Klasse zusammengesetzte **vollständige Name der Klasse** lautet daher:

```
com.herd . java9 . kap09 . forms . Rectangle
```

Dieser zusammengesetzte vollständige Name der Klasse wird auch als **qualifizierter Klassenname** bezeichnet.

Die CLASSPATH-Variable verwenden

Damit der Compiler bzw. der Interpreter eine Datei findet, verknüpft er den vollständigen Package-Namen nacheinander mit den jeweiligen Pfadangaben, die Sie als Classpath festgelegt haben (`-classpath ...` bzw. `CLASSPATH = ...`). Wird eine Klasse nicht gefunden, so erhalten Sie einen entsprechenden Fehlerhinweis.

Beachten Sie, dass Klassennamen üblicherweise mit einem Großbuchstaben beginnen.

Beispiel

Das aktuelle Verzeichnis ist beispielsweise `c:\Daten\Beispiele`, und Sie haben die CLASSPATH-Variable mit dem CLASSPATH-Parameter folgendermaßen angegeben:

```
-CLASSPATH .;c:\uebung\java9
```

Die CLASSPATH-Variable enthält in diesem Beispiel zwei Verzeichnisse, die durch ein Semikolon getrennt sind. Der Punkt steht für das aktuelle Verzeichnis.

Um die Klasse `com.herd.java9.kap09.forms.Rectangle` zu finden, sucht der Compiler bzw. der Interpreter nach einer Datei `Rectangle.java` (bzw. `.class`) in folgenden Ordnern:

- ✓ `c:\Daten\Beispiele\com\herdt\java9\kap09\forms`
- ✓ `c:\uebung\java9\com\herdt\java9\kap09\forms`

Ohne die CLASSPATH-Variable arbeiten

Wenn Sie die CLASSPATH-Variable **nicht** festgelegt haben, müssen Sie den Compiler bzw. den Interpreter in dem Verzeichnis aufrufen, in dem er die Package-Struktur findet.

Beispiel

Der Bytecode der Klasse `com.herd.java9.kap09.forms.Rectangle` ist in der Datei `Rectangle.class` im Ordner `c:\uebung\java9\com\herdt\java8\kap09\forms` gespeichert. Um die Klasse verwenden zu können, müssen Sie den Interpreter im Ordner `c:\uebung\java9\` ausführen.

Klassen einem Package zuordnen

Um eine `*.java`-Datei zu einem Package hinzuzufügen, fügen Sie zu Beginn der Quelltextdatei eine sogenannte `package`-Vereinbarung ein. Es handelt sich **nicht** um eine Anweisung, denn Anweisungen stehen innerhalb von Klassen. Die `package`-Vereinbarung **muss** am Anfang einer Quelltextdatei stehen (eine Ausnahme bilden Kommentare).

Syntax der `package`-Vereinbarung

- ✓ Die `package`-Vereinbarung beginnt mit dem Schlüsselwort `package`.
- ✓ Geben Sie anschließend den vollständigen Package-Namen ein, der sich aus mehreren durch Punkte getrennten Subpackages zusammensetzen kann.
- ✓ Schließen Sie die `package`-Vereinbarung mit einem Semikolon ab.

```
package  
packageName [ . subpackageName . . . ] ;
```

Speichern Sie die Quelltextdatei in dem entsprechenden Unterordner. Jeder der Bestandteile des vollständigen Package-Namens kennzeichnet dabei beispielsweise den Namen eines Ordners. Zum Beispiel speichern Sie eine Klasse, die Sie dem Package `com.ihrefirma.tools` zugeordnet haben, in einer Quelltextdatei im Ordner `..\com\ihrefirma\tools`.

Wenn Sie keine `package`-Vereinbarung einfügen, gehört die Klasse zum Standard-Package. Dies entspricht dem Ordner, den Sie mit der Classpath-Variablen festgelegt haben.

9.2 Zugriffsrechte in Packages

Das Default-Zugriffsrecht

In den bisherigen Beispielen wurde kein Zugriffsmodifizierer bei der Definition einer Klasse angegeben. Dies bedeutet, dass das **Default-Zugriffsrecht**, auch Package-Zugriffsrecht genannt, verwendet wird.

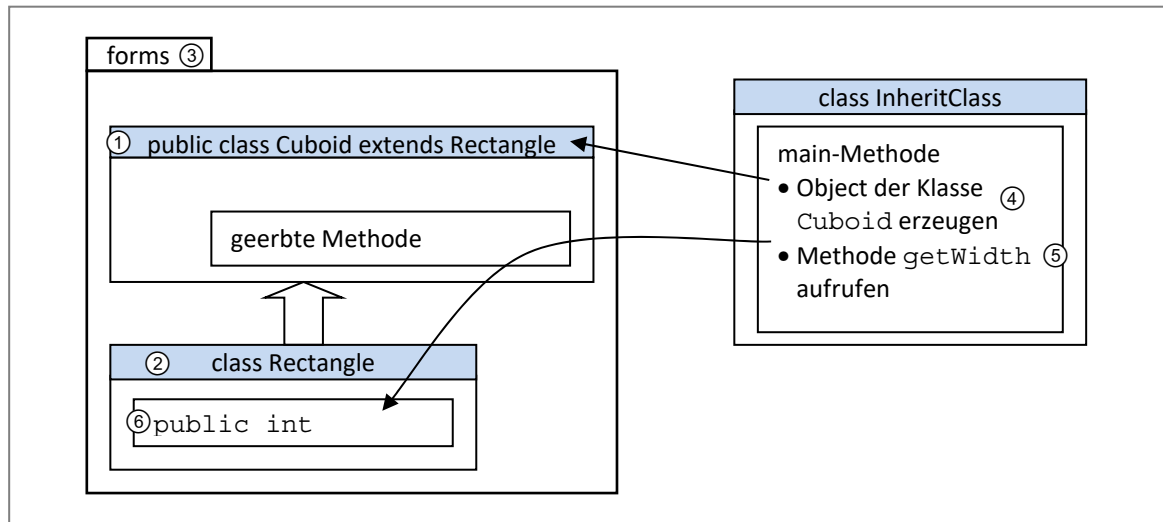
Auf eine Klasse mit Default-Zugriffsrecht können alle anderen Klassen desselben Packages zugreifen. Sofern auch die Methoden der Klasse mit Default-Zugriffsrecht versehen sind, können diese Methoden von den anderen Klassen des Packages verwendet werden. Gleiches gilt für die Attribute, sofern sie nicht wie gewöhnlich üblich als `private` gekapselt sind.

Zugriff aus anderen Packages ermöglichen

Um auch Klassen anderer Packages den Zugriff auf eine Klasse zu ermöglichen, definieren Sie die Klasse als öffentlich (`public`).

- ✓ Zur Definition einer öffentlichen Klasse verwenden Sie den Zugriffsmodifizierer `public`. Für den Zugriff bestehen dann keine Einschränkungen.
- ✓ Alle Methoden, die von Klassen anderer Packages verwendet werden sollen, müssen Sie ebenfalls mit dem Zugriffsmodifizierer `public` definieren.

Beispiel: *com.herdt.java9.kap09*



- ✓ Die Klasse Cuboid ① erbt von der Klasse Rectangle ② die Methode getWidth.
- ✓ Die Klassen Cuboid und Rectangle gehören zum Package `com.herdt.java9.kap09.forms` ③.
- ✓ In der Klasse InheritClass in einem anderen Package erzeugt die main-Methode ein Objekt der Klasse Cuboid ④ und führt für dieses Objekt die geerbte Methode getWidth aus ⑤.
- ✓ Da dieser Methodenaufruf aus einem anderen Package heraus erfolgt, ist das Zugriffsrecht `public` erforderlich.
- ✓ Die Methode getWidth muss daher in der Basisklasse Rectangle als `public` definiert sein ⑥.

Beim Vererben einer Methode verändern sich die Zugriffsrechte nicht.

Legen Sie die Zugriffsrechte sorgsam fest. Geben Sie Zugriffsrechte nur in dem Umfang, in dem sie benötigt werden.

9.3 Packages einbinden

Direkt auf Klassen in anderen Packages zugreifen

Um Klassen aus anderen Packages nutzen zu können, müssen Sie dem Compiler nicht nur den Klassennamen, sondern auch den vollständigen Package-Namen angeben. Dazu können Sie wie im folgenden Beispiel den qualifizierten Klassennamen verwenden (① und ②):

Beispiel: `com\herdt\java9\kap09\UsePackage.java`

```
package com.herdt.java9.kap09;
class UsePackage
{
    public static void main(String[] args)
    {
        ① com.herdt.java9.kap09.forms.Cuboid oneForm =
        ②      new com.herdt.java9.kap09.forms.Cuboid(3, 5, 4);

        System.out.printf(
            "Der Quader (%d x %d x %d)%nhat die Oberfläche %d%n%n",
            oneForm.getWidth(), oneForm.getLength(),
            oneForm.getHeight(), oneForm.getArea());
    }
}
```

Mit qualifizierten Klassennamen arbeiten

Klassen importieren

Über die `import`-Vereinbarung können Sie eine Klasse importieren. Sie können die importierte Klasse anschließend verwenden, ohne dass Sie dabei jeweils den qualifizierten Klassennamen angeben müssen. Bei Bedarf ist es auch möglich, mit einer einzigen `import`-Vereinbarung **alle Klassen** eines Packages einzubinden:

Syntax der `import`-Vereinbarung

```
import completePackageName.className; ①
import completePackageName.*; ②
```

- ✓ Beginnen Sie die `import`-Vereinbarung mit dem Schlüsselwort `import`.
- ✓ Geben Sie den qualifizierten Klassennamen an, bestehend aus dem vollständigen Package-Namen und dem Klassennamen ①.
- ✓ Oder geben Sie den vollständigen Package-Namen ein und mit einem Punkt abgetrennt einen Stern (*) als Joker-Zeichen, um alle Klassen innerhalb dieses Packages zu importieren.
- ✓ Um die Klassen eines Subpackages ebenfalls zu importieren, müssen Sie auch für dieses Package eine entsprechende `import`-Vereinbarung formulieren.

Beispiel für das Importieren von Klassen: `com\herdt\java9\kap09\ImportClass.java`

- ① Die import-Vereinbarung müssen Sie nach einer optionalen package-Vereinbarung und vor der ersten Klassendefinition angeben.
- ② Die importierte Klasse können Sie anschließend wie gewohnt verwenden.

```

package com.herdt.java9.kap09;
① import com.herdt.java9.kap09.forms.Cuboid;
import com.herdt.java9.kap09.forms.Pyramid;

class ImportClass
{
    public static void main(String[] args)
    {
        Cuboid oneForm = new Cuboid(3, 5, 4);
        Pyramid anotherForm = new Pyramid(3, 3, 6);
        ...
    }
}

```

Einzelne Klassen eines Packages importieren

Beispiel für das Importieren aller Klassen eines Packages: `com\herdt\java9\kap09\ImportPackage.java`

- ① Wenn Sie mehrere Klassen eines Packages benötigen, können Sie das Importieren dieser Klassen in einer einzigen Vereinbarung zusammenfassen.

```

package com.herdt.java9.kap09;
① import com.herdt.java9.kap09.forms.*;

class ImportPackage
{
    public static void main(String[] args)
    {
        Cuboid oneForm = new Cuboid(3, 5, 4);
        Pyramid anotherForm = new Pyramid(3, 3, 6);
        ...
    }
}

```

Alle Klassen eines Packages importieren

Besonderheiten beim Arbeiten mit der **import**-Vereinbarung

- ✓ Über **eine** import-Vereinbarung können Sie immer nur **ein** Package importieren. Besitzt das Package weitere Subpackages, die Sie ebenfalls einbinden möchten, müssen Sie diese separat importieren.
- ✓ Sie haben jederzeit auch ohne import-Vereinbarung Zugriff auf alle Klassen in den vorhandenen Packages. Der Nachteil ist der höhere Schreibaufwand, da Sie den qualifizierten Klassennamen bei Methodenaufrufen usw. verwenden müssen, z. B. `com.herdt.java9.kap09.forms.Rectangle.setWidth(25);`

```

import com.herdt.java9.kap09.*;
import com.herdt.java9.kap09.forms.*;

```

- ✓ Benötigen Sie zwei Klassen oder Interfaces mit dem gleichen Namen aus verschiedenen Packages, müssen Sie, auch wenn Sie eine Import-Vereinbarung getroffen haben, den qualifizierten Klassennamen verwenden wie beispielsweise:

`com.herdtd.java9.kap09.forms.Rectangle` und

`com.herdtd.java9.kap09.Rectangle`

Nur so ist die jeweilige Klasse eindeutig identifiziert.

9.4 Statisches Importieren

Statische Attribute und Methoden, also Elemente, die Sie verwenden können, ohne ein Objekt der Klasse zu erzeugen, lassen sich direkt mit einer `import`-Vereinbarung importieren. Sie ersparen sich dadurch Schreibaufwand und verbessern die Lesbarkeit des Quelltextes.

Syntax des statischen Imports

```
import static completePackageName.className.identifier; ①  
import static completePackageName.className.*; ②
```

- ✓ Sie beginnen die Vereinbarung mit dem Schlüsselwort `import`.
- ✓ Dann folgt das Schlüsselwort `static`.
- ✓ Anschließend geben Sie den qualifizierten Klassen-Namen ein und mit einem Punkt abtrennt den Namen der Konstanten bzw. den Namen der statischen Methode ①.
- ✓ Die Vereinbarung beenden Sie mit einem Semikolon.
- ✓ Statt mehrere Methoden oder Konstanten einer Klasse einzeln zu importieren, können Sie diese `import`-Vereinbarungen zu einer Vereinbarung zusammenfassen: Geben Sie hinter dem Klassennamen mit einem Punkt abgetrennt das Jokerzeichen `*` ein ②.

Beispiel: `com\herdt\java9\kap09\NormalImport.java` und
`com\herdt\java9\kap09\StaticImport.java`

Um die Methode `sqrt` (Quadratwurzel) oder die Konstante `PI` (Kreiskonstante π) der Klasse `Math` im Package `java.lang` zu verwenden, können Sie mit dem **statischen Import** direkt die Methode bzw. die Konstante importieren. Sie können diese Methoden und Konstanten anschließend in der Klasse verwenden, als wären sie Bestandteil der Klasse. Es ist nicht erforderlich, den Klassennamen vor dem Namen der Konstanten bzw. der Methode anzugeben.

Die beiden nachfolgenden Quelltexte stellen den Import einer Klasse und den statischen Import gegenüber. Das Beispiel zeigt die Vereinfachung der Schreibweise bei der Verwendung der Konstanten und statischen Methoden nach einem statischen Import.

Klasse importieren

```
package com.herdt.java9.kap09;
① import java.lang.Math;
class NormalImport
{
    public static void main(String[] args)
    {
        double d = 256.00;
        ② System.out.printf
            ("Die Kreiszahl PI lautet: %f%n", Math.PI);
        ③ System.out.printf
            ("Die Quadratwurzel von %f ist %f%n", d, Math.sqrt(d));
    }
}
```

Die Klasse `Math` importieren

- ① Über die `import`-Vereinbarung importieren Sie die Klasse `Math`.
- ②-③ Um ein Attribut oder eine Methode zu verwenden, müssen Sie den Klassen-Namen angeben.

Statischer Import

```
package com.herdt.java9.kap09;
① import static java.lang.Math.PI;
② import static java.lang.Math.sqrt;
class StaticImport
{
    public static void main(String[] args)
    {
        double d = 256.00;
        ③ System.out.printf("Die Kreiszahl PI lautet: %f%n", PI);
        ④ System.out.printf
            ("Die Quadratwurzel von %f ist %f%n", d, sqrt(d));
    }
}
```

Statischer Import der statischen Methode `sqrt` und der statischen Konstanten `PI` der Klasse `Math`

- ①-② Über die statische `import`-Vereinbarung importieren Sie die Konstante `PI` und die statische Methode `sqrt` der Klasse `Math`.
- ③-④ Die Konstante und die statische Methode können Sie wie Elemente der eigenen Klasse ohne eine Klassen-Angabe verwenden.

Die beiden statischen `import`-Vereinbarungen lassen sich alternativ zu einer Vereinbarung zusammenfassen ⑤, da sich beide auf dieselbe Klasse beziehen. Mit dieser Vereinbarung werden **alle** Konstanten und **alle** statischen Methoden der Klasse importiert.

```
package com.herdt.java9.kap09;
import static java.lang.Math.*; ⑤
class StaticImport
{
    ...
}
```

Besonderheit beim Importieren

Sofern Sie in der Klasse eine eigene Konstante oder statische Methode mit demselben Namen wie die importierten definiert haben, wird automatisch die Konstante bzw. Methode der Klasse verwendet.

Das nebenstehende Beispiel zeigt, dass die selbst definierte Konstante `PI` ② ausgegeben wird und nicht die in der Klasse `java.lang.Math` definierte Konstante `PI` für die Kreiszahl π ①.

```
package com.herdt.java9.kap09;
import static java.lang.Math.PI; ①
class TestImport
{
    public static void main(String[] args)
    {
        ② int PI = 256; //PI entspricht hier
                       //nicht der Kreiszahl!
        System.out.printf("PI: " + PI);
    }
}
```

```
PI: 256
```

Die Ausgabe des Programms

9.5 Mit dem JDK mitgelieferte Packages

Standard-Packages

Mit dem JDK wird eine Vielzahl von Klassen mitgeliefert, die als sogenannte Bibliotheken hierarchisch in vielen Packages strukturiert sind. Diese Packages befinden sich im Installationsverzeichnis des JDK und werden automatisch gefunden. Der Verzeichnispfad zu diesen mitgelieferten Packages muss nicht explizit als Classpath festgelegt werden.

Eine dieser Bibliotheken umfasst die Sprachelemente und bildet die Grundlage für alle weiteren Klassen. Hierzu gehört auch die Klasse `Object`. Diese Bibliothek ist unter dem Package-Namen `java.lang` gespeichert.

Das Package `java.lang` wird immer automatisch eingebunden, da es Klassen enthält, die für alle Java-Anwendungen benötigt werden.

Die folgende Tabelle gibt Ihnen einen Überblick über einige weitere Packages:

<code>java.lang</code>	Standard-Funktionalitäten – wird immer automatisch eingebunden
<code>java.io</code>	Klassen zur Bildschirm- und Dateieingabe und -ausgabe
<code>java.nio</code>	Package mit Klassen für das Dateihandling und den Zugriff auf das Dateisystem (seit Java 7)
<code>java.math</code>	Klassen mit mathematischen Methoden und Konstanten
<code>java.net</code>	Netzwerkzugriffe und -kommunikation
<code>java.util</code>	Datenstrukturen, Tools- und Hilfsklassen
<code>java.awt</code>	Erstellung von Programmen mit grafischer Benutzeroberfläche (teilweise unterschiedliche Darstellung auf verschiedenen Plattformen (z. B. Windows und Linux))
<code>javax.swing</code>	Erstellung von Programmen mit grafischer Benutzeroberfläche, aber gleicher Darstellung auf verschiedenen Plattformen (z. B. Windows und Linux)

Archive

Grundsätzlich müssten sich in Java alle Klassen in Packages, also in einer Verzeichnisstruktur, befinden. Wenn Sie aber beispielsweise nach den Unterverzeichnissen `.. \java\lang` oder `.. \java\math` innerhalb Ihrer JDK-Installation suchen, werden Sie diese nicht finden. Die Verwaltung der Klassen und Pakete innerhalb einer Verzeichnisstruktur hat mehrere Nachteile.

- ✓ Es existieren sehr viele kleine Dateien (Speicherplatzverschwendung).
- ✓ Sie benötigen längere Zugriffszeiten auf diese Dateien, z. B. für Verzeichniswechsel.
- ✓ Wenn Sie die Dateien weitergeben wollen, müssen Sie diese entweder alle kopieren oder packen und später wieder entpacken.

Java fasst aus diesen Gründen zusammengehörige Klassen in Archiven zusammen, die die Struktur einer Package-Hierarchie nachbilden. Ein Archiv ist eine einzelne Datei, die mehrere Dateien (komprimiert oder unkomprimiert) umfasst. Dies ermöglicht einen schnelleren Zugriff auf die Dateien, ist platzsparend (einzelne kleinere Dateien benötigen mehr Speicherplatz) und übersichtlicher. Archivdateien sind z. B. die Dateien `... \java\jdk-9\lib\tools.jar` und `... \java\jdk-9\jre\lib\rt.jar`. In dem Archiv `rt.jar` finden Sie auch das Package `java.lang` wieder. Dateien mit der Dateierweiterung `.jar` können Sie z. B. mit Programmen wie dem PowerArchiver oder Winzip öffnen (das Format der `*.jar`-Dateien ist ein Zip-Format und dient hier dem Zusammenfassen der Dateien).



In `*.jar`-Archiven befinden sich noch weitere Zusatzinformationen, die nur über eine `*.jar`-Datei korrekt von Java verwendet werden können. Aus diesem Grund können Sie `*.jar`-Dateien nicht ohne Weiteres entpacken und dann auf diese Weise nutzen.

In frühen Versionen des JDK wurden die Archive mit der Endung `.zip` versehen, z. B. `classes.zip`.

9.6 Module

Das Java 9 Platform Module System (JPMS)

Mit der Version 9 von Java steht mit dem Java 9 Platform Module System (JPMS) eine völlig neue Möglichkeit der Strukturierung und Modularisierung von komplexen Java-Anwendungen zur Verfügung. Die Umsetzung dieses Projekts (ursprünglicher Projektname: „Jigsaw“) erfolgte über viele Jahre und war ursprünglich bereits für Java in der Version 7 vorgesehen. Bedingt durch die Tatsache, dass damit sehr tiefgreifende Veränderungen in ein seit über 20 Jahren existierendes System verbunden waren, hat sich die Implementierung verzögert und kann mit der Einführung auch noch nicht als abgeschlossen betrachtet werden. Sowohl die Abwärtskompatibilität zu bestehenden Projekten, die Praktikabilität der Überführung von bestehenden Anwendungen in die neue Umsetzungsform mit Modulen als auch die anspruchsvollen Ziele des Projekts sorgten sowohl für die lange Umsetzungsdauer als auch dafür, dass es eine Reihe von Punkten gibt, die in zukünftigen Versionen noch weiter verbessert und angepasst werden.




Das Modulsystem verfolgt folgende Ziele:

- ✓ Bereitstellung von Mechanismen zur Definition von Abhängigkeiten zwischen Modulen, sowohl zum Zeitpunkt des Kompilierens als auch der Ausführung
- ✓ Verbesserung der Sicherheit von Anwendungen durch eine stärkere Kapselung von Komponenten
- ✓ Skalierbarkeit von Anwendungen durch die Bereitstellung von für eine Anwendung passgenaue Anwendungsumgebung
- ✓ Java interne Klassen werden für einen nicht gewünschten Zugriff und eine direkte Verwendungen in Anwendungen verborgen
- ✓ Steigerung der Performance der Anwendungen; die Optimierung der Laufzeit durch die Java Virtual Machine (JVM) führt zu besseren Resultaten, wenn Typzuordnungen und Gültigkeiten auf einzelne Module bekannt sind

Skalierbarkeit der JDK

Im Gegensatz zu allen Versionen vor Java 9, welche einen kompletten unteilbaren Block darstellten, besteht Java nun selbst aus ca. 100 Modulen, die nicht alle einer Laufzeitumgebung zugeordnet werden müssen. Somit kann für eine Anwendung passgenau eine Laufzeitumgebung erstellt werden, welche nur die Module des JDK enthält, die sie auch benötigt. Dies führt zu deutlich geringeren Größen einer Anwendung zum Zeitpunkt Ihrer Ausführung.

Die Anzeige der einzelnen Module des JDK kann nicht nur in der Dokumentation sondern auch auf der Java-Konsole erfolgen:

- ▶ Öffnen Sie die Kommandozeile über  .
- ▶ Starten Sie die Konsole. Geben Sie dazu `cmd` ein und betätigen Sie die Taste .
- ▶ Wechseln Sie mit dem Befehl `cd\programme\java\jdk-9\bin` ins Unterverzeichnis `bin` der Java-Installation.
- ▶ Mit dem Aufruf des Befehls `java --list-modules` erhalten Sie eine Übersicht der aktuell vorhandenen Module des JDK. Dabei wird für jedes Modul am Ende seines Namens die Versionsnummern angezeigt. `@9` steht für die Version Java 9.

```

Auswählen C:\Windows\system32\cmd.exe
C:\Programme\java\jdk-9\bin>java --list-modules
java.activation@9
java.base@9
java.compiler@9
java.corba@9
java.datatransfer@9
java.desktop@9
java.instrument@9
java.jnlp@9
java.logging@9
java.management@9
java.management.rmi@9
java.naming@9
java.prefs@9
java.rmi@9
java.scripting@9
java.se@9
java.se.ee@9
java.security.jgss@9
java.security.sasl@9
java.smartcardio@9

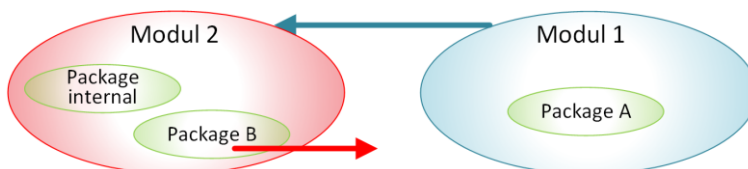
```

Moduleigenschaften

Stellung und Umfang von Modulen

Module sind im Vergleich zu Packages eine höhere Form der Aggregation. Im Gegensatz zu den Packages, welche lose nebeneinander existieren, kann für Module eindeutig festgelegt werden, welche Komponenten (Packages, Klassen und Ressourcen wie beispielsweise XML-Dateien und Bilder) zur Anwendungen gehören.

Ein Modul umfasst immer genau eine fachliche oder technische Komponente einer Anwendung, welche sich aus mehreren einzelnen Modulen zusammensetzt. Es kann exakt bestimmt werden, ob ein Modul ein anderes lesen darf und welchen Programmcode dieses wiederum für das lesende Modul sichtbar macht. So kann ein Modul ein Package freigeben, weitere aber verbergen, so dass diese nur für die interne Nutzung bereitstehen.



Definition von Zugriffen zwischen zwei Modulen



Die Zugriffsmodifizierer `public`, `private` und `protected` gelten unverändert, sind jedoch dem Sichtbarkeitsprinzip der Module untergeordnet. Darf ein Modul ein anderes nicht lesen hat es keinen Zugriff auf dessen Packages, auch wenn dieses mit `public` definierte Komponenten enthält.

Modulaufbau

Modularisierte Anwendungen besitzen entsprechend den allgemeinen Konventionen folgende Verzeichnisstruktur:

①	<i>Kap09BsplApp</i>
②	<i>src</i>
③	<i>com.herdt.modA</i>
④	<i>com</i>
⑤	<i>herdt</i>
	<i>packageA</i>
	<i>Java-Klassen</i>
④	<i>module-info.java (Beschreibungsdatei)</i>

Prinzipieller Aufbau der Verzeichnisstruktur einer modularisierten Anwendung

- ① Die oberste Ebene ist der Name der Anwendung.
- ② Die zweite Ebene bildet das Verzeichnis *src*, das den gesamten Programmcode der Anwendung enthält.
- ③ Die dritte Ebene sind die Modulnamen, welche die Wurzelverzeichnisse der Module bilden.
- ④ Jedes der Modulwurzelverzeichnisse enthält die oberste Verzeichnisebene der Packages sowie die Modulbeschreibungsdatei.
- ⑤ Innerhalb der Verzeichnisstrukturen der Packages befinden sich die einzelnen Java-Klassen.

Modulbeschreibung

Ein Modul wird durch eine Modulbeschreibung in Form einer Datei *module-info.java* definiert.

```
module <eindeutigerName> {  
    ...  
}
```

- ✓ Die Modulbeschreibung beginnt mit dem Schlüsselwort `module`.
- ✓ Hinter diesem steht ein eindeutiger Name, gefolgt von den in geschweiften Klammern eingefassten Zeilen der Beschreibung.

Als Bestandteil des Modulnamens kann, analog zur Packagestruktur, der Internetdomainname Ihres Unternehmens in umgekehrter Reihenfolge verwendet werden, also beispielsweise *com.herdt.modul1*. Ist der Modulname nicht eindeutig, bricht das Kompilieren mit einer Fehlermeldung ab. Findet die Laufzeitumgebung während der Ausführung namensgleiche Module, wird eine Ausnahme ausgelöst.

Modul- und Packagenamen können identisch sein, da diese unterschiedlich verwaltet werden.

Die Beschreibung enthält

- ✓ den Namen des Moduls,
- ✓ die Festlegung der Abhängigkeiten des Moduls, also die Angabe von welchen anderen Modulen das Modul abhängt,
- ✓ Angaben, welche seiner Packages das Modul als sichtbar definiert; wird ein Package nicht explizit als sichtbar definiert, ist es für jedes andere Modul verborgen,
- ✓ welche Services es anbietet bzw. nutzt,
- ✓ für welche Module es die Verwendung von Reflections auf seine eigenen Packages erlaubt (Reflections können in Java verwendet werden um über Klassen Informationen auszulesen; das Thema ist nicht Gegenstand dieses Buches).

Innerhalb der Beschreibung werden über die in der Tabelle aufgelisteten Schlüsselwörter die einzelnen Direktiven zur Beschreibung des Moduls festgelegt. Im Ausnahmefall kann die Modulbeschreibung zwischen den beiden geschweiften Klammern leer sein.

<code>exports</code> <code>exports ... to</code>	Spezifiziert die Packages des Moduls, welche von anderen Modulen gelesen werden dürfen; damit stehen die als <code>public</code> gekennzeichneten Typen dieses Packages in den lesenden Modulen zur Verfügung Durch den Zusatz <code>to</code> kann der Export auf ein oder mehrere Module (Angabe als kommaseparierte Liste) beschränkt werden
<code>provides ... with</code>	Gibt an, dass das Modul einen Service bereitstellt
<code>open</code> <code>opens</code> <code>opens ... to</code>	Steuert die Auswertung des Moduls für Reflections, es können ein gesamtes Modul (<code>open</code>) oder ausgewählte Packages (<code>opens ...</code>) freigegeben werden; die Freigabe wiederum kann für alle oder ebenfalls nur für ausgewählte Module (<code>to</code>) erfolgen
<code>requires</code> <code>requires transitive</code>	Legt die Abhängigkeit von einem anderen Modul und damit den lesenden Zugriff auf dieses fest, jede Abhängigkeit muss einzeln definiert werden durch den Zusatz <code>transitive</code> wird sichergestellt, dass Module die vom aktuellen Modul selbst abhängig sind, dessen Abhängigkeiten beachten
<code>uses</code>	Gibt einen Service an, welchen das Modul nutzt

Classpath und Modulpfad

Bis Java 8 wurden die aufgerufenen und verwendeten Klassen über den Classpath lokalisiert. Mit Java 9 steht alternativ dazu der Modulpfad zur Verfügung. Dieser steht für das oder die Verzeichnisse, in welchen sich das oder die Module einer Anwendung befinden. Wird auf die Modularisierung einer Anwendung nach dem JPMS verzichtet, wirken die Regularien und Zugriffsmechanismen wie gewohnt. Damit ist auch die Kompatibilität bestehender Anwendungen in einer Java 9-Umgebung gewährleistet.

Prinzipiell ist die Verwendung des Modulpfads dem des Classpath vorzuziehen, da man mit dem Modulpfad eine Schwäche des Classpath-Systems – dem Verwenden der ersten gefundenen einer mehrfach in unterschiedlichen Versionen auf einem System vorhandenen Klasse – umgeht.

Werden Classpath und Modulpfad parallel verwendet, nutzt die Laufzeitumgebung zur Suche nach Klassen immer zuerst den Modulpfad und erst dann den Classpath.

Die Angabe des Modulpfads erfolgt bei der Kompilierung bzw. Ausführung der Anwendung.

9.7 Anwendung des Modulsystems

Beispiel

Anwendungsaufbau

Die Beispielanwendung besteht aus zwei Modulen mit insgesamt 3 Packages. Die Abbildung verdeutlicht die Struktur der Verzeichnisse und Ablageorte der einzelnen Dateien.



Modul 1

Das Modul *kap09modA* umfasst seine Beschreibungsdatei *module-info.java* sowie im Package *com.herdtkap09pgckA* die Hauptklasse der Anwendung *UseWelcome.java*.

```

package com.herdtkap09pgckA;
① import com.herdtkap09pgckB.Welcome;
public class UseWelcome
{
    public static void main(String args[])
    {
        Welcome welcome = new Welcome();
        System.out.println( welcome.sayWelcome() );
        System.out.println ("Ende");
    }
}
②

```

Klasse *com.herdtkap09pgckA.UseWelcome.java*

- ① Die Klasse `com.herdt.kap09pgckB.Welcome` des Moduls wird importiert.
- ② Ein Objekt der importierten Klasse wird deklariert und seine Methode `sayWelcome()` ausgeführt.

```

① module com.herdt.kap09modA {
②     requires com.herdt.kap09modB;
③     requires java.base;
}
```

Die Beschreibungsdatei `module-info.java` des Moduls `kap09modA`

- ① Die Modulbeschreibung beginnt immer mit dem Schlüsselwort `module`, gefolgt vom Modulnamen und dem Beschreibungsbereich in geschweiften Klammern.
- ② Über die Angabe des Schlüsselworts `requires` wird die Abhängigkeit des Moduls vom Modul `com.herdt.kap09modB` bestimmt. Auf das angegebene Modul erfolgt ein lesender Zugriff.
- ③ Jedes Modul benötigt lesenden Zugriff auf das Modul `java.base`. Diese Definition ist deshalb implizit in jeder Modulbeschreibungsdatei vorhanden und kann auch weggelassen werden.

Modul 2

Das Modul `com.herdt.kap09modB` umfasst seine Beschreibungsdatei `module-info.java` sowie zwei Packages. Im Package `com.herdt.kap09pgckB` ist die Klasse der Anwendung `Welcome.java` definiert, welche von `UseWelcome.java` des Moduls `com.herdt.kap09modA` genutzt werden soll. Das zweite Package `com.herdt.kap09pgckC` beinhaltet die Klasse `Intern.java`, welche für das Modul `com.herdt.kap09modA` nicht sichtbar sein soll.

```

① package com.herdt.kap09pgckB;
   public class Welcome
   {
       public String sayWelcome()
       {
           return ("Willkommen!");
       }
   }
```

Klasse `com.herdt.kap09pgckB.Welcome.java`

- ① Die Klasse umfasst eine öffentliche Methode zur Rückgabe eines Textes.

```

module com.herdt.kap09modB {
①     exports com.herdt.kap09pgckB;
}
```

Die Beschreibungsdatei `module-info.java` des Moduls `kap09modB`

- ① Mit dem Schlüsselwort `exports` wird bestimmt, dass das Package `com.herdt.kap09pgckB` von Modulen, welche eine Abhängigkeit vom Modul `com.herdt.kap09modB` definieren, gelesen werden darf. Innerhalb des Packages wirken wie gewohnt die Zugriffsmodifizierer. Wäre die Klasse `Welcome` nicht als `public` definiert, wäre sie für Packages anderer Module trotz der `exports`-Angabe in der Beschreibungsdatei nicht sichtbar.

Da das Package `com.herdt.kap09modC` nicht per `exports` als sichtbar definiert wird, ist es vor dem Modul `com.herdt.kap09modA` verborgen.

Kompilieren

Das Kompilieren der Anwendung erfolgt auf der Kommandozeile.

```
javac -d mods --module-path src --module-source-path src ^
src/com.herdt.kap09modA/module-info.java ^
src/com.herdt.kap09modA/com/herdt/kap09pgckA/UseWelcome.java
```

- ✓ Der Kompilierungsvorgang wird wie gewohnt mit `javac` aufgerufen.
- ✓ Über die Option `-d` wird das Verzeichnis festgelegt, in dem der kompilierte Code der Anwendung abgelegt werden soll. Für einen Ordner, welche Module enthält, wird entsprechend der Konvention `mods` verwendet. In diesem Verzeichnis befinden sich dann die Struktur der einzelnen Module, Packages und Klassen.
- ✓ Nach der Option `--module-path` werden Verzeichnisse angegeben, in welchen sich schon kompilierte Module befinden.
- ✓ Mit der Option `--module-source-path` werden die Verzeichnisse mit noch nicht kompilierten Modulen angegeben.
- ✓ Neben den Verzeichnisangaben sind die Beschreibungsdatei und die Hauptklasse der Anwendung anzugeben.
- ✓ Das Zeichen `^` steht für einen Zeilenumbruch bei der Eingabe in Windows. Verwenden Sie unter Linux den Backslash `\`.

```
C:\Windows\system32\cmd.exe

C:\uebung\jav9\kap09BsplApp>javac -d mods --module-path src --module-source-path src ^
Mehr? src/com.herdt.kap09modA/module-info.java ^
Mehr? src/com.herdt.kap09modA/com/herdt/kap09pgckA/UseWelcome.java
```

Fehler in der Definition der Sichtbarkeitsbeziehungen, wie dem Lesen auf nicht exportierte Komponenten eines Moduls oder fehlenden Definitionen von Abhängigkeitsbeziehungen, führen beim Kompilieren zur Fehlermeldung `error: ... does not exist`.

```
C:\Windows\system32\cmd.exe

C:\uebung\jav9\kap09BsplApp>javac -d mods --module-path src --module-source-path src ^
Mehr? src/com.herdt.kap09modA/module-info.java ^
Mehr? src/com.herdt.kap09modA/com/herdt/kap09pgckA/UseWelcome.java
src/com.herdt.kap09modA\com\herdt\kap09pgckA\UseWelcome.java:2: error: package com.herdt.kap09pgckB does not exist
import com.herdt.kap09pgckB>Welcome;
^
```

Ausführen der Anwendung

```
java --module-path mods -m ^
com.herdn.kap09modA/com.herdn.kap09pgckAUseWelcome
```

- ✓ Der Start der Anwendung erfolgt über den Aufruf von `java`.
- ✓ Mit der Option `--module-path` bzw. verkürzt `-p` wird der Modulpfad angegeben.
- ✓ Hinter der Option `-module` bzw. verkürzt `-m` wird der Modulname und die auszuführende Hauptklasse bestimmt. Über diese Angabe wird der Auflösungsprozess gestartet und es werden alle Abhängigkeiten ermittelt und gefunden.


```
C:\Windows\system32\cmd.exe
C:\uebung\jav9\kap09bsp1App>java --module-path mods -m com.herdn.kap09modA/com.herdn.kap09pgckA.UseWelcome
Willkommen!
Ende
```

Verwendung von Packages und Modulen in diesem Buch

In den weiteren Kapiteln des Buches wird der Modulpfad nicht weiter genutzt. Dies würde zum einen durch die erforderlichen Verzeichnisstrukturen und die Beschreibungsdatei(en) nur die Komplexität der Beispiele ohne zusätzlichen Nutzen für diese erhöhen. Zum anderen ist es damit auch möglich, das Buch in Verbindung mit einer älteren Version des JDK ohne Anpassungen und Beachtung von Hinweisen zu nutzen.

9.8 Übung

Packages verwenden

Level		Zeit	ca. 10 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Package erstellen ... ✓ Klassen einem Package zuordnen ... ✓ Eine Klasse importieren 		
Übungsdatei	--		
Ergebnisdatei	--		

1. Erzeugen Sie ein Package mit dem Namen Ihrer Firma oder mit Ihrem Namen, z. B. `com.herdn`. (Es ist nicht üblich, als Top-Level-Domain `.de` zu verwenden.)
2. Fügen Sie in das Package eine Klasse ein, die Ihren Namen (Firmennamen) zurückliefert.
3. Verwenden Sie die Klasse in einem Programm. Setzen Sie dazu die `import`-Vereinbarung und den qualifizierten Klassennamen ein.

10

Interfaces und Adapterklassen

10.1 Interfaces

Ein Interface (Schnittstelle) definiert in Java einen Typ mit folgenden Eigenschaften:

- ✓ Ein Interface enthält nur öffentliche (`public`) Methoden und Konstanten.
- ✓ Bis zur Version Java 7 standen in einem Interface nur die Methodenköpfe. Seit Java 8 kann ein Interface Default-Methoden und statische Methoden enthalten.
- ✓ In Java 9 wurden als Ergänzung zu diesen `private` Methoden innerhalb eines Interfaces erlaubt. Diese ermöglichen die gemeinsame Codeverwendung in statischen bzw. Default-Methoden. Private Methoden müssen innerhalb des Interface implementiert werden und sind für den Verwender des Interfaces nicht sichtbar.
- ✓ Verwendet eine Klasse ein Interface, muss die Klasse **alle** Methoden des Interfaces implementieren, welche nicht als `default` oder `static` gekennzeichnet sind.
- ✓ Eine Klasse kann mehrere Interfaces implementieren.
- ✓ Alle Klassen, die das gleiche Interface implementieren, müssen alle Methoden des Interface bereitstellen. Sie besitzen somit eine gemeinsame Schnittstelle.

Interfaces erstellen

Syntax der `interface`-Definition

```
[public] interface identifier [extends interfaceName1[,  
interfaceName2...]]  
{  
    [public] [static] [final] type identifier1 = value; ①  
    [public] [default|static] type identifier2([parameter]); ②  
    [private] type identifier2([parameter]); ③  
}
```

- ✓ Die Definition eines Interface beginnt mit dem Schlüsselwort `interface`.
- ✓ Mit dem Zugriffsmodifizierer `public` können Sie das Interface als öffentlich deklarieren. Ohne die Angabe des Zugriffsmodifizierers kann das Interface nur innerhalb des Packages verwendet werden.

- ✓ Es folgt der Name des Interface, der sich an die Regeln von Bezeichnern halten muss.
- ✓ Optional können Sie ein Interface – im Gegensatz zu Klassen – mit dem Schlüsselwort `extends` aus einem oder mehreren anderen Interfaces erweitern. Mehrere Interfaces werden durch Kommata getrennt angegeben.
- ✓ In einem Interface können Sie Konstanten ① und Methoden ② definieren.
- ✓ Öffentliche Methoden, welche nicht als `default` oder `static` gekennzeichnet sind, sind automatisch `abstract`. Aus diesem Grund kann diese Angabe entfallen. Diese Methoden enthalten statt eines Methodenrumpfs nur ein Semikolon nach der Parameterliste.
- ✓ Als `default` bzw. `static` gekennzeichnete Methoden in einem Interface können einen Methodenrumpf oder eine vollständige Methodenimplementierung enthalten. In Klassen, welche das Interface implementieren, können diese Methoden überschrieben werden.
- ✓ Private Methoden ③ dienen ausschließlich der Verwendung innerhalb des Interfaces in anderen privaten Methoden oder in Methoden, die als `default` oder `static` gekennzeichnet sind. Sie müssen vollständig implementiert werden.
- ✓ Methoden eines Interfaces sind immer öffentlich. Die Angabe des Zugriffsmodifizierers `public` ist daher nicht notwendig.
- ✓ Die Konstanten in einem Interface sind immer öffentlich und statisch. Die Angabe der Zugriffsmodifizierer `public` und die Modifizierer `static` und `final` können entfallen.
- ✓ Interfaces besitzen keine Konstruktoren.

Wie eine Klasse wird ein Interface gewöhnlich in einer eigenständigen Datei mit dem Interface-Namen als Dateinamen gespeichert.

Beispiel: `com\herdt\java9\kap10\forms\Forms2D.java`

- ① Das Interface `Forms2D` wird definiert.
- ②-③ Das Interface schreibt vor, dass die Klassen, die das Interface implementieren, eine Methode `getArea` zur Flächenberechnung und eine Methode `getPerimeter` zur Berechnung des Umfangs implementieren müssen.

```
package com.herdt.java9.kap10.forms;
/**
 * Beschreibung eines Interface für
 * zweidimensionale Formen
 */
① public interface Forms2D
{
  ② double getArea(); //Flaecheninhalt
  ③ double getPerimeter(); //Umfang
}
```

In dem Interface wird nur der Methodenkopf, aber nicht der Methodenrumpf beschrieben.

Interfaces implementieren

Ein Interface wird verwendet, indem eine Klasse das Interface implementiert. Die Klasse verpflichtet sich damit, alle im Interface definierten abstrakten Methoden zu implementieren. Damit eine Klasse ein Interface implementieren kann, muss bei der Erstellung der Klasse angegeben werden, welches Interface verwendet werden soll.

Syntax

```
[public] class identifier implements interfaceName1[,  
interfaceName2...]  
{  
    ...  
}
```

- ✓ Damit eine Klasse ein Interface implementiert, werden dem Klassennamen das Schlüsselwort `implements` und der Name des Interface angehängt.
- ✓ Eine Klasse kann mehrere Interfaces implementieren. Die Interface-Namen werden mit Komma getrennt hinter `implements` angegeben.
- ✓ Wenn eine Klasse mehrere Interfaces implementiert, muss sie alle abstrakten Methoden dieser Interfaces implementieren. Falls eine Methode von mehreren Interfaces gefordert wird, so müssen Sie diese Methode in der Klasse nur einmal definieren.

Häufige Schreibweise

Häufig wird im Quelltext mit dem Schlüsselwort `implements` eine neue Zeile begonnen. Dies ist vor allem bei der Implementierung mehrerer Interfaces übersichtlicher.

```
[public] class identifier  
    implements interfaceName1[, interfaceName2...]  
{  
    ...  
}
```

Beispiel: `com\herdt\java9\kap10\forms\Rectangle.java` und `com\herdt\java9\kap10\forms\Circle.java`

Das Interface `Forms2D` soll von einer Klasse `Rectangle` und einer Klasse `Circle` implementiert werden.

```
package com.herdt.java9.kap10.forms;  
① public class Rectangle implements Forms2D  
{  
    ...  
    //Implementierung der Interface-Methoden  
    ② public double getArea()  
    {  
        return getWidth() * getLength();  
    }  
    public double getPerimeter()  
    {  
        return 2.0 * (getWidth() + getLength());  
    }  
}
```

Die Klasse `Rectangle`

```
package com.herdt.java9.kap10.forms;
import static java.lang.Math.PI;
① public class Circle implements Forms2D
{
    ...
    //Implementierung der Interface-Methoden
    ② public double getArea()
    {
        return PI * getRadius() * getRadius();
    }
    public double getPerimeter()
    {
        return 2.0 * PI * getRadius();
    }
}
```

Die Klasse *Circle*

- ① Die Klassen *Rectangle* und *Circle* werden definiert und implementieren das Interface *Forms2D*.
- ② Die Klassen *Rectangle* und *Circle* müssen alle im Interface aufgeführten Methoden implementieren.

Java besitzt eine große Anzahl von vordefinierten Interfaces. Eine Beschreibung der vordefinierten Interfaces finden Sie in der Dokumentation zu Java.

Abstrakte Klassen und Interfaces im Vergleich

Wenn Sie eine abstrakte Klasse definieren, die nur abstrakte Methoden besitzt, können Sie stattdessen ein Interface einsetzen. Die Verwendung von Interfaces hat den Vorteil, dass Sie zusätzlich zur Implementierung eines Interface die Klasse noch von einer anderen Klasse ableiten können. Mit abstrakten Klassen ist dies nicht möglich, da es in Java nur Einfachvererbung gibt. Außerdem kann eine Klasse mehrere Interfaces implementieren. Abstrakte Klassen haben jedoch den Vorteil, dass Sie hier Variablen einfügen können. Dies ist bei Interfaces nicht möglich.

Interfaces nutzen

- ✓ Sie können Referenzvariablen vom Typ eines Interface definieren.
- ✓ Den Referenzvariablen können dann aufgrund der Typkompatibilität Objekte zugewiesen werden, deren Klasse das Interface implementiert hat.
- ✓ Alle Methoden, die im Interface vorgeschrieben sind, können Sie für diese Objekte anwenden.
- ✓ Es ist sichergestellt, dass alle Methoden, die im Interface vorgeschrieben sind, in der Klasse, die das Interface implementiert, beschrieben sind.

```
interfaceName
identifizier;
```

Beispiel: `com\herdt\java9\kap10\UseForms2D.java`

```
package com.herdt.java9.kap10;
import com.herdt.java9.kap10.forms.*;
class UseForms2D
{
    public static void main(String[] args)
    {
        ① Forms2D firstForm = new Rectangle(2.0, 3.5);
        Forms2D secondForm = new Circle(3.0);
        ② System.out.printf("Form 1:%nFlächeninhalt: %g%nUmfang: %g%n",
                           firstForm.getArea(),
                           firstForm.getPerimeter());
        System.out.printf("Form 2:%nFlächeninhalt: %g%nUmfang: %g%n",
                           secondForm.getArea(),
                           secondForm.getPerimeter());
    }
}
```

Variablen vom Typ eines Interface verwenden

- ① Die Referenzvariablen `firstForm` und `secondForm` vom Typ des Interface `Forms2D` werden definiert. Ihnen wird ein Objekt vom Typ `Rectangle` bzw. `Circle` zugewiesen. Dies ist möglich, da beide Klassen das Interface `Forms2D` implementieren.
- ② Die im Interface vorgeschriebenen und in der jeweiligen Klasse (`Rectangle` bzw. `Circle`) implementierten Methoden werden verwendet.

Der Compiler verwendet die statische Bindung der Variablen `firstForm` und `secondForm` und betrachtet sie als Variablen vom Typ des Interface `Forms2D`. Die verwendeten Methoden (hier `getArea` und `getPerimeter`) sind im Interface vorgeschrieben.

Der Interpreter stellt eine dynamische Bindung zwischen der Referenzvariablen vom Typ des Interface (`Forms2D`) und dem mit `new` erzeugten Objekt (`Rectangle` bzw. `Circle` ①) her.

Datentyp zur Sicherheit überprüfen

Über den Operator `instanceof` können Sie überprüfen, ob ein Objekt kompatibel zu einer Schnittstelle (oder einer Klasse) ist.

Im vorherigen Beispiel implementiert die Klasse `Rectangle` das Interface `Forms2D`. Die beiden ersten der nebenstehenden Ausdrücke liefern daher den Wert `true`.

```
if (objectName instanceof
    interfaceName)
    ...
```

```
firstForm instanceof Rectangle
//true
firstForm instanceof Forms2D
//true
firstForm instanceof Circle
//false
```

10.2 Adapterklassen

Was sind Adapterklassen?

Häufig benötigt eine Klasse nur einige Methoden eines Interfaces. Bis einschließlich Java 7 war es aber erforderlich, dass die Klasse **alle** Methoden des Interfaces implementiert. Um nicht bei jeder Verwendung eines Interfaces in einer Klasse alle Methoden implementieren zu müssen, können Sie eine sogenannte Adapterklasse erstellen. Eine **Adapterklasse** implementiert ein Interface mit leeren Methodenrumpfen bzw. mit Methodenrumpfen, die einen Standardwert zurückliefern.

Statt das Interface zu implementieren, erweitert die Klasse anschließend die Adapterklasse und überschreibt nur die gewünschten Methoden.

- ✓ Eine Adapterklasse kann mehrere Interfaces implementieren und muss dann alle abstrakten Methoden dieser Interfaces implementieren.
- ✓ Adapterklassen können außer den vom Interface implementierten Methoden weitere Methoden besitzen.
- ✓ Adapterklassen werden intensiv bei der Programmierung von grafischen Anwendungen eingesetzt.
- ✓ Java stellt zu verschiedenen Interfaces auch Adapterklassen zur Verfügung (z. B. WindowAdapter im Package `java.awt.event`).

Der Name für eine Adapterklasse wird gewöhnlich aus dem Namen bzw. Namensanfang des Interface und dem Wort „Adapter“ zusammengesetzt (z. B. MouseAdapter im Package `java.awt.event`).

Seit Java 8 besteht die Möglichkeit, **Default-Methoden** mit leeren Methodenrumpfen bzw. mit kompletten Methodenimplementierungen im Interface zu erstellen. Damit entfällt die Notwendigkeit der Erstellung von Adapterklassen zur Arbeitserleichterung. Deren Bedeutung wird damit zukünftig geringer. Viele Bestandteile von Java (z. B. Collection/AbstractCollection im Collection Framework, WindowListener/WindowAdapter in der Programmierung von Benutzeroberflächen) nutzen jedoch die Technik der Adapterklassen. Deren Kenntnis ist deshalb auch weiterhin wichtig.

Eine Adapterklasse erstellen

Beispiel des Interface: `com\herdt\java9\kap10\forms\Forms2D2.java`

Das Interface Forms2D2 erweitert das Interface Forms2D aus dem vorherigen Abschnitt.

Das Interface definiert eine weitere Methode `turn90`, mit der das Grafikobjekt um 90 Grad gedreht werden kann.

```
package com.herdt.java9.kap10.forms;
public interface Forms2D2 extends Forms2D
{
    // geerbte Methoden
    // double getArea();           //Flaecheninhalt
    // double getPerimeter();      //Umfang
    void turn90();                //um 90 Grad drehen
}
```

Die Methode besitzt keine Parameter und keinen Rückgabewert.

Beispiel der Adapterklasse: *com\herdt\java9\kap10\forms\Forms2D2Adapter.java*

Im Folgenden wird eine Adapterklasse für das Interface Forms2D2 definiert.

```
package com.herdt.java9.kap10.forms;
① public class Forms2D2Adapter implements Forms2D2
{
  ② public double getArea() {return 0.0;}           //Flaeche berechnen
  ③ public double getPerimeter() {return 0.0;}      //Umfang berechnen
  ④ public void turn90() {}                        //um 90 Grad drehen
}
```

- ① Die Adapterklasse Forms2D2Adapter implementiert das Interface Forms2D2. Ansonsten hat die Klasse keine weiteren Aufgaben.
- ②-④ Die Methoden werden mit leeren Rümpfen bzw. mit Rümpfen, die einen Standardwert zurückgeben, implementiert.



Beachten Sie, dass die beiden Methoden ② und ③ einen Rückgabewert besitzen. Daher muss der Rumpf mindestens eine entsprechende `return`-Anweisung enthalten.

Adapterklassen anwenden

Die Adapterklasse wird als Basisklasse für weitere Klassen verwendet. Diese Klassen implementieren somit nicht das Interface, sondern erweitern die Adapterklasse. Beachten Sie, dass diese Klassen dann keine weiteren Klassen erweitern können. In Java ist nur eine Einfachvererbung möglich.

Beispiel zur Verwendung der Adapterklasse:

com\herdt\java9\kap10\forms\Circle2.java

Ein Kreis ändert seine Form nicht, wenn er gedreht wird. Daher wird die Methode `turn90` von der Klasse `Circle2` nicht benötigt. Die Klasse überschreibt nur die benötigten Methoden der Adapterklasse.

```
① public class Circle2 extends Forms2D2Adapter
import static java.lang.Math.PI;
{
  ...
  //Implementierung der benoetigten Adapter-Methoden
  ② public double getArea()
  {
    return PI * getRadius() * getRadius();
  }
}
```

```
③ public double getPerimeter()  
{  
    return 2.0 * PI * getRadius();  
}
```

- ① Die Klasse `Circle2` erweitert die Adapterklasse `Forms2D2Adapter`.
②-③ Für die Klasse `Circle2` werden nur die Methoden `getArea` und `getPerimeter` benötigt. Sie überschreibt die beiden Methoden.

Die Methode (`turn90`) wird nicht benötigt und daher auch nicht überschrieben. Die Methode ist aber aufgrund der Vererbung vorhanden, denn sie wurde in der Adapterklasse implementiert.

10.3 Direkte Methodenimplementierung im Interface

Default- und statische Methoden

Default-Methoden

Default-Methoden eines Interfaces sind Instanzmethoden. Sie werden durch das Schlüsselwort `default` gekennzeichnet. Jede Klasse, die das Interface implementiert, erbt die Methoden und kann sie – wenn notwendig – überschreiben. Dies kann nicht unterbunden werden, da Default-Methoden nicht als `final` gekennzeichnet werden können. Der Aufruf der Methode erfolgt über eine Instanz der Klasse.



Auch wenn ein Interface eine Methodenimplementierung besitzt, kann dennoch keine Instanz eines Interfaces erstellt werden. Dies ist nur von Klassen, welche das Interface implementieren, möglich.

Default-Methoden im Interface bieten folgende Vorteile:

- ✓ Bis einschließlich Java 7 ergab sich bei einer nachträglichen Erweiterung eines Interfaces mit zusätzlichen Methoden die Notwendigkeit der Implementierung in allen das Interface implementierenden Klassen. Um diesen Aufwand zu umgehen, wurden häufig neue Interface erstellt. Diese erbten vom bereits vorhandenen Interface und beinhalteten die zusätzlichen Methoden. Im Ergebnis dieser Vorgehensweise entstanden teilweise komplexe Interface-Hierarchien, was die Übersichtlichkeit über die einzelnen Komponenten erheblich beeinträchtigte.
Seit Java 8 ermöglichen die Default-Methoden das nachträgliche Hinzufügen einer Methode zu einem Interface ohne Beeinträchtigung der das Interface implementierenden Klassen.
- ✓ Die Verwendung von Default-Methoden erspart die Notwendigkeit, dass eine das Interface implementierende Klasse, auch dann, wenn sie nur eine Methode benötigt, alle im Interface enthaltenen Methoden implementieren muss. Die bisher übliche Erstellung von Adapterklassen zur Vermeidung dieses Aufwands ist damit nicht mehr notwendig.
- ✓ Die Default-Methoden eines Interfaces können aufeinander zugreifen. Damit kann Logik direkt im Interface ohne Erstellung einer abstrakten Klasse abgebildet werden.

Statische Methoden

Statische Methoden eines Interfaces sind Klassenmethoden. Alle Instanzen einer Klasse nutzen die Methode gemeinsam. Im Interface-Kontext werden statische Methoden häufig als Hilfsmethoden für Default-Methoden verwendet (beispielsweise im Interface `java.util.Comparator`). Statische Interface-Methoden werden durch das Schlüsselwort `static` gekennzeichnet.

Nach dem Implementieren des Interfaces durch eine Klasse ist die statische Methode weiterhin ein Bestandteil des Interfaces, in welchem sie definiert wurde. Der Aufruf erfolgt deshalb immer über den Namen des Interfaces und nicht über den der Klasse.

Beispiel zur Verwendung von Default- und statischen Interface-Methoden:
com\herdt\java9\kap10\Text.java
com\herdt\java9\kap10\PrintText.java

Das Interface `Text` definiert zwei Default- und eine statische Methode.

```
package com.herdt.java9.kap10;

public interface Text
{
    ① public default String supplyText()
    {
        return "Guten Tag.";
    }
    ② public default void writeText()
    {
        System.out.println("Noch da?");
    }
    ③ public static void writeTextStatic()
    {
        System.out.println("Auf Wiedersehen!");
    }
}
```

- ① Die Default-Methode `supplyText()` liefert als Rückgabewert einen String.
- ② Die Default-Methode `writeText()` gibt einen String auf der Konsole aus.
- ③ Die statische Methode `writeTextStatic()` gibt ebenfalls einen String auf der Konsole aus.

Die Klasse `PrintText` implementiert das Interface `Text` und nutzt deren Methoden. Da für diese bereits eine Implementierung im Interface erfolgt ist, können Sie sofort genutzt werden. Zusätzlich wird in der Klasse die statische Methode `writeTextStatic()` überschrieben.

```
package com.herdt.java9.kap10;

① public class PrintText implements Text
{
    ② public static void writeTextStatic()
    {
        System.out.println("Tschüss!");
    }
    public static void main (String [] args)
    {
        ③ PrintText pt = new PrintText();
        ④ System.out.println(pt.supplyText());
        ⑤ Text.writeTextStatic();
        ⑥ pt.writeText();
        ⑦ pt.writeTextStatic();
    }
}
```

- ① Über das Schlüsselwort `implements` wird das Interface `Text` implementiert.
- ② Die statische Methode des Interfaces wird überschrieben.
- ③ Eine Instanz der Klasse `PrintText` wird erstellt.
- ④ Über die Instanzvariable wird die Default-Methode `supplyText()` aufgerufen und deren Rückgabewert auf der Konsole ausgegeben.
- ⑤ Der Aufruf der statischen Methode `writeTextStatic()` erfolgt über den Namen des Interfaces.
- ⑥ Über die Instanzvariable wird die Default-Methode `writeText()` aufgerufen.
- ⑦ Der Aufruf der statischen Methode `writeTextStatic()` erfolgt die Instanzvariable. Daher wird die in der Klasse überschriebene Version genutzt.

Guten Tag.
Auf Wiedersehen!
Noch da?
Tschüss!

Die Ausgabe des Programms

Vererbung und Mehrdeutigkeit

Es ist möglich, dass eine Klasse mehrere Interfaces implementiert, welche die gleiche Default-Methode besitzen oder dass die Klasse ein Interface implementiert und zusätzlich von einer anderen Klasse erbt, welche beide die gleiche Methode definieren. In beiden Fällen muss eindeutig geklärt sein, welche Methode zu verwenden ist.

Dabei wird nach folgenden Regeln entschieden:

Regel 1: Wird eine Default-Methode überschrieben, wird diejenige genutzt, welche zur aktuellen Klasse am nächsten steht.

Regel 2: Methoden in Superklassen haben Vorrang vor Default-Methoden eines Interfaces.

Regel 3: Existiert die gleiche Default-Methode in verschiedenen zu implementierenden Interfaces, welche nicht in einer Vererbungsbeziehung stehen, so muss im Programm eindeutig entschieden werden, welche Default-Methode genutzt werden soll. Erfolgt dies nicht, kommt es zu einer Fehlermeldung beim Compilieren.

```
C:\uebung\jav9\com\herdt\java8\kap10\UseExampleIF2.java:3:  
error: class UseExampleIF2 inherits unrelated defaults for  
saySomething() from types ExampleIF and AnotherIF
```

Fehlermeldung des Compilers für das nachfolgende Beispiel bei fehlender Steuerung durch das Programm.

Beispiel Regel 1 – Überschriebene Methoden:

com\herdt\java9\kap10\ExampleSuperIF.java

com\herdt\java9\kap10\ExampleIF.java

com\herdt\java9\kap10\UseExampleIF.java

Das Interface ExampleSuperIF definiert die Default-Methode `saySomething()`.

```
package com.herdt.java9.kap10;  
  
public interface ExampleSuperIF  
{  
    ① public default String saySomething()  
    {  
        return "Hallo aus dem Interface ExampleSuperIF";  
    }  
}
```

① Definition der Default-Methode `saySomething()`.

Das Interface ExampleIF erbt von ExampleSuperIF und überschreibt die Default-Methode.

```
package com.herdt.java9.kap10;  
  
public interface ExampleIF extends ExampleSuperIF  
{  
    ① public default String saySomething()  
    {  
        return "Hallo aus dem Interface ExampleIF";  
    }  
}
```

① Überschreiben der Default-Methode `saySomething()`.

Die Klasse `UseExampleIF` implementiert das Interface `ExampleIF`.

```
package com.herdt.java9.kap10;

public class UseExampleIF implements ExampleIF
{
    public static void main (String[] args)
    {
        UseExampleIF instance = new UseExampleIF();
        ① System.out.println(instance.saySomething());
    }
}
```

- ① Beim Aufruf der Methode `saySomething()` wird die Implementierung des Interfaces `ExampleIF` verwendet, da in diesem die Methode des übergeordneten Interfaces `ExampleIF` überschrieben wurde. Die Ausgabe lautet entsprechend *"Hallo aus dem Interface ExampleIF"*.

Beispiel Regel 2 – Superklasse vor Interface:

com\herdt\java9\kap10\ExampleIF.java

com\herdt\java9\kap10\ExampleClass.java

com\herdt\java9\kap10\UseExampleIF2.java

Es wird wiederum das Interface `ExampleIF` mit der Default-Methode `saySomething()` verwendet. Die Klasse `ExampleClass` besitzt ebenfalls eine Methode `saySomething()`.

```
package com.herdt.java9.kap10;

public class ExampleClass
{
    ① public String saySomething()
    {
        return "Hallo aus der Klasse ExampleClass";
    }
}
```

- ① Definition der Methode `saySomething()`.

Die Klasse `UseExampleIF2` erbt von der Klasse `ExampleClass` und implementiert das Interface `ExampleIF`.

```
package com.herdt.java9.kap10;

public class UseExampleIF implements ExampleIF
{
    public static void main (String[] args)
    {
        UseExampleIF instance = new UseExampleIF();
    }
}
```

```

①      System.out.println(instance.saySomething());
      }
    }

```

- ① Beim Aufruf der Methode `saySomething()` wird die Implementierung der Klasse `ExampleClass` verwendet. Die Ausgabe lautet entsprechend *"Hallo aus der Klasse ExampleClass"*.

Beispiel Regel 3 – Entscheidung im Programm:

com\herdt\java9\kap10\ExampleIF.java

com\herdt\java9\kap10\AnotherIF.java

com\herdt\java9\kap10\UseExampleIF3.java

Es wird wiederum das Interface `ExampleIF` mit der Default-Methode `saySomething()` verwendet. Daneben existiert ein weiteres Interface `AnotherIF`. Dieses besitzt ebenfalls eine Default-Methode `saySomething()`.

```

package com.herdt.java9.kap10;

public interface AnotherIF
{
①    public default String saySomething()
    {
        return "Hallo aus dem Interface AnotherIF";
    }
}

```

- ① Definition der Default-Methode `saySomething()`.

Die Klasse `UseExampleIF3` implementiert sowohl das Interface `ExampleIF` als auch das Interface `AnotherIF`. Durch das Überschreiben der Methode `saySomething()` wird eindeutig geklärt, welche Methodenimplementierung zu verwenden ist.

```

package com.herdt.java9.kap10;

public class UseExampleIF3 implements ExampleIF, AnotherIF
{
①    public String saySomething()
    {
②        return AnotherIF.super.saySomething();
    }


    public static void main (String[] args)
    {
        UseExampleIF3 instance = new UseExampleIF3();
③        System.out.println(instance.saySomething());
    }
}

```

- ① Die Methode `saySomething()` wird überschrieben.
- ② Der Rückgabewert der Methode wird durch den Aufruf der Default-Methode aus dem Interface `AnotherIF` bestimmt.
- ③ Beim Aufruf der Methode `saySomething()` wird die durch Überschreibung in der aktuellen Klasse `UseExampleIF3` erstellte Implementierung verwendet. Die Ausgabe lautet entsprechend *"Hallo aus dem Interface AnotherIF"*.

10.4 Übung

Interfaces und Adapterklassen

Level		Zeit	ca. 20 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Interface erstellen ✓ Adapterklasse zum Interface erstellen ✓ Klassen erstellen, welche die Adapterklasse verwenden 		
Übungsdatei	--		
Ergebnisdateien	<i>Media.java, MediaAdapter.java, Picture.java, Video.java, Audio.java, TestMedia.java</i>		

1. Erstellen Sie ein Interface `Media` mit folgenden drei Methoden:
 - ✓ `play`
 - ✓ `stop`
 - ✓ `display`
2. Erzeugen Sie eine Adapterklasse zum Interface `Media`.
3. Erstellen Sie drei Klassen `Picture`, `Video`, `Audio`, in denen Sie die benötigten Methoden implementieren. Die Methoden sollen hier einen Text ausgeben: Beispielsweise gibt die Methode `play` der Klasse `Video` den Namen der Klasse einschließlich Package und den Text *Play video* aus (vgl. Abbildung).

```

Medium 1:
class com.herdt.java9.kap09.Picture: Display picture
Medium 2:
class com.herdt.java9.kap09.Video: Display video
class com.herdt.java9.kap09.Video: Play video
class com.herdt.java9.kap09.Video: Video stopped
Medium 3:
class com.herdt.java9.kap09.Audio: Play audio
class com.herdt.java9.kap09.Audio: Audio stopped

```

Die Ausgabe des Programms

4. Schreiben Sie ein Programm `TestMedia`, in dem Sie die drei Klassen nutzen.

11

Mit Strings und Wrapper-Klassen arbeiten

11.1 Die Klasse `String`

Unveränderliche Strings

Für die Arbeit mit Zeichenketten gibt es in Java die Klasse `String`. `String` ist kein primitiver Datentyp, sondern ein Referenz-Typ.

- ✓ `String` ist eine Klasse und stellt den Datentyp `String` zur Verfügung, somit können Variablen Objekte vom Typ `String` referenzieren.
- ✓ Die Klasse `String` besitzt Methoden zur Arbeit mit Zeichenketten.
- ✓ Obwohl es sich bei `String` um eine Klasse handelt, müssen Sie Objekte vom Typ `String` nicht mit `new` erzeugen. Die Verwendung von `new` ist aber erlaubt.
- ✓ Die Klasse `String` ist als `final` definiert, sodass Sie von ihr keine weiteren Klassen ableiten können.
- ✓ Die Klasse `String` besitzt keine Setter-Methoden zum Ändern der gespeicherten Zeichenkette. Daher werden `String`-Objekte auch **immutable** (unveränderlich) genannt.
- ✓ Der Inhalt und die Länge eines `String`-Objekts sind immer konstant. Wenn Sie einer Referenzvariablen auf ein `String`-Objekt eine neue Zeichenkette zuweisen, wird automatisch ein neues `String`-Objekt erzeugt und das alte freigegeben.
- ✓ Die Zeichenketten bestehen aus Unicode-Zeichen.
- ✓ Zeichenketten werden in doppelte Anführungszeichen eingeschlossen.
- ✓ Es gibt auch leere Zeichenketten. Ein entsprechender `String` hat die Länge 0 und kann beispielsweise durch zwei direkt aufeinander folgende doppelte Anführungszeichen erzeugt werden (`""`).

Ein String-Objekt erzeugen

Syntax für die Erzeugung eines Strings

- ✓ Sie definieren eine Variable vom Typ `String` ①.
- ✓ Der Variablen weisen Sie ein Literal (Zeichenkette) zu. Dazu geben Sie auf der rechten Seite des Gleichheitszeichens die Zeichenkette an und schließen sie in doppelte Anführungszeichen `" "` ein ②.
- ✓ Sie können die Definition und die Initialisierung des String-Objekts auch in einer Anweisung zusammenfassen ③.

```
String identifier;           ①
identifier = "Text";        ②
String identifier1 = "Text"; ③
```

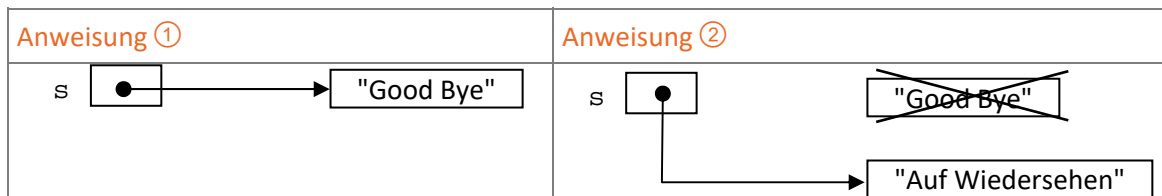
Die Ausgabe des Programms

Einer String-Variablen eine neue Zeichenkette zuweisen

Wenn Sie einer String-Variablen eine neue Zeichenkette zuweisen, wird mit der Zeichenkette ein neues String-Objekt erzeugt. Die Variable verweist auf das neue Objekt.

```
① String s = "Good bye";
...
② String s = "Auf Wiedersehen";
```

Das alte String-Objekt wird nicht mehr referenziert und daher nicht mehr benötigt. Es wird von dem sogenannten Garbage-Collector von Java bereinigt. Der Garbage-Collector läuft bei der Programmausführung automatisch im Hintergrund und gibt den Speicherplatz für nicht mehr verwendete Objekte frei.



Zeilenumbrüche in einen String einfügen

Mit der Escape-Sequenz `\n`, die Sie als Text in einen String eingeben können ("`\n`"), erzeugen Sie bei der Ausgabe des Strings an der entsprechenden Stelle einen Zeilenumbruch.

Strings mit der Methode `System.out.printf` ausgeben: `com\herdt\java9\kap11\PrintfString.java`

Neben der Methode `System.out.println` bzw. `System.out.print` können Sie Strings auch mit der Methode `System.out.printf` ausgeben. Dazu nutzen Sie den Platzhalter `%s`.

```
String text = "Rechteck";
int i = 4;
System.out.printf("Ein %s hat %d Ecken.\n",
                  text, i);
```

```
Ein Rechteck hat 4 Ecken.
```

Die Ausgabe des Programms

11.2 Strings verketten und vergleichen

Strings verketten

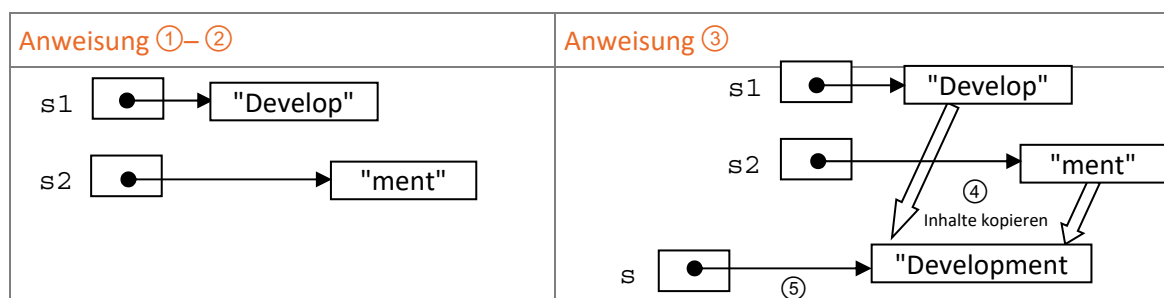
Das Verketteten von Strings können Sie über das Zeichen `+` durchführen. Dabei führt Java bei der folgenden Verkettung die angegebenen Operationen durch:

- ①-② Die String-Variablen `s1` und `s2` verweisen auf String-Objekte mit dem Inhalt `"Develop"` bzw. `"ment"`.

- ④ Aus dem Ergebnis der Verkettung `s1 + s2` wird ein **neues** String-Objekt erzeugt.

Die Variable `s` referenziert den neu erzeugten String `"Development"` ⑤.

```
① String s1 = "Develop";
② String s2 = "ment";
③ String s = s1 + s2;
```



Soweit möglich, nimmt der Compiler bereits Optimierungen vor. So kann z. B. eine Verkettung von zwei Strings, die wie in diesem Beispiel direkt als Zeichenkette eingegeben wurden, bereits während des Kompilierens erfolgen.

Strings vergleichen

Es gibt verschiedene Möglichkeiten, die Gleichheit von Strings zu testen. Dabei spielt es eine große Rolle, wie Java Strings intern verwaltet.

- ✓ Strings, deren Inhalt beim Kompilieren bereits feststeht, z. B. `String s = "Text"`, werden nur einmal angelegt (ein String-Pool wird erzeugt). Existieren während des Kompilierens weitere Strings mit dem gleichen Inhalt, verweisen alle auf die gleiche interne Zeichenkette `"Text"`, d. h. auf das gleiche String-Objekt.
- ✓ Für Strings, die dynamisch, d. h. erst zur Laufzeit eines Programms, gebildet werden, erzeugt Java immer ein neues separates String-Objekt.

Strings inhaltlich vergleichen

Für den inhaltlichen Vergleich von Strings können Sie die Methoden `equals` und `equalsIgnoreCase` der Klasse `String` verwenden.

- ✓ Die Methode `equals` ① vergleicht die Inhalte von String-Objekten, d. h. die Zeichenketten selbst, und liefert als Ergebnis entsprechend `true` oder `false`.

```
boolean equals(Object value) ①
boolean equalsIgnoreCase(String value) ②
```

Die Ausgabe des Programms

- ✓ Die Methode `equalsIgnoreCase` ② überprüft den String auf inhaltliche Gleichheit mit einem anderen String, ignoriert dabei aber die Groß-/Kleinschreibung.

Strings lexikografisch vergleichen

Über eine weitere Methode der Klasse `String`, die Methode `compareTo`, haben Sie zusätzlich die Möglichkeit, Strings lexikografisch zu vergleichen. Rückgabewerte sind 0 (bei Gleichheit) oder entsprechend der lexikografischen Ordnung kleiner bzw. größer als 0.

Teilmehalte von Strings vergleichen

Mit der Methode `contains` lässt sich überprüfen, ob eine Zeichenkette in einer anderen Zeichenkette enthalten ist.

Beispiel zum inhaltlichen Vergleich von Strings: *com\herdt\java9\kap11\StringCompare.java*

Das folgende Beispiel zeigt die möglichen Vergleiche von Zeichenketten und gibt deren Ergebnisse auf der Konsole aus.

```
package com.herdt.java9.kap11;
import static java.lang.System.out;
class StringCompare
{
    public static void main(String[] args)
    {
        ① String s1 = "text";
        ② String s2 = new String("Text");
        ③ String s3 = new String("Context");
        out.println("Gross- und Kleinschreibung berücksichtigen:");
        ④ out.printf
            ("%s und %s: %b%n", s1, s2, (s1.equals(s2))); //false
        out.println
            ("\nGross- und Kleinschreibung nicht berücksichtigen:");
        ⑤ out.printf
            ("%s und %s: %b%n", s1, s2, (s1.equalsIgnoreCase(s2))); //true
        out.println("\nLexikografischer Vergleich:");
        ⑥ out.printf
            ("%s und %s: %d%n", s1, s2, (s1.compareTo(s2))); //32
        out.println
            ("\nPruefen ob ein String in einem anderen enthalten ist:");
        ⑦ out.printf
            ("%s in %s: %b%n", s1, s3, (s3.contains(s1))); //true
        ⑧ out.printf
            ("%s in %s: %b%n", s2, s3, (s3.contains(s2))); //false
    }
}
```

- ① Die Variable `s1` wird mit dem Literal `text` initialisiert.
- ③ Der Variablen `s2` wird dynamisch (erst zur Laufzeit) die Zeichenkette `Text` zugewiesen.
- ③ Der Variablen `s3` wird dynamisch die Zeichenkette `Context` zugewiesen.
- ④ Mit `equals` werden die Zeichenketten selbst verglichen. Da die Groß- und Kleinschreibung berücksichtigt wird, wird `false` zurückgegeben.
- ⑤ Der Vergleich mit `equalsIgnoreCase` von `s1` und `s2` liefert `true`. Die Groß- und Kleinschreibung wird nicht berücksichtigt.
- ⑥ Die `s1` ist lexikografisch größer als `s2`. `compareTo` liefert ein positives Ergebnis (32).
- ⑦ `s1` ist in `s3` nicht enthalten, denn die Groß- und Kleinschreibung wird berücksichtigt. `contains` liefert das Ergebnis `true`.
- ⑧ `s2` ist in `s3` enthalten. `contains` liefert das Ergebnis `false`.

Referenzvariablen aus String-Objekten vergleichen

Der Vergleich von Referenzvariablen aus String-Objekten mit gleichem Inhalt mit dem Operator `==` kann aufgrund des String-Pools unterschiedliche Ergebnisse liefern:

- ✓ Bei dem Vergleich von zwei identischen String-**Konstanten** (① und ②) liefert `==` das Ergebnis `true` (④), da es sich um Referenzen auf dasselbe String-Objekt im String-Pool handelt.
- ✓ Werden Strings im Programm dynamisch mit identischem Inhalt erzeugt (③), liefert der Vergleich der Referenzvariablen den Wert `false` (⑤), da es sich um zwei verschiedene String-Objekte handelt.

	...
①	<code>String s1 = "Text";</code>
②	<code>String s2 = "Text";</code>
③	<code>String s3 = new String("Text");</code>
④	<code>System.out.println("s1 == s2 : " + (s1 == s2)); // liefert true</code>
⑤	<code>System.out.println("s2 == s3 : " + (s2 == s3)); // liefert false</code>

11.3 Weitere Methoden der Klasse `String`

Die Klasse `String` bietet einige nützliche Methoden, die im Folgenden erläutert werden.

Methode	Beschreibung	Beispiel
<code>char charAt(int index)</code>	Liefert das Zeichen an der Position <code>index</code> (0..n-1)	<code>char c = "abc".charAt(1);</code> <code>// c hat den Inhalt b</code>
<code>boolean endsWith(String value)</code>	Überprüft, ob der String mit <code>value</code> endet; liefert <code>true</code> , wenn <code>value</code> gleich "" oder gleich dem String ist	<code>String s = "Text";</code> <code>s.endsWith("Text") //true</code> <code>s.endsWith("xt") //true</code>

Methode	Beschreibung	Beispiel
<code>int indexOf (String value)</code>	Liefert den Index innerhalb eines Strings, an dem der übergebene Teil-String <code>value</code> beginnt. Es gibt noch weitere Varianten von <code>indexOf</code> .	<pre>String s = "Programmcode"; int i = s.indexOf("code"); // i = 8</pre>
<code>boolean isEmpty()</code>	Diese Methode gibt es seit der Version Java 6. Sie liefert als Ergebnis <code>true</code> , wenn der String leer ist. (Die Methode <code>length()</code> liefert in diesem Fall das Ergebnis 0.)	<pre>String s1 = "Programmcode"; String s2 = ""; boolean b1 = s1.isEmpty(); boolean b2 = s2.isEmpty(); // b1 = false // b2 = true</pre>
<code>String join (CharSequence delimiter, String)</code>	Die Methode gibt es seit Java 8. Sie erstellt einen String, der sich aus den übergebenen Zeichenketten, getrennt durch das bzw. die Trennzeichen (engl. <code>delimiter</code>), zusammensetzt.	<pre>String s1 = String.join("-", "String", "mit", "Trennzeichen"); // s1 = "String--mit--Trennzeichen"</pre>
<code>int length()</code>	Liefert die Länge der Zeichenkette	<pre>String s = "Programmcode"; int len = s.length(); // len = 12</pre>
<code>String replace (char oldC, char newC)</code>	Ersetzt alle vorkommenden Zeichen <code>oldC</code> durch das Zeichen <code>newC</code>	<pre>String s = "Text"; s = s.replace('x', 's'); // s = "Test";</pre>
<code>String substring(int s) String substring (int s1, int s2)</code>	Liefert einen Teil-String beginnend beim Index <code>s</code> bis zum Ende bzw. von <code>s1</code> bis (<code>s2-1</code>)	<pre>String s = "Programmcode"; s = s.substring(2, 4); // s = "og"</pre>
<code>String toLowerCase() String toUpperCase()</code>	Diese Methoden wandeln die Zeichen eines Strings in Klein- bzw. Großschreibung um. Prüfen Sie gegebenenfalls, wie spezielle Buchstaben, wie beispielsweise das Zeichen ß konvertiert werden.	<pre>String s = "Text"; s = s.toUpperCase(); // s = "TEXT"</pre>
<code>String trim()</code>	Erstellt eine Kopie des Strings und entfernt darin am Beginn und Ende des Strings alle Zeichen, die im Zeichensatz vor dem Leerzeichen (' <code>\u0020</code> ') stehen, und das Leerzeichen selbst	<pre>String s = " Text "; String s1 = s.trim(); // s = " Text " // s1 = "Text"</pre>
<code>String valueOf (type var)</code>	Liefert die String-Darstellung (representation) des Datentyps bzw. des Objekts	<pre>int i = 10; String s; s = String.valueOf(i); // s = "10"</pre>

11.4 Die Klassen `StringBuffer` und `StringBuilder`

Veränderliche Zeichenketten

Die Klasse `String` bietet keine Möglichkeit, einen `String` nachträglich zu ändern. Ersetzen Sie z. B. Zeichen eines `String`s, wird immer ein neues `String`-Objekt erzeugt und der entsprechenden Variablen zugewiesen.

Die Klasse `StringBuffer` stellt einen Datentyp bereit, bei dem sich die Größe dynamisch der gespeicherten Zeichenkette anpasst.

Die Klasse `StringBuffer` hat folgende Eigenschaften:

- ✓ Die Klasse `StringBuffer` kann veränderliche `Strings` verwalten und bearbeiten.
- ✓ Aufgrund der dynamischen Verwaltung der Zeichenkette sind Operationen, wie z. B. das Verketteten von `Strings`, mit einem `StringBuffer` in der Ausführung schneller als mit Zeichenketten vom Typ `String`.
- ✓ Die Größe des Puffers (Buffer) wird automatisch an die Größe des `Strings` dynamisch angepasst, sie ist aber größer als die gespeicherte Länge des `Strings`. Die Vergrößerung des Puffers wird jeweils durch Verdopplung der momentanen Speichergröße erreicht.

Eine veränderliche Zeichenkette mit `StringBuffer` erzeugen

- ✓ Um einen `StringBuffer` mit der vorgegebenen Kapazität von 16 Zeichen zu erzeugen, verwenden Sie den Konstruktor ohne Parameter ①.
- ✓ Durch die Angabe einer Länge als Parameter ② legen Sie die Initialkapazität (Kapazität bei der Erzeugung des Objekts) fest.
- ✓ Die Übergabe eines `Strings` als Parameter ③ bewirkt, dass eine Kopie des `Strings` im `StringBuffer` erzeugt wird. Die Kapazität wird der Größe des `Strings` angepasst.

```
StringBuffer(); ①  
StringBuffer(int length); ②  
StringBuffer(String str); ③
```

Die Ausgabe des Programms

Ein `StringBuffer`-Objekt in einen `String` umwandeln

- ✓ Verwenden Sie die Methode `toString` der Klasse `StringBuffer`.

```
StringBuffer sb = new StringBuffer("Text");  
String s = StringBuffer.toString();
```

oder

- ✓ Verwenden Sie einen speziellen Konstruktor von `String`, der einen Parameter vom Typ `StringBuffer` erlaubt.

```
StringBuffer sb = new StringBuffer("Text");  
String s = new String(sb);
```

Methoden der Klasse `StringBuffer`

Methode	Beschreibung	Beispiel
<code>StringBuffer append (type value)</code>	Es gibt verschiedene Varianten von <code>append</code> , die unterschiedliche Parametertypen verwenden (<code>int</code> , <code>double</code> , <code>Object</code> ,...). Diese Typen werden in einen <code>String</code> umgewandelt und an den Puffer angehängt.	<pre>StringBuffer s; s = new StringBuffer(); s.append("1 + 2 = "); s.append(1 + 2); // s = "1 + 2 = 3"</pre>
<code>int capacity()</code>	Liefert die aktuelle Kapazität des Puffers	<pre>int i = s.capacity(); //16</pre>
<code>char charAt(int index)</code> <code>void setCharAt (int index, char c)</code>	Liefert das Zeichen mit dem betreffenden Index bzw. ändert es	<pre>char c = s.charAt(2); // c = '+'</pre>
<code>StringBuffer insert (int index, type value)</code>	Fügt ab der Position <code>index</code> die <code>String</code> -Repräsentation des Parameters <code>value</code> ein	<pre>s.insert(0, "("); s.insert(6, ")"); // s = "(1 + 2) = 3"</pre>
<code>int length()</code>	Liefert die Länge des gespeicherten Strings	<pre>int i = s.length(); // 11</pre>
<code>String toString()</code>	Liefert die <code>String</code> -Repräsentation des <code>StringBuffer</code>	<pre>String str = s.toString();</pre>

Die Klasse `StringBuilder` verwenden

Diese Klasse entspricht in der Funktionsweise der Klasse `StringBuffer`. `StringBuilder` sind jedoch **nicht** synchronisiert. Das bedeutet, dass sie **nicht** für den Zugriff von mehreren Threads (gleichzeitig ausgeführten Programmmodulen) geeignet sind. Da gleichzeitig ablaufende Programmmodule nicht überwacht werden, ergeben sich in vielen Anwendungsfällen durch den Einsatz der Klasse `StringBuilder` nochmals Steigerungen in der Ausführungsgeschwindigkeit.

11.5 Wrapper-Klassen

Was sind Wrapper-Klassen?

Viele Methoden innerhalb von Java erwarten als Parameter einen `Object`-Typ. Da Sie von primitiven Datentypen keine Objekte bilden können, stellt Java für jeden primitiven Datentyp eine sogenannte **Wrapper-Klasse** bereit. Die Klassen besitzen ein gekapseltes Attribut, das den Wert

speichert, und Methoden, um beispielsweise den entsprechenden primitiven Datentyp zurückzugeben oder eine Konvertierung von `String`-Typen in den betreffenden Datentyp vorzunehmen.

Folgende Wrapper-Klassen existieren in Java und sind im Package `java.lang` enthalten:

- | | | | |
|----------------------|------------------------|-----------------------|--------------------------|
| ✓ <code>Void</code> | ✓ <code>Integer</code> | ✓ <code>Float</code> | ✓ <code>Boolean</code> |
| ✓ <code>Byte</code> | ✓ <code>Long</code> | ✓ <code>Double</code> | ✓ <code>Character</code> |
| ✓ <code>Short</code> | | | |

Wrapper-Klassen verwenden

Am Beispiel der Wrapper-Klasse `Integer` werden im Folgenden einige Methoden der Klassen vorgestellt. Für die Wrapper-Klassen der anderen primitiven Datentypen existieren ähnliche Methoden. Informationen dazu erhalten Sie über die Java-Dokumentation.

Der Aufruf der statischen Methoden `valueOf(int value)` bzw. `valueOf(String value)` erzeugt ein `Integer`-Objekt mit dem Wert von `value` und liefert dieses zurück.

Die bis Version Java 8 verfügbaren Konstruktoren in der Form `Integer(int value)` und `Integer(String s)` sind in Java 9 deprecated. Ihre Verwendung erzeugt bei der Kompilierung einen Fehler.

Für die Wrapper-Klasse existieren Methoden, mit denen Sie beispielsweise Werte auslesen oder in Strings umwandeln können:

<code>int intValue()</code>	Diese Methode liefert den gespeicherten <code>int</code> -Wert zurück.	
<code>int parseInt(String s)</code>	Versucht, den String <code>s</code> in einen <code>int</code> -Wert zu konvertieren. Falls eine Konvertierung nicht möglich ist, wird eine <code>NumberFormatException</code> ausgelöst.	*
<code>String toString()</code>	Liefert einen String zurück, der den gespeicherten Wert repräsentiert	
<code>String toString(int value)</code>	Liefert einen String zurück, der den Wert von <code>value</code> repräsentiert	*
<code>Integer valueOf(String s)</code>	Versucht, den String <code>s</code> in einen <code>int</code> -Wert zu konvertieren, erzeugt mit diesem Wert ein <code>Integer</code> -Objekt und liefert dieses zurück. Falls eine Konvertierung nicht möglich ist, wird eine <code>NumberFormatException</code> ausgelöst.	*
<code>int signum(int i)</code>	Die Methode wendet die Signum-Funktion auf den <code>int</code> -Wert <code>value</code> an. Ist <code>value</code> eine negative Zahl, liefert die Methode den Wert <code>-1</code> , bei der Zahl <code>0</code> den Wert <code>0</code> und bei positiven Zahlen den Wert <code>1</code> .	*

Die mit * gekennzeichneten Methoden sind statisch, sodass Sie diese auch ohne ein konkretes Objekt der betreffenden Wrapper-Klasse nutzen können.

Während Sie Variablen von primitiven Datentypen wie beispielsweise `int` mit dem Operator `==` vergleichen, beachten Sie, dass Sie zum Vergleich zweier Wrapper-Objekte die Methode `equals` verwenden, um auf inhaltliche Gleichheit zu prüfen.

Beispiel: `com\herdt\java9\kap11\Wrapper.java`

```
//neues Integer-Objekt mit dem Wert 24 erzeugen:
Integer i1 = Integer.valueOf("24");
//gespeicherten int-Wert von i1 liefern:
int i = i1.intValue();
//Stringkonvertierung nach int
int j = Integer.parseInt("35");
//int-Wert von i1 in einen String umwandeln:
String s1 = i1.toString();
//Umwandlung einer int-Zahl in einen String
String s2 = Integer.toString(74);
```

Die Wrapper-Klassen besitzen keine Methoden zum Ändern des gespeicherten Wertes (Setter-Methoden). Sie können durch die Verwendung von Wrapper-Klassen keine Parameter an Methoden übergeben, die veränderlich sind.

Autoboxing


Die Umwandlung von einem Objekt einer Wrapper-Klasse in den entsprechenden primitiven Datentyp können Sie mit der entsprechenden Methode wie z. B. `intValue` erreichen. Um ein Objekt zu erstellen, verwenden Sie den Konstruktor der Wrapper-Klasse. Um diese aufwendige Schreibweise zu vereinfachen, ist seit der Version Java 5 eine automatische Umwandlung implementiert, das sogenannte **Autoboxing**.

- ✓ Die Konvertierung in ein Objekt der entsprechenden Wrapper-Klasse wird **Boxing** genannt.
- ✓ Die Rückumwandlung in einen primitiven Datentyp heißt entsprechend **Unboxing**.

	Ausführliche Schreibweise	Vereinfachte Schreibweise mit Autoboxing
	<code>int i = 10;</code>	<code>int i = 10;</code>
Boxing	<code>Integer j = Integer.valueOf(i);</code>	<code>Integer j = i;</code>
Unboxing	<code>int k = j.intValue();</code>	<code>int k = j;</code>

11.6 Übung

Zeichenketten manipulieren

Level		Zeit	ca. 25 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Sortierung von Zeichenketten ✓ Bearbeitung von Zeichenketten 		
Übungsdatei	--		
Ergebnisdateien	<i>Exercise1.java, Exercise2.java, Exercise3.java, Exercise4.java</i>		

1. Sortieren Sie die Namen *Meier* und *Mayer* über die Methode `compareTo` der Klasse `String` und geben Sie diese alphabetisch sortiert aus.
2. Verändern Sie die Übung ① so, dass die Namen vor der Ausgabe in Kleinbuchstaben umgewandelt werden.
3. Schreiben Sie eine Methode `printLetter`, die einen Namen als Parameter entgegennimmt. In der Methode soll über die Klasse `StringBuffer` der folgende String zusammengesetzt und ausgegeben werden:
 Sehr geehrte(r) Frau/Herr <hier soll der als Parameter übergebene Name stehen>,
 wir gratulieren Ihnen zur erfolgreichen Lösung der Übungsaufgabe.
4. Schreiben Sie eine zweite Methode `printLetter1`, die ebenfalls einen Namen als Parameter entgegennimmt. In der Methode soll über die Klasse `StringBuilder` zunächst der folgende String zusammengesetzt werden:
 Sehr geehrte(r) Frau/Herr,
 wir gratulieren Ihnen zur erfolgreichen Lösung der Übungsaufgabe.
 Anschließend soll der als Parameter übergebene Name an der entsprechenden Position in den Text eingefügt werden. Dann soll der Text ausgegeben werden.

12

Arrays und Enums

12.1 Arrays

Was sind Arrays?

Über **Arrays** (Felder) können Sie mehrere Daten eines Typs in einer einzigen Struktur speichern. Über deren **Index** besitzen Sie einen wahlfreien Zugriff auf die einzelnen Daten, die sogenannten **Array-Elemente (Feldelemente)**. Der Index ist ein ganzzahliger Wert, der jeweils zu einem Feldelement gehört. Das erste Feldelement besitzt den Index **0**, für jedes weitere Element erhöht sich der Index jeweils um 1.

- ✓ In Java sind Arrays Referenztypen.
- ✓ Array-Objekte werden daher über `new` erzeugt.
- ✓ Eine Array-Variable ist eine Referenzvariable.
- ✓ Bei der Erzeugung eines Array-Objekts wird die Größe, d. h. die Anzahl der Elemente, vereinbart.
- ✓ Die Größe kann nachträglich nicht mehr geändert werden.

Feldelement 4	0	122
	1	56
	2	145
	3	157
	4	87
	5	57
	6	145
	7	155
	8	68
	9	23

Ein Array mit zehn Elementen vom Typ `int`

Syntax für die Definition, Erzeugung und Initialisierung von Arrays

Ein Array definieren

- ✓ Sie definieren eine Array-Variable, indem Sie dem Datentyp ① ein Klammerpaar `[]` anfügen.
- ✓ Es ist auch möglich, die Klammern hinter dem Variablennamen zu platzieren ②. Die Schreibweise ① wird üblicherweise verwendet: Sie beschreibt anschaulich, dass es sich bei der Variablen um eine Referenz auf ein Feld (Array) handelt.

```
type[] identifier; ①
type identifier[]; ②
```

Ein Array erzeugen

- ✓ Um ein Array zu erzeugen, verwenden Sie den Operator `new` ③.
Die Array-Größe (die Anzahl der Feldelemente) wird dazu in eckigen Klammern hinter dem Datentyp angegeben. Sie können die Array-Größe später nicht mehr ändern.
- ✓ Die Feldgröße muss vom Datentyp `int` sein. Sie können sie mit einer Zahl (Literal), mit einer Konstanten oder einem Ausdruck angeben. Der Ausdruck wird zur Laufzeit ausgewertet, und somit wird auch die Größe erst zur Laufzeit festgelegt.
- ✓ Bei der Erzeugung des Arrays werden die Feldelemente automatisch mit default-Werten für den jeweiligen Datentyp initialisiert. So erhalten beispielsweise Feldelemente vom Typ `int` den Wert 0.
- ✓ Sie können die Definition und das Erzeugen des Arrays in einer Anweisung durchführen ④.

```
identifizier = new type[size];③
type[] identifizier1 = new type[size1];④
```

Array-Literale: Feldelemente individuell initialisieren

- ✓ Bereits bei der Erzeugung des Arrays können Sie die Feldelemente mit Daten initialisieren.
- ✓ Geben Sie auf der rechten Seite des Gleichheitszeichens in geschweiften Klammern `{ }` die gewünschten Werte als Literale ein. Trennen Sie dabei die einzelnen Werte durch Kommata.
- ✓ Durch die Anzahl der Werte, die Sie innerhalb der geschweiften Klammern angeben, wird gleichzeitig die Größe des Arrays festgelegt.
- ✓ Der Operator `new` wird nicht benötigt.

```
type[] identifizier1 = {value0, value1, ...valueN};
```

Beispiele für die Definition, Erzeugung und Initialisierung von Arrays

```
① double[] field1;
   field1 = new double[15];    //Array mit 15 Elementen vom Typ double
② int[] field2 = new int[20]; //Array mit 20 Elementen vom Typ int
③ double[] field3 = {1.2, 42.3, 27.0, 12.567};
   //Feldelemente mit Literalen initialisieren
④ int a = 1;
   int b = 6;
   int c = 4;
   int[] field4 = {a, b, c, a + b + c}; //Ausdrücke verwenden
```

- ① Definition und Erzeugung eines Arrays in zwei separaten Anweisungen
- ② Definition und Erzeugung eines Arrays in einer einzigen Anweisung
- ③-④ Definition, Erzeugung und Initialisierung eines Arrays mit individuellen Werten

Über den Index auf Array-Elemente zugreifen

Wie auch beispielsweise Variablen primitiver Datentypen können Sie Array-Elementen Werte zuweisen und die Inhalte auslesen.

Syntax für Wertzuweisungen und das Auslesen von Array-Elementen

- ✓ Der Zugriff auf Feld-elemente erfolgt durch den Index, der bei 0 beginnt und entsprechend für das letzte Feldelement den Wert Feldlänge -1 besitzt.
- ✓ Der Index kann ein beliebiger Ausdruck sein, der ein Ergebnis vom Typ `int` liefert.
- ✓ Sie können einem über den Index bestimmten Array-Element einen Wert zuweisen ①
- ✓ Über den Index (`intExpression1`) können Sie den Wert des entsprechenden Array-Elements auslesen ② und beispielsweise in einer anderen Variablen speichern.

```
identifier[intExpression] = expression; ①
type identifier1 = identifier[intExpression1]; ②
```

Fehlerhafte Indexangaben

Der Wertebereich für den Index erstreckt sich von 0 für das erste Element bis zu Feldlänge -1 für das letzte Element. Wenn Sie einen ungültigen Index angeben, wird dieser Fehler vom Interpreter erkannt und es wird eine sogenannte `Exception ArrayIndexOutOfBoundsException` erzeugt. Ein entsprechender Hinweis wird auf der Konsole ausgegeben.

Die Anzahl der Array-Elemente ermitteln

Jedes Array besitzt ein Attribut `length`, mit dem Sie die Größe des Arrays, d. h. die **Anzahl der Array-Elemente**, bestimmen können.

```
int identifier = arrayIdentifier.length;
```

Die Ausgabe des Programms

Beispiele für die Arbeit mit Arrays: `com\herdt\java9\kap12\UseArray.java`

```
package com.herdt.java9.kap12;
class UseArray
{
    public static void main(String[] args)
    {
        int a = 2;
        int b = 6;
        ① int[] field = {3, 4, 5, a + b};
        ② field[2] = 7;
        ③ System.out.printf("Anzahl Elemente: %d %n", field.length);
    }
}
```

```

④ for (int i = 0; i < field.length; i++)
    System.out.printf("Element %d: %d%n", i, field[i]);
    }
}

```

- ① Definition und Erzeugung und Initialisierung eines Arrays mit individuellen Werten
- ② Wertzuweisung an das dritte Feldelement
- ③ Die Anzahl der Feldelemente wird dem Attribut `field.length` entnommen und ausgegeben.
- ④ Für jedes Array-Element werden der Index und der Wert ausgegeben.

```

Anzahl Elemente: 4
Element 0: 3
Element 1: 4
Element 2: 7
Element 3: 8

```

Die Ausgabe des Programms

Arrays von Referenztypen

Arrays können Sie von Daten eines beliebigen Typs erzeugen. So können Sie nicht nur Arrays von primitiven Datentypen, sondern beispielsweise auch für selbst erstellte Datentypen erzeugen. Wie bei Referenzvariablen werden dann in den Feldelementen Referenzen auf die entsprechenden Objekte gespeichert.

12.2 Mit Arrays arbeiten

Zuweisungen bei Array-Variablen

Array-Variablen sind Referenzvariablen. Daher können Sie ein Array nicht kopieren, indem Sie einer Array-Variablen eine andere Array-Variable zuweisen. Bei dieser Zuweisung wird lediglich die Referenz kopiert. Beide Array-Variablen verweisen anschließend auf dasselbe Array.

Ausgangszustand	Zuweisung	Ergebnis
<pre> int[] field1 = {23, 74}; int[] field2 = {17, 65}; </pre>	<pre> field2 = field1; </pre>	<p>Beide Array-Variablen verweisen auf dasselbe Array.</p>

Arrays kopieren

Um ein Array zu kopieren, können Sie eine `for`-Schleife nutzen oder Sie verwenden die Methode `arraycopy` der Klasse `System`.

```
System.arraycopy(field1, indexOfField1, field2, indexOfField2,
    numberOfElements);
```

- ✓ Die Methode `arraycopy` benötigt fünf Parameter.
- ✓ Als ersten Parameter (`field1`) geben Sie das Array an, dessen Inhalte Sie kopieren möchten.
- ✓ Als zweiten Parameter (`indexOfField1`) geben Sie den Index des ersten Elements im `field1` an, ab dem kopiert werden soll.
- ✓ Als dritten Parameter (`field2`) geben Sie das Array an, in das Sie kopieren möchten.
- ✓ Der vierte Parameter (`indexOfField2`) beschreibt die Position im Ziel-Array, an die kopiert werden soll.
- ✓ Als letzten Parameter geben Sie an, wie viele Elemente kopiert werden sollen.

Beachten Sie, dass kein Indexwert außerhalb des gültigen Bereichs entsteht. Dies kann durch die Parameter `indexOfField1` und `indexOfField2`, aber auch durch den Parameter `numberOfElements` eintreten. Wird der gültige Bereich überschritten, wird eine sogenannte `ArrayIndexOutOfBoundsException` ausgelöst.

Es existieren in der Klasse `java.util.Arrays` weitere Methoden zum Kopieren von Arrays, wie z. B. die Methoden `copyOf` und `copyOfRange`.

```
field2 =
    java.util.Arrays.copyOf(orgField, numberOfElements);
field2 =
    java.util.Arrays.copyOfRange(orgField, FromElement, ToElement);
```

- ✓ Mit der Methode `copyOf` kopieren Sie die ersten Elemente eines Arrays `orgField` in ein neues Array `field2`. `numberOfElements` legt die Anzahl der Elemente des kopierten Arrays fest. Verfügt das zu kopierende Array über mehr Elemente, werden diese nicht kopiert. Liegt die angegebene Anzahl über der Anzahl der Elemente des zu kopierenden Arrays, werden diese Elemente im kopierten Array mit einem Standardwert vorbelegt, z. B. mit 0 beim Datentyp `int`.
- ✓ Mit der Methode `copyOfRange` kopieren Sie ebenfalls einen Teil eines Arrays `orgField` in ein neues Array `field2`. Mit dem Argument `FromElement` geben Sie den Index des ersten zu kopierenden Elements an. Den Index des letzten zu kopierenden Elements legen Sie mit dem Argument `ToElement - 1` fest.

Beispiele für das Kopieren eines Arrays: `com\herdt\java9\kap12\CopyArray.java`

```
package com.herdt.java9.kap12;
class CopyArray
{
    public static void main(String[] args)
    {
        ① int[] field = new int[10];
        for (int i = 0; i < field.length; i++)
        {
            field[i] = 2 * i;
            System.out.printf("Element %d: %d%n",i, field[i]);
        }

        System.out.printf("Die ersten fuenf Elemente:%n");
        ② int[] firstFive = java.util.Arrays.copyOf(field, 5);
        ③ for (int i = 0; i < firstFive.length; i++)
            System.out.printf("Element %d: %d%n",i, firstFive[i]);

        System.out.printf("Element 3 bis Element 7:%n");
        ④ int[] threeToSeven = java.util.Arrays.copyOfRange(field,
            3, 7);
        ⑤ for (int i = 0; i < threeToSeven.length; i++)
            System.out.printf("Element %d: %d%n",i,
            threeToSeven[i]);
    }
}
```

- ① Definition, Erzeugung und Initialisierung eines Arrays mit zehn Elementen und Ausgabe der Elemente.
- ② Die ersten fünf Elemente werden mit der Methode `copyOf` in ein neues Feld kopiert, das der Variablen `firstFive` zugewiesen wird.
- ③ Das neue Array mit den fünf Elementen wird ausgegeben.
- ④ Mit der Methode `copyOfRange` wird ein Teilbereich des Arrays kopiert. Beginnend mit dem vierten Element (Index 3) werden die Elemente bis einschließlich des sechsten Elements (Index 7) in ein neues Feld kopiert, das der Variablen `threeToSeven` zugewiesen wird.
- ⑤ Für jedes Array-Element werden der Index und der Wert ausgegeben.

```
Element 0: 0
Element 1: 2
Element 2: 4
...
Element 8: 16
Element 9: 18
Die ersten fuenf Elemente:
Element 0: 0
Element 1: 2
Element 2: 4
Element 3: 6
Element 4: 8
Element 3 bis Element 7:
Element 0: 6
Element 1: 8
Element 2: 10
Element 3: 12
```

Die Ausgabe des Programms

Arrays in Schleifen bearbeiten

Häufig werden Arrays innerhalb von Schleifen bearbeitet. Beispielsweise sollen alle Elemente eines Arrays auf einen bestimmten Wert gesetzt werden ① oder Sie möchten alle Werte eines Arrays ausgeben ②.

Die Schreibweise für das Auslesen von Arrays in Schleifen können Sie abkürzen ③, wenn ...

```
...
int[] square = new int[10];
for (int i = 0; i < square.length; i++) ①
    square[i] = i * i;

for (int i = 0; i < square.length; i++) ②
    System.out.println(square[i]);

for (int sq : square) ③
    System.out.println(sq);
...
```

Beispiel: „com\herdt\java9\kap12\ArrayLoops.java“

- ✓ Array-Elemente nur ausgelesen und nicht verändert werden,
- ✓ beginnend beim ersten Element alle Elemente nacheinander durchlaufen werden sollen,
- ✓ nur **ein** Array durchlaufen wird.

Diese Schreibweise der Schleife wird auch als **foreach-Schleife** bezeichnet. Im Schleifenkopf definieren Sie innerhalb der Klammern eine Variable vom Typ eines Array-Elements (hier: `int`). Dann geben Sie einen Doppelpunkt `:` und die Array-Variable an.

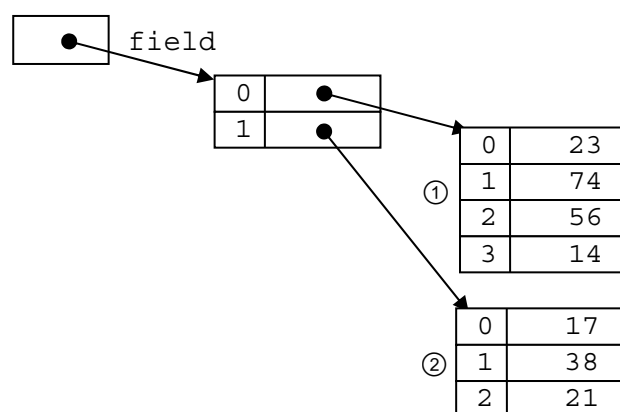
In jedem Schleifendurchlauf wird der neu definierten Variablen (hier `sq`) der Wert des entsprechenden Array-Elements zugewiesen. Sie können diese Variable direkt verwenden und beispielsweise in einer `println`-Anweisung ausgeben.

12.3 Mehrdimensionale Arrays

Der Aufbau mehrdimensionaler Arrays in Java

Bei mehrdimensionalen Arrays handelt es sich um geschachtelte Arrays. Da zu jedem Datentyp ein Array gebildet werden kann, können Sie auch ein Array erzeugen, dessen Array-Elemente wiederum Referenzen auf Arrays enthalten. Sie erzeugen somit ein Array von Arrays. Bei den zuvor beschriebenen eindimensionalen Arrays geben Sie den Index in eckigen Klammern an.

Bei mehrdimensionalen, geschachtelten Arrays geben Sie für jede Schachtelungsebene ein Klammerpaar `[]` an.



Mehrdimensionales Array (zweidimensional)

- ✓ Jedes dieser Arrays kann eine unterschiedliche Größe haben (① und ②).
- ✓ Alle geschachtelten Arrays müssen vom selben Datentyp sein.

Beispiel für ein zweidimensionales Array: *com\herdt\java9\kap12\TwoDimensional.java*

Mit dem nebenstehenden Quelltext erzeugen Sie das oben abgebildete zweidimensionale Array.

- ① Sie erzeugen ein zweidimensionales Array `field` vom Typ `int`. Das Array soll aus zwei Elementen bestehen, die jedoch selbst keine `int`-Werte speichern, sondern wiederum Referenzen auf Arrays vom Typ `int` enthalten.
- ② Das erste dieser Arrays soll aus 4 Elementen vom Typ `int` bestehen.
- ③ Die Werte der Elemente werden festgelegt.
- ④ Das zweite geschachtelte Array besteht aus 3 Elementen vom Typ `int`.
- ⑤ Die Werte der Elemente des zweiten Arrays werden festgelegt.

```

① int [] [] field = new
② int [2] [] ;
③ field[0] = new int [4] ;
   field[0][0] = 23;
   field[0][1] = 74;
   field[0][2] = 56;
   field[0][3] = 14;
④ field[1] = new int [3] ;
⑤ field[1][0] = 17;
   field[1][1] = 38;
   field[1][2] = 21;

```

Schleifen für mehrdimensionale Arrays

Um mehrdimensionale Arrays zu bearbeiten, müssen Sie entsprechend der Schachtelung der Arrays auch die Schleifen schachteln ①/②.

```

① for (int i = 0; i < field.length; i++)
{
    ② for (int j = 0; j < field[i].length; j++)
        System.out.println(field[i][j]);
}

```

Die innere, geschachtelte Schleife ② lässt sich hier auch in der verkürzten Form schreiben:

```

for (int element : field[i])
    System.out.println(element);

```

Arrays gleicher Größe schachteln

Sofern alle Arrays, die in einem Array geschachtelt sind, die gleiche Größe besitzen, können Sie die Erzeugung dieser Arrays in einer Anweisung durchführen.

```
int [] [] field = new int [4] [5];
```

Zweidimensionale Arrays dieser Art werden häufig für die Speicherung von tabellarischen Daten genutzt. Das äußere Array enthält beispielsweise die Zeilen der Tabelle. Die zweite Ebene enthält als Array die Zellen der jeweiligen Zeile.

	0	1	2	3	4
0					
1					
2					
3					

12.4 Spezielle Methoden zur Arbeit mit Arrays

In der Klasse `java.util.Arrays` befinden sich einige nützliche Methoden zum Sortieren und Füllen numerischer Felder. Die nachfolgende Tabelle zeigt eine Auswahl der Methoden. Die Methoden stehen als überladene Methoden jeweils für Arrays verschiedener Datentypen zur Verfügung.

<code>java.util.Arrays.sort(field)</code>	Elemente des Arrays entsprechend den Inhalten aufsteigend sortieren
<code>java.util.Arrays.equals(field1, field2)</code>	Die Arrays <code>field1</code> und <code>field2</code> auf gleiche Elementinhalte vergleichen
<code>java.util.Arrays.fill(field1, value)</code>	Alle Elemente des Array <code>field</code> mit dem Wert <code>value</code> füllen
<code>java.util.Arrays.toString(field)</code>	Liefert einen String, der die Werte der Elemente des Arrays <code>field</code> als String repräsentiert

Beispiel: `com\herdt\java9\kap12\SortArray.java`

	<pre>public static void main(String[] args) { ① int[] someNumbers = {11, 23, 4, 17, 6}; ② java.util.Arrays.sort(someNumbers); ③ for (int value : someNumbers) System.out.println(value); ④ String s = java.util.Arrays.toString(someNumbers); ⑤ System.out.println(s); }</pre>
--	--

- ① Ein Array wird erzeugt und mit Werten initialisiert.
- ② Anschließend wird die Array-Variable (die Referenz auf das Array) an die Methode `sort` der Klasse `Array` im Package `java.util` übergeben. Die Methode sortiert die Elemente des Arrays entsprechend den Werten.
- ③ Die Werte der Array-Elemente werden in einer Schleife ausgegeben.
- ④/⑤ Die Werte der Array-Elemente werden in einen String umgewandelt. Der String enthält in eckigen Klammern die Werte der Elemente als Text mit Kommata getrennt ⑥.

```
4
6
11
17
23
[4, 6, 11, 17, 23] ⑥
```

Die Ausgabe des Programms

12.5 Parameterübergabe an die `main`-Methode

Die `main`-Methode innerhalb einer Java-Anwendung besitzt einen Parameter vom Typ `Array` von `Strings`. Alle Zeichenketten, die Sie in dem Befehl zur Ausführung des Java-Interpreters hinter dem Klassennamen anfügen, werden als Parameter der `main`-Methode übergeben. Durch den Aufruf

```
java ParameterInfo Param1 10 12.23
```

werden der Methode `main` der Klasse `ParameterInfo` die Parameterwerte `Param1`, `10` und `10.23` als `Strings` übergeben. Die Zeichenketten müssen durch Leerzeichen getrennt werden.

Beispiel: `com\herdt\java9\kap12\ParameterInfo.java`

Das Programm zeigt alle Parameter, die Sie zusätzlich zum Klassennamen dem Java-Interpreter übergeben, an. Der Aufruf

```
java com.herdt.java9.kap12.ParameterInfo Hallo Java-Programmierer
```

erzeugt beispielsweise die Ausgabe

```
Hallo
```

```
Java-Programmierer
```

- ① Der `main`-Methode werden alle Parameter als Feld von `Strings` übergeben, die Sie zusätzlich zum Klassennamen dem Java-Interpreter anfügen.

- ② Die Anzahl der Parameter können Sie über das Attribut `length` ermitteln. Haben Sie keine Parameter übergeben, wird die `for`-Schleife nicht durchlaufen.

- ③ Die Parameter werden zeilenweise ausgegeben.

- ④ Die zweite `for`-Schleife zeigt die verkürzte `foreach`-Schreibweise.

```
① public static void main(String[] args)
  {
    ②   for (int i = 0; i < args.length; i++)
    ③     System.out.println(args[i]);

    //verkuerzte Schreibweise - foreach
    ④   for (String s : args)
        System.out.println(s);
  }
```

12.6 Methoden mit variabler Anzahl von Parametern

Was sind Varargs?

Durch das Überladen von Methoden haben Sie die Möglichkeit, Methoden mit gleichem Namen, aber mit unterschiedlichen Parametern (Anzahl und Typ) in einer Klasse bereitzustellen. Mit sogenannten **Varargs** (**V**ariable-length **A**rgument list) können Sie **einer** Methode beliebig viele Parameter **eines Typs** übergeben. Varargs stellen eine spezielle Parameterart dar.

- ✓ Innerhalb der Parameterliste einer Methode kann nur ein Vararg-Parameter verwendet werden.
- ✓ Wenn die Parameterliste aus mehreren Parametern besteht, muss der Vararg-Parameter als letzter Parameter aufgeführt werden.

Beispiel für die Verwendung von Varargs:
com\herdt\java9\kap12\UsingVarargs.java

- ① Bei der Methodendefinition beginnen Sie die Angabe des Vararg-Parameters mit dem Datentyp und drei Punkten (. . .). Anschließend folgt der Variablenname (values).
- ②-③ Innerhalb der Methode können Sie den Vararg-Parameter wie ein Array des entsprechenden Datentyps verwenden.

```
① double average(double ... values)
{
    ② double sum = 0.0;
    for (double d : values)
        sum += d;
    ③ return sum / values.length;
}
```

```
① double result1 = average(1.2, 9.3, 4.5, 6.7);
    //Wert von result1: 5.425
② double result2 = average(4.8, 9.2);
    //Wert von result2: 7.0
```

- ①-② Die mit einem Vararg-Parameter definierte Methode können Sie anschließend mit einer beliebigen Parameteranzahl ausführen.

Anonyme Arrays

Anonyme Arrays besitzen keinen Arraynamen. Sie werden beispielsweise verwendet, wenn eine Methode als Parameter ein Array erwartet, aber das Array sonst nicht benötigt wird. Innerhalb der Methode können Sie über den Parameternamen auf das Array zugreifen.

```
// Methode mit Array
// als Parameter aufrufen
somethingToDo
    (new int[] {1, 2, 3});
```

Wie die nachfolgende Anweisung zu dem vorherigen Beispiel zeigt, lassen sich anonyme Arrays als Vararg-Parameter verwenden.

```
double result3 = average(new double[] {4.8, 9.2});
    //Wert von result3: 7.0
```

Auswahl der Methode

Sofern überladene Methoden zur Verfügung stehen, wird anhand der Signatur die entsprechende Methode ausgeführt. Dabei werden die Methoden ohne Vararg-Parameter bevorzugt berücksichtigt.

```
① todo(5);
② todo(3, 9);
③ todo(4, 8, 6);
```

- ① Die Methode ④ wird ausgeführt.
- ② Die Methode ⑤ wird ausgeführt.
- ③ Die Methode ⑥ wird ausgeführt.

```
void todo(int param1) ④
{...}
void todo(int param1, int param2) ⑤
{...}
void todo(int ... param3) ⑥
{...}
```

12.7 Mit Aufzählungstypen (Enumerations) arbeiten

Was sind Wertemengen?

Häufig werden in Programmen sogenannte Wertemengen benötigt, wie beispielsweise:

- ✓ Wochentage
- ✓ Werktage
- ✓ Monate
- ✓ Jahreszeiten

Wertemengen lassen sich als Aufzählungstypen, sogenannte **Enumerations** bzw. **Enums**, definieren.

Syntax für einfache Enumerations

Die Definition beginnt mit dem Schlüsselwort `enum` (von engl. *enumerate* = aufzählen).

```
[modifier] enum ClassName {name1[, name2...]}
```

Die Ausgabe des Programms

- ✓ Der Name der Wertemenge entspricht den Anforderungen für Bezeichner und beginnt wie ein Klassenname gewöhnlich mit einem Großbuchstaben.
- ✓ In geschweiften Klammern `{ }` folgen mit Kommata getrennt die Aufzählungselemente, die häufig wie Konstanten in Großbuchstaben geschrieben werden.

Beispiele für einfache Enumerations: `com\herdt\java9\kap12\Enumerations.java`

```
enum Days
{MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY}
enum WorkDays {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY}
enum Grades {VERYGOOD, GOOD, SATISFACTORY, ADEQUATE, FAIL}
```

Enumerations verwenden

Auf die Elemente einer Aufzählung greifen Sie wie auf statische Attribute zu.

```
Days.MONDAY
```

Beispiel für den Zugriff auf ein Element

Beachten Sie, dass in dem vorherigen Beispiel `Days.MONDAY` und `WorkDays.MONDAY` nicht kompatibel sind. Die nebenstehende `if`-Anweisung führt aufgrund der unterschiedlichen Datentypen zu einem Compilerfehler.

```
if (Days.MONDAY == WorkDays.MONDAY)
```

Diese Anweisung führt zu einem Compilerfehler

Attribute und Methoden für Enumerations definieren

Enumerations werden vom Compiler als einzelne Klasse kompiliert und als `.class` Datei gespeichert. Im Regelfall werden sie deshalb auch in einer eigenen Datei definiert.

Jede Enumeration erbt von der Klasse `Enum` und besitzt dadurch einige grundlegende Methoden:

<code>values()</code>	gibt alle Elemente der Enumeration als Array zurück
<code>valueOf(String name)</code>	gibt das Element der Enumeration mit dem angegebenen Namen zurück
<code>valueOf(Class c, String name)</code>	gibt das Element der Enumeration der angegebenen Klasse mit dem angegebenen Namen zurück

Wie eine Klasse kann eine Enumeration auch Attribute und Methoden beinhalten. Diese werden ebenfalls innerhalb der geschweiften Klammern definiert. Die Aufzählung der Elemente wird mit einem Semikolon abgeschlossen.

Beispiel: `com\herdt\java9\kap12\Creature.java`

Im folgenden Beispiel wird eine Enumeration `Creature` definiert. Jede der enthaltenen Tiergruppen soll dabei durch die Anzahl der Beine beschrieben werden.

```

① public enum Creature
  {
②     SNAKE(0), LIZARD(4), FISH(0), SPIDER(8), INSECT(6);
③     private int legs;
④     Creature(int legs)
        {
            this.legs = legs;
        }
⑤     int getLegs()
        {
            return legs;
        }
  }
```

```
⑥ String description()
{
⑦   String s = this.toString() + "s have ";
   if (getLegs() > 0)
       s = s + getLegs();
   else
       s = s + "no";
   return s + " legs.";
}
```

- ① Die Enumeration `Creature` wird mit dem Schlüsselwort `enum` definiert.
- ② Es werden fünf Aufzählungselemente festgelegt.
- ③ Die Enumeration erhält ein gekapseltes Attribut `legs`.
- ④ Ein spezieller Konstruktor nimmt als Parameter die Anzahl der Beine (`legs`) entgegen.
- ⑤ Die Getter-Methode ermöglicht die Ausgabe des Attributs `legs`.
- ⑥ Die Methode `description` erstellt eine Beschreibung als `String`.
- ⑦ Mithilfe der Methode `toString` wird der Name des Aufzählungselements in einen `String` ausgegeben.

Obwohl für die Enumeration `Creature` ein Konstruktor definiert wurde, können nachträglich keine weiteren Elemente hinzugefügt werden.

Beispiel zur Verwendung der Aufzählung `Creature`:
com\herdt\java9\kap12\EnumerationTyp.java

```
package com.herdt.java9.kap12;
import static com.herdt.java9.kap12.Creature.*;

public class EnumerationTyp
{
    public static void main(String[] args)
    {
①      Creature oneCreature = SNAKE;
        Creature anotherCreature = SPIDER;
②      System.out.println(oneCreature.description());
        System.out.println(anotherCreature.description());
    }
}
```

- ① Zwei Variablen werden mit der Enumeration `Creature` erzeugt.
- ② Auf die Werte bzw. die Methoden können Sie wie bei statischen Methoden und Attributen zugreifen.

Vorteile von Enumerations

Im Unterschied zu einzelnen statischen Konstanten (z. B. MONTAG, DIENSTAG etc. für Wochentage), welche vor der Einführung der Enumerations in Java 1.5 für derartige Anwendungsfälle genutzt werden mussten, können in einer Enumeration die Werte der Menge als Ganzes behandelt werden. Auch die Verwendung als Kriterium in einem `switch-case`-Block ist möglich.

Durch die Verwendung einer Enumeration kann in einem Programm Typsicherheit erreicht werden. Erwartet beispielsweise eine Methode einen Wochentag und der Parameter ist auf `String` typisiert, kann ihr prinzipiell jeder `String` übergeben werden. Erfolgt die Typisierung des Parameters auf eine Enumeration, nimmt die Methode nur noch Elemente des Aufzählungstyps entgegen.

Beispiel: `com\herdt\java9\kap12\Weekday.java`

Im folgenden Beispiel wird eine Enumeration `Weekday` definiert. Neben den einzelnen Wochentagen enthält die Enumeration eine statische Methode zur Ausgabe aller Tage.

```
package com.herdt.java9.kap12;  
① public enum Weekday  
{  
②     MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,  
    SUNDAY;  
③     static void writeAllDays()  
    {  
④         for (Weekday day : Weekday.values())  
            System.out.println(day);  
    }  
}}
```

- ① Die Enumeration `Weekday` wird mit dem Schlüsselwort `enum` definiert.
- ② Die Wochentage werden als Aufzählungselemente festgelegt.
- ③ Die Methode `writeAllDays` gibt alle Wochentage aus.
- ④ In einer `for`-Schleife wird die Enumeration durchlaufen und die Ausgabe erzeugt. Dabei findet die von der Klasse `Enum` geerbte Methode `values` Anwendung, welche alle Elemente der Enumeration liefert.

Beispiel zur Verwendung der Aufzählung `Weekday`: `com\herdt\java9\kap12\UseWeekday.java`

```
package com.herdt.java9.kap12;  
import com.herdt.java9.kap12.Weekday;  
  
public class UseWeekday  
{
```



```
① public void writeWeekday(Weekday day)
    {
        System.out.println("Heutiger Wochentag: " + day);
    }

    public static void main(String[] args)
    {
        ② Weekday.writeAllDays();
        UseWeekday instance = new UseWeekday();
        ③ instance.writeWeekday(Weekday.MONDAY);
    }
}
```


- ① Es wird die Methode `writeWeekday` definiert. Durch die Typisierung des Übergabeparameters auf die Enumeration `Weekday` nimmt die Methode nur in der Enumeration festgelegte Elemente entgegen. Bei einer Typisierung auf `String` könnte der Methode jede beliebige Zeichenkette (z. B. "Januar") übergeben werden, was eine inhaltlich falsche Ausgabe zur Folge hätte.
- ② Die statische Methode `writeAllDays` der Enumeration `Weekday` wird aufgerufen. Im Ergebnis werden alle Wochentage auf der Konsole ausgegeben.
- ③ Ein einzelner Wochentag soll ausgegeben werden. Durch die Typisierung der Methode auf die Enumeration sind hier nur deren Elemente als Übergabeparameter erlaubt. Die Angabe einer beliebigen Zeichenkette führt zu einer Fehlermeldung des Compilers.

```
C:\uebung\jav9 \com\herdt\java9\kap12\UseWeekday.java:15: error:
incompatible types: String cannot be converted to Weekday
    instance.writeWeekday("Januar");
                        ^
```

Compilermeldung bei Übergabe einer beliebigen Zeichenkette an die Methode `writeWeekday`


12.8 Übungen

Übung 1: Arrays verwenden

Level		Zeit	ca. 10 min
Übungsinhalte	✓ Anwendung von Arrays		
Übungsdatei	--		
Ergebnisdatei	CalcSquare.java		

1. Erzeugen Sie mit einem Array-Literal ein Array `prime` mit den sechs Werten 2, 3, 5, 7, 11, 13.
2. Lassen Sie mit einer sogenannten `foreach`-Schleife alle Werte ausgeben.
3. Erstellen Sie ein Array `squares` mit 10 Werten vom Typ `double`. Speichern Sie in dem Array mit einer Schleifenstruktur das Quadrat des jeweiligen Indexwertes.
4. Lassen Sie mit einer `for`-Schleife alle Werte ausgeben.
Beispiel: Das Quadrat von 5.0 ist 25.0
5. Kopieren Sie das Array `squares` in ein neues Array mit dem Namen `squaresCopy`.
6. Ändern Sie alle Werte des Arrays `squaresCopy` auf den Wert 0.5;
7. Lassen Sie mit einer Schleife die Werte der beiden Arrays `squares` und `squaresCopy` ausgeben.

Übung 2: Zweidimensionale Arrays verwenden

Level		Zeit	ca. 20 min
Übungsinhalte	✓ Anwendung von Arrays		
Übungsdatei	com.herdt.java9.kap09.forms.*		
Ergebnisdatei	CompareRectangle.java		

1. Erstellen Sie ein Programm mit dem Namen `CompareRectangle`.
2. Erstellen Sie ein zweidimensionales Array mit Objekten vom Typ `Rectangle` (im Package `com.herdt.java9.kap09.forms`).
3. Die erste Dimension des Arrays soll der Breite des Rechtecks und die zweite Dimension der Länge des Rechtecks entsprechen. Erzeugen Sie in einem Konstruktor alle Rechtecke mit einer Breite von 1 bis 8 und einer Länge von 1 bis 8. Die Schrittweite soll 1 betragen.
4. Schreiben Sie eine Methode `compare` mit einem Objekt vom Typ `Rectangle` als Parameter. Vergleichen Sie dieses Rechteck mit allen im Array gespeicherten Rechtecken. Handelt es sich nicht um dasselbe Objekt und sind die Flächeninhalte gleich (Methode `getArea`), dann soll eine Ausgabe erfolgen: Die Rechtecke (3 x 6) und (2 x 9) sind gleich gross.
5. Fügen Sie im Konstruktor eine Schleife ein, in der alle im Array gespeicherten `Rectangle`-Objekte durchlaufen werden, und rufen Sie für jedes dieser Objekte die Vergleichsmethode `Compare` auf.
6. Erzeugen Sie in der `main`-Methode mit dem Konstruktor ein Objekt der Klasse `CompareRectangle`.

13

Collections-Framework – Grundlagen

13.1 Grundlagen zum Java-Collections-Framework

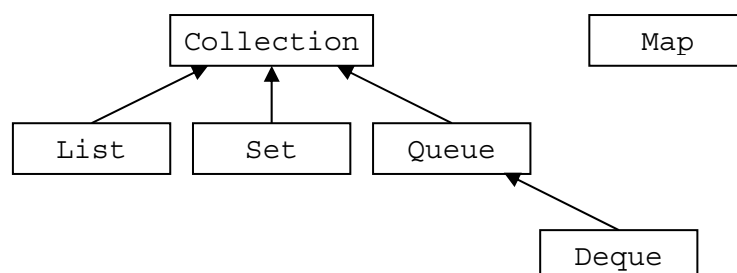
Was sind Collections?

Collections, auch **Container** genannt, sind Datenstrukturen, die eine Gruppe von Daten in einer Einheit zusammenfassen. Die Daten (Objekte) der Collections werden auch als Elemente bezeichnet. Auf die Daten darf nur mit entsprechenden Methoden zugegriffen werden. Beispiele für Datenstrukturen sind:

- ✓ **Liste**: erlaubt Duplikate, kann sortiert sein, der Zugriff auf die Elemente kann sequenziell oder wahlfrei erfolgen. Die Anzahl der Elemente ist beliebig.
- ✓ **Menge**: Duplikate sind verboten; sie entspricht der Menge im mathematischen Sinn.

Interfaces-Hierarchie des Collections-Frameworks

Zur Verwaltung der Datenstrukturen wird im Package `java.util` eine Hierarchie von Interfaces bereitgestellt, die Bestandteil des sogenannten Java-Collections-Framework sind. Jedes dieser Interfaces wird von mehreren Klassen implementiert. Die Namen dieser Klassen beinhalten den Namen des implementierten Interface. So implementiert beispielsweise die Klasse `ArrayList` das Interface `List`. Sie können so am Namen der Klasse erkennen, zu welchem Grundtyp sie gehört.



Einige Interfaces des Collections-Frameworks

Die Interfaces `List`, `Set` und `Queue` sind von dem Interface `Collection` abgeleitet. Das Interface `Map` besitzt kein Basis-Interface, steht also an der Spitze der Hierarchie.

Collection	Enthält als Basis-Interface grundlegende Methoden zur Arbeit mit Collections
List	Eine beliebig große Liste von Elementen unterschiedlichen Typs, auf deren Elemente sequenziell oder wahlfrei zugegriffen werden kann
Set	Eine Menge von Elementen (im mathematischen Sinn) ohne Duplikate, auf die mit Mengenoperationen zugegriffen werden kann
Queue	Häufig als Warteschlange verwendete Liste, auf die nur sequenziell beispielsweise nach dem FIFO-Prinzip (First in first out) zugegriffen werden kann. Ein wahlfreier Zugriff ist nicht erlaubt.
Deque	Deque (double ended queue) beschreibt eine spezielle Form der Warteschlange, die es gestattet, sowohl am Anfang als auch am Ende Elemente hinzuzufügen oder zu entfernen. Ein wahlfreier Zugriff ist auch hier nicht erlaubt.
Map	Dient zur Implementierung von Schlüsselwertpaaren, also einer Menge zusammengehöriger Objektpaare

Vorteile des Collections-Frameworks

Mit dem Collections-Konzept stehen Ihnen viele Anwendungsmöglichkeiten und Funktionalitäten bezüglich der Speicherung und Verwaltung von großen Datenmengen zur Verfügung. Eine flexible Schnittstellengestaltung bei der Übergabe von Collection-Objekten wird durch deren Vererbungshierarchie ermöglicht.



Bis zur Version 7 waren Collections nicht synchronisiert. Wenn mehrere Threads (gleichzeitig ausgeführte Programmmodule) gleichzeitig auf dieselbe Collection zugreifen, mussten Sie als Programmierer den Zugriff auf die Collection kontrollieren. Es konnte sonst zu inkonsistenten Daten und zu Programmfehlern kommen. Die Arbeit mit mehreren Threads ist nicht Gegenstand dieses Buches. Die hier im Folgenden gezeigte Vorgehensweise entspricht der in Java 7. Seit Java 8 gibt es mit der Stream-API eine Erweiterung der Collections, welche die Programmierung mit mehreren Threads unterstützt. Grundlage dieser Erweiterung sind Lambda-Ausdrücke. Unter BuchPlus, Zusätzliche Inhalte, *JAV9_19_Funktionale_Programmierung.pdf* finden Sie Informationen für einen ersten Einblick in diese Technik, ohne dass dabei jedoch auf die Stream-API eingegangen wird.

Einheitliche Datentypen verwenden

Um sicherzustellen, dass die Datenelemente dem gewünschten Typ entsprechen, können Sie ein sogenanntes **Typargument** verwenden. Dazu geben Sie bei der Definition der Liste in spitzen Klammern `<>` den gewünschten Datentyp an.

```

① ArrayList<String> arrList = new ArrayList <String>();
   for (int i = 1; i <= 10; i++)
       arrList.add("Obj" + i);
② arrList.add(new Integer(12));

```

① Die Liste wird als „ArrayList mit String-Elementen“ definiert.

② Beim Anfügen eines Elements vom Typ `Integer` erfolgt beim Kompilieren eine Fehlermeldung. Die Liste soll laut der Definition ① `String`-Elemente enthalten. Klassen, die Typargumente besitzen, heißen **generische Klassen**. Durch die Angabe des Datentyps entsteht ein **generischer Datentyp**, wie in diesem Beispiel der generische Datentyp `ArrayList<String>`. Java 9 unterstützt einen Rückschluss auf den Typ (engl. *type inference*) bei der Erstellung generischer Instanzen. Solange der Compiler die notwendigen Argumente aus dem Kontext erschließen kann, kann als Parameter ein leerer Typ in der Form `<>`, der sogenannte **Diamond-Operator**, verwendet werden.

Die Angabe von Typargumenten wird in diesem Buch nur in einigen Beispielen gezeigt. Generische Datentypen sind nicht Gegenstand dieses Buches.

Unveränderliche Collections erstellen

Neu in Java 9 ist die einfache Erstellung einer Vorgabeliste unveränderlicher Werte auf Basis statischer Methoden, welche in den einzelnen Interfaces (`Set`, `List` und `Map`) implementiert sind.

①	<code>List<String> list = List.of("Montag", "Mittwoch", "Freitag");</code>
②	<code>Set<String> set = Set.of("Java", "C#", "VB.net", "C");</code>
③	<code>Map<String, Integer> PunktNummer = Map.of("A", 1, "B", 2);</code>

- ① Die Liste wird als `List` mit drei Texten definiert.
- ② Die Collection wird als `Set` mit vier Zeichenketten festgelegt.
- ③ Definition einer Collection vom Typ `Map` mit zwei Text-/Nummern-Paaren.

Hinweis zum Kompilieren von Quelltext

Beim Kompilieren eines Programms weist Sie der Compiler möglicherweise in einer Warnung darauf hin, dass unsichere und ungeprüfte Operationen ausgeführt werden sollen. Es erfolgt keine Überprüfung, ob die Datentypen der Elemente zueinanderpassen. Um detaillierte Informationen zu erhalten, kompilieren Sie den Quelltext, indem Sie an den Befehl die Option `-xlint` anfügen: `javac -xlint ...`

13.2 Das Interface `Collection`

Methoden des Basis-Interface `Collection<E>`

<code>boolean add(E o)</code>	<p>Fügt der Collection das übergebene Element hinzu.</p> <p>Der Rückgabewert der <code>add</code>-Methode ist <code>true</code>, wenn sich die Liste verändert hat, d. h. wenn Elemente angefügt wurden. Die Methode liefert <code>false</code>, wenn sich die Liste nicht verändert hat. Das kann z. B. auftreten, wenn ein Element doppelt in eine Liste eingefügt werden soll, die keine Duplikate erlaubt. Konnten Elemente nicht eingefügt werden, tritt eine Exception auf.</p>
-------------------------------	---

<code>void clear()</code>	Löscht alle Elemente der Collection
<code>boolean contains(Object o)</code>	Liefert <code>true</code> zurück, wenn das übergebene Objekt in der Collection enthalten ist
<code>boolean equals(Object o)</code>	Prüft das übergebene Objekt mit dieser Collection auf Gleichheit
<code>int hashCode()</code>	Gibt den Hashcode für diese Collection zurück
<code>boolean isEmpty()</code>	Gibt <code>true</code> zurück, wenn die Collection keine Elemente enthält
<code>Iterator<E> iterator()</code>	Gibt ein <code>Iterator</code> -Objekt über die Elemente dieser Collection zurück
<code>boolean remove(Object o)</code>	Entfernt das übergebene Objekt aus der Collection, wenn es vorhanden ist
<code>int size()</code>	Gibt die Anzahl der Elemente in der Collection zurück
<code>Object[] toArray()</code>	Gibt ein Array zurück, das alle Elemente der Collection enthält
<code><T> T[] toArray(T[] a)</code>	Gibt ein Array zurück, das alle Elemente der Collection enthält, dessen Laufzeittyp dem des übergebenen Arrays entspricht

Die Methoden werden von den Klassen, die das Interface `Collection` oder die nachfolgend beschriebenen Interfaces implementieren, bereitgestellt. Jedoch werden einige dieser Methoden in verschiedenen Klassen nicht unterstützt und lösen eine sogenannte Exception aus.

In der Java-Dokumentation der Interfaces sind diese Methoden als „optional“ gekennzeichnet ①. In der Dokumentation zu den jeweiligen Klassen finden Sie die Erläuterungen, welche Methoden für die Klasse zur Verfügung stehen und genutzt werden können

add

```
boolean add(E e)
```

①

Ensures that this collection contains the specified element (optional operation). Rel collection does not permit duplicates and already contains the specified element.)

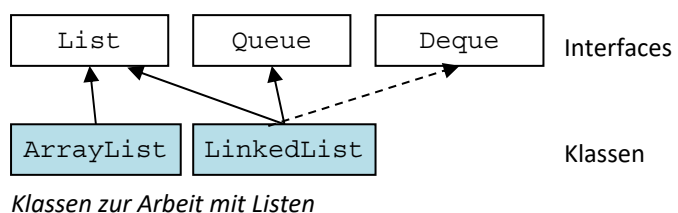
Als optional gekennzeichnete Methode

13.3 Mit Listen arbeiten

Objekte in einer Liste verwalten

Für geordnete Mengen von Daten (Objekten), auf die wahlfrei (über einen Index) oder sequenziell (der Reihe nach) zugegriffen werden soll, eignen sich **Listen**. Dies sind Collections vom Typ `List`, `Queue` bzw. vom Typ `Deque`.

- ✓ Ebenso wie bei Arrays beginnt der Index bei 0. Das letzte Element hat den Index `size() - 1`.
- ✓ Im Unterschied zu Arrays sind Listen dynamisch. Sie können während der Laufzeit Elemente aus der Liste löschen und neue



Elemente einfügen.

Eine Liste bearbeiten

Neben den vom Interface `Collection` geerbten Methoden stehen im Interface `List` zusätzliche Methoden z. B. für den wahlfreien Zugriff zur Verfügung.

<code>void add(int index, E element)</code>	Fügt das angegebene Element des entsprechenden Typs an der angegebenen Position (<code>index</code>) ein (optional)
<code>E get(int index)</code>	Gibt das Element zurück, das sich an der angegebenen Position befindet
<code>int indexOf(Object o)</code>	Ermittelt den Index des ersten Vorkommens des übergebenen Objekts. Ist das Objekt nicht enthalten, liefert die Methode den Wert <code>-1</code> zurück.
<code>E set(int index, E element)</code>	Ersetzt das Element an der durch <code>index</code> angegebenen Position durch das übergebene Element. Das ersetzte Element wird zurückgegeben.

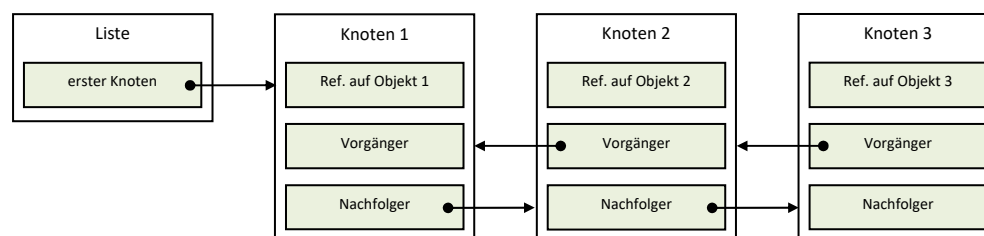
Einfach verkettete Listen erstellen

Eine Klasse, die das Interface `List` implementiert, ist die Klasse `ArrayList`. Mit der Klasse `ArrayList` lassen sich Listen erstellen, in denen die Objekte linear hintereinander gespeichert sind. Die Klasse `ArrayList` ist von der abstrakten Basisklasse `AbstractList` abgeleitet. Innerhalb einer Liste vom Typ `ArrayList` werden die Daten intern in einem Array gespeichert.

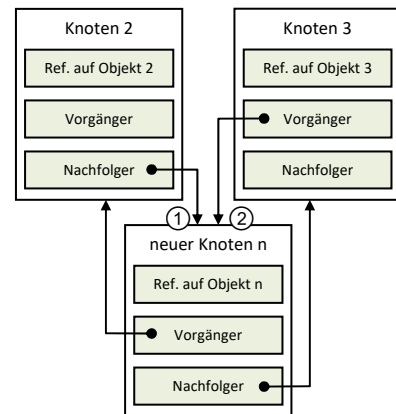
Doppelt verkettete Listen erstellen

Eine einfache Liste vom Typ `ArrayList` hat den Nachteil, dass beim Löschen eines Objekts innerhalb der Liste der Rest der Liste jeweils um eine Position nach vorn kopiert werden muss. Beim Einfügen eines neuen Objekts in die Liste finden diese Kopiervorgänge ebenfalls statt, nur in die andere Richtung, damit eine Position für das neue Objekt frei wird.

Das Problem wird durch die Klasse `LinkedList` gelöst.



- ✓ Eine Liste vom Typ `LinkedList` besitzt für jedes Objekt einen sogenannten Knoten.
- ✓ Neben der Referenz auf das Objekt sind in jedem Knoten ein Verweis auf den vorherigen und ein Verweis auf den nachfolgenden Knoten gespeichert. Die Liste wird daher als **doppelt verkettet** bezeichnet.
- ✓ Beim Einfügen eines Objekts müssen somit im Vorgänger- und im Nachfolgerknoten nur noch die Verweise auf den Nachfolger ① und den Vorgänger ② geändert werden.
- ✓ Auch beim Löschen von Objekten sind nur die Verweise im Vorgänger- und im Nachfolgerknoten anzupassen.
- ✓ Diese Vorgehensweise macht sich als deutlicher Geschwindigkeitsvorteil besonders bei großen Datenmengen bemerkbar, bei denen häufig Einfüge- und Löschoperationen anfallen.



13.4 Listen sequenziell durchlaufen

Für die sequenzielle Verarbeitung einer `ArrayList` bzw. einer `LinkedList` verwenden Sie einen Positionszeiger, der auf das jeweils aktuelle Element der Liste verweist. Die Klasse `ArrayList` implementiert über die abstrakte Klasse `AbstractList` das Interface `Iterable`. Die Methoden eines Positionszeigers sind in den Interfaces `Iterator` bzw. `ListIterator` festgelegt.

Die entsprechenden Klassen, die die Interfaces `Iterator` bzw. `ListIterator` implementieren, sind intern verborgen in der Klasse `ArrayList` bzw. `LinkedList` beschrieben.

Positionszeiger vom Typ `Iterator<E>` verwenden

Ein `Iterator` dient dazu, eine Collection sequenziell (von vorn nach hinten) zu durchlaufen. Folgende Methoden werden dazu implementiert:

boolean <code>hasNext()</code>	Prüft, ob weitere Elemente in der Collection vorhanden sind
E <code>next()</code>	Holt das nächste Element der Collection und setzt den Positionszeiger auf das nächste Element
void <code>remove()</code>	Entfernt das Element aus der Collection, welches beim letzten Aufruf der Methode <code>next</code> zurückgegeben wurde (optional)

- ✓ Mit der Methode `iterator` der Klasse `ArrayList` bzw. der Klasse `LinkedList` erzeugen Sie einen Positionszeiger für die Liste.
- ✓ Nach der Erzeugung des Positionszeigers verweist dieser auf das erste Element der Liste.
- ✓ Der Positionszeiger genügt den Anforderungen des Interface `Iterator`.

Seit der Version Java 6 implementiert die Klasse `LinkedList` über das Interface `Deque` eine Methode `descendingIterator`. Diese Methode erzeugt einen weiteren Positionszeiger, mit dem die `LinkedList` in umgekehrter Richtung, d. h. von hinten nach vorn, durchlaufen werden kann.

Positionszeiger vom Typ `ListIterator<E>` verwenden

Das von `Iterator` abgeleitete Interface `ListIterator` erlaubt zusätzlich das Durchlaufen einer Liste in umgekehrter Richtung und das Modifizieren der Liste während der Iteration. Ein `ListIteration`-Objekt erhalten Sie durch Aufrufen der Methode `listIterator` eines `List`-Objekts. Das Interface bietet zusätzlich folgende Methoden:

<code>boolean hasNext()</code>	Prüft, ob es ein Vorgängerelement gibt. Diese Methode wird benötigt, wenn Sie die Liste rückwärts durchlaufen wollen.
<code>E previous()</code>	Liefert das vorhergehende Element
<code>int nextIndex()</code>	Liefert den Index des Elements, das beim nächsten Aufruf der Methode <code>next</code> angesprochen wird
<code>int previousIndex()</code>	Liefert den Index des Elements, das beim nächsten Aufruf der Methode <code>previous</code> angesprochen wird
<code>void add(E o)</code>	Fügt das angegebene Objekt <code>o</code> des entsprechenden Typs vor dem nächsten Element in die Liste ein
<code>void set(E o)</code>	Ersetzt das zuletzt über <code>next</code> oder <code>previous</code> gelieferte Element durch das übergebene Objekt des entsprechenden Typs

- ✓ Mit der Methode `listIterator` der Klasse `ArrayList` bzw. der Klasse `LinkedList` erzeugen Sie einen `ListIterator` für die Liste.
- ✓ Wenn Sie die Methode `listIterator` ohne Parameter verwenden, verweist der Iterator nach der Erzeugung auf das erste Element der Liste.
- ✓ Sie haben die Möglichkeit, einen Iterator zu erzeugen, der auf ein spezielles Element der Liste verweist. Dazu existiert eine zweite Methode `listIterator`. Die Methode benötigt als Parameter den entsprechenden Index des Listenelements.
- ✓ Der `ListIterator` genügt den Anforderungen des Interface `ListIterator`.

Der Aufruf von `next` löst eine Exception vom Typ `NoSuchElementException` aus, falls die Liste keine weiteren Elemente enthält. Prüfen Sie daher immer mit der Methode `hasNext`, ob noch Elemente vorhanden sind, bevor Sie die Methode `next` anwenden. Beim Rückwärtsdurchlaufen der Liste prüfen Sie entsprechend mit `hasPrevious`.

Beispiel für die sequenzielle Bearbeitung von Listen: *com\herdt\java9\kap13\Iteration.java*

Das folgende Beispiel zeigt die Verwendung eines Iterators und eines `ListIterator`s. Zunächst wird eine `ArrayList` mit Elementen gefüllt. Danach werden einzelne Elemente gelöscht und es wird geprüft, ob ein Element in der Liste enthalten ist. Mithilfe eines `ListIterator`s wird eine Teilliste erzeugt und durchlaufen.

```

package com.herdt.java9.kap13;
import java.util.*;
import static java.lang.System.*;

public class Iteration
{
    public static void main(String[] args)
    {
        ① ArrayList<String> arrList = new ArrayList<String>();
        int x = 0;
        ② for (int i = 1; i <= 10; i++)
            arrList.add("Obj" + i);
        ③ Iterator<String> iter = arrList.iterator();
        ④ while (iter.hasNext())
        {
            ⑤ out.print(iter.next()); //das 1. Objekt und jedes 2.
            //jedes 2. Objekt soll geloescht werden
            ⑥ if ((x++ % 2) == 0)
            {
                ⑥ iter.remove();
                out.println(" - Objekt geloescht" );
            }
            else
                out.println("");
        }
        //Positionszeiger erneut erzeugen
        ⑦ iter = arrList.iterator(); //(auf des erste Element)
        out.println("\nAusgabe der einzelnen Listelemente:");
        ⑧ while (iter.hasNext())
            out.print(iter.next() + " - ");
        out.print("\nPruefung: ");
        ⑨ if (arrList.contains("Obj8"))
            out.println("Objekt 8 ist in der Liste enthalten");
        ⑩ ListIterator<String> listIter =
            arrList.listIterator(arrList.indexOf("Obj8"));
        out.println
        ("\nListe ab aktueller Position rueckwaerts durchlaufen:");
        ⑪ while (listIter.hasPrevious())
            out.print(listIter.previous() + " - ");
        out.println();
    }
}

```

- ① Ein ArrayList-Objekt für Elemente vom Typ String wird erzeugt.
- ② Die Liste arrList wird mit 10 Elementen (String-Objekten) gefüllt. Das Hinzufügen der Elemente erfolgt mithilfe der Methode add.

- ③ Für die Liste `arrList` wird mit der Methode `iterator` ein `Iterator`-Objekt `iter` angelegt. `iter` zeigt auf das erste Element der Liste.
- ④ Mithilfe der `while`-Schleife wird die Liste so lange durchlaufen, bis alle Elemente der Liste verarbeitet sind. Das Listenende ist erreicht, wenn die Methode `hasNext` den Wert `false` liefert.
- ⑤ Mit der Methode `next` wird das aktuelle Element geliefert und der Zeiger auf das nächste Element der Liste gesetzt. Da die in der Liste verwalteten Objekte vom Typ `String` sind, kann das Element ohne Konvertierung ausgegeben werden. Bei der Verwaltung anderer Objekttypen wäre eine Umwandlung in den Typ `String` notwendig (Methode `toString`).
- ⑥ Aus der Liste wird jedes zweite Element mit der Methode `remove` entfernt.
- ⑦ Um das Ergebnis der Bearbeitung anzuzeigen, sollen die Listenelemente erneut sequenziell angezeigt werden. Mit der Methode `iterator` wird ein neuer Positionszeiger für die Liste erzeugt. `iter` zeigt wieder auf das erste Element der Liste.
- ⑧ Die Liste, die nun nur noch aus geradzahligen Objektnamen besteht, wird erneut durchlaufen.
- ⑨ Über die Methode `contains` wird überprüft, ob das Objekt `Obj8` in der Liste enthalten ist.
- ⑩ Es wird ein Positionszeiger vom Typ `ListIterator` erzeugt, der auf das Element `Obj8` zeigt.
- ⑪ Mithilfe eines `ListIterator`s wird die Liste ausgehend vom Element `Obj8` rückwärts durchlaufen werden. Dazu liefert die Methode `previous` das Vorgängerobjekt.

```
Obj1 - Objekt gelöscht
Obj2
Obj3 - Objekt gelöscht
Obj4
Obj5 - Objekt gelöscht
Obj6
Obj7 - Objekt gelöscht
Obj8
Obj9 - Objekt gelöscht
Obj10
```

Ausgabe der einzelnen Elemente der Liste:

```
Obj2 - Obj4 - Obj6 - Obj8 - Obj10 -
```

Pruefung: Objekt 8 ist in der Liste enthalten

Liste ab der aktuellen Position rueckwaerts durchlaufen:

```
Obj6 - Obj4 - Obj2 -
```

Die Ausgabe des Programms

Foreach-Schleifen bei Listen verwenden

Wie bei Arrays können Sie die verkürzte Schreibweise für Schleifen, die **foreach-Schleifen**, auch bei Listen verwenden ①, wenn ...

```
ArrayList<String> arrList = new ArrayList <String>();
for (int i = 1; i <= 10; i++)
    arrList.add("Obj" + i);
```

```
for (String element : arrList) ①
    System.out.println(element);
```

Beispiel: `com\herdt\java9\kap13\ArrayListDemo.java`

- ✓ die Listenelemente nur ausgelesen und nicht verändert werden,
- ✓ die Liste sequenziell beginnend beim ersten Element durchlaufen werden soll,
- ✓ nur **eine** Liste durchlaufen wird.

Tipp zur Ausgabe von Listen mit wenigen Elementen

Eine Methode von Listen, die beim Testen sehr nützlich sein kann, ist die Methode `toString`. Wenden Sie diese Methode auf eine Liste an, erhalten Sie in eckigen Klammern eine Aufzählung der enthaltenen Elemente.

Die Anweisung

```
System.out.println(arrList.toString());
```

liefert beispielsweise die Ausgabe:

```
[Obj2, Obj4, Obj6, Obj8, Obj10]
```



Ergänzende Lerninhalte: *Jav09_Kap13_Collections-Framework_Weiterführende_Themen.pdf*

Inhalt:

- ✓ Hash-Tabellen und Bäume
- ✓ Sets – Collections vom Typ `Set`
- ✓ Maps – Collections vom Typ `Map<K, V>`

13.5 Übung

Collections

Level		Zeit	ca. 30 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Arbeit mit Collections ✓ Verwendung von ArrayList 		
Übungsdatei	--		
Ergebnisdateien	<i>Book.java, Exercise.java</i>		

1. Definieren Sie die Klasse `Book`. Sie dient zum Speichern der Daten über ein Buch und hat die Eigenschaften `author`, `title` und `issue`. Erstellen Sie Getter- und Setter-Methoden zum Speichern und Auslesen der Attribute. Im Konstruktor werden die übergebenen Werte gesetzt.

2. Erstellen Sie eine Anwendung mit dem Namen `Exercise`.
3. Definieren Sie ein Array, in dem sechs Objekte der Klasse `Book` gespeichert werden sollen.
4. Lassen Sie im Konstruktor der Klasse `Exercise` sechs Objekte der Klasse `Book` erstellen und in dem Array speichern. Dieses Array wird für die folgenden Übungsaufgaben ⑥–⑧ benötigt.
5. Für den nachfolgenden Aufgabenteil ⑥ ist eine Methode `part6` zu programmieren.
(Die Übung in der Datei *JAV9_13_Weiterführende_Themen* (BuchPlus, Ergänzende Lerninhalte) beinhaltet die Aufgabenteile ⑦ und ⑧ mit den dort zu erstellenden Methoden `part7` und `part8`.)
Der Aufruf der entsprechenden Methode erfolgt in Abhängigkeit vom Parameter, der der Anwendung übergeben wird. Der Parameter entspricht der Aufgabennummer. Wird das Programm z. B. mit Parameter 6 aufgerufen, soll die Methode `Part6` ausgeführt werden. Erstellen Sie im Konstruktor eine `if`-Verzweigung, in der Sie die entsprechende Methode aufrufen.
6. Erzeugen Sie für die Verwaltung der Bücher eine `ArrayList`. Fügen Sie in die `ArrayList` die sechs Buchobjekte des Arrays ein und geben Sie diese unsortiert, sortiert und in umgekehrter Reihenfolge sortiert aus. Die Ausgabe sollte den folgenden Aufbau besitzen:

```
Goethe: "Faust I"           Auflage: 20000 Stueck
Schiller: "Wilhelm Tell"    Auflage: 10000 Stueck
...
Fontane: "Effi Briest"      Auflage: 10000 Stueck
*** in umgekehrter Reihenfolge ***
Schiller: "Wilhelm Tell"    Auflage: 10000 Stueck
...
Fontane: "Effi Briest"      Auflage: 10000 Stueck
*** in sortierter Reihenfolge ***
Fontane: "Effi Briest"      Auflage: 10000 Stueck
...
Schiller: "Wilhelm Tell"    Auflage: 10000 Stueck
```

Um eine Sortierung in der gewünschten Form zu erhalten, müssen Sie in die Klasse `Book` die Methode `compareTo` implementieren.

14

Ausnahmebehandlung mit Exceptions

14.1 Auf Laufzeitfehler reagieren

Während der Laufzeit eines Programms können Fehler auftreten, die zum Zeitpunkt des Kompilierens nicht abzusehen sind. Diese Fehler werden **Laufzeitfehler** genannt und können aus dem Programm oder der Java- Laufzeitumgebung (der Laufzeitbibliothek bzw. der virtuellen Maschine) resultieren.

Sobald ein Laufzeitfehler auftritt, besteht eine Ausnahmesituation. Eine sogenannte **Exception** (Ausnahme) wird ausgelöst. Außerdem können Sie als Programmierer Exceptions auslösen. Java macht intensiven Gebrauch von Exceptions und der Compiler erwartet an bestimmten Stellen im Programm, dass Sie ein Exceptionhandling (Ausnahmebehandlung) zwingend durchführen.

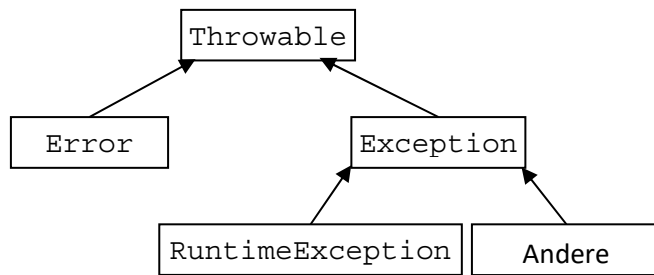
- ✓ Das Auslösen einer Exception wird **throw** (werfen) genannt.
- ✓ Exceptions werden immer innerhalb einer Methode ausgelöst.
- ✓ Das Behandeln einer Exception wird **catch** (auffangen) genannt.

Häufig wird der Begriff Exception im Zusammenhang mit einem aufgetretenen Fehler/Problem genannt. Dies ist nicht immer die treffende Bezeichnung. Eine Exception kennzeichnet eine besondere Situation, die eingetreten ist. Das muss kein Fehler im Programm sein. Zum Beispiel kann eine Exception ausgelöst werden, wenn Sie etwas drucken möchten, aber kein Papier im Drucker eingelegt ist. Durch das Abfangen dieser Exception können Sie dem Anwender mitteilen, dass er Papier einlegen und danach den Druckvorgang erneut starten soll.

Klassifizierung der Exceptions

In Java existiert eine Klassenhierarchie für Exceptions. Eine Exception ist ein Objekt einer Exception-Klasse.

- ✓ Durch das Auslösen einer Exception wird ein Objekt der betreffenden Exception-Klasse erzeugt.
- ✓ Durch den Garbage-Collector wird das Objekt automatisch wieder beseitigt.



Ableitungsbaum für die wichtigsten Exception-Klassen

Throwable	Dies ist die Superklasse aller Fehlerklassen in Java. Sie fangen normalerweise keine Exceptions dieser Klasse direkt ab. Throwable besitzt eine Methode getMessage, die einen (optionalen) Informationstext zur aufgetretenen Exception ausgibt.
Error	Diese Klasse ist die Superklasse aller Exceptions, die hauptsächlich im Zusammenhang mit der virtuellen Maschine auftreten. Diese sollten Sie nicht abfangen, sondern die Standardbehandlung beibehalten. Eine stabile Ausführung des Programms kann durch die virtuelle Maschine nicht mehr gewährleistet werden und das Programm wird daher abgebrochen.
Exception	Exceptions, die durch Ihr Programm behandelt werden können, werden von der Klasse Exception abgeleitet. Eigene Exception-Klassen sollten Sie von der Klasse Exception oder einer ihrer Subklassen bilden.
RuntimeException	RuntimeExceptions treten aufgrund eines Programmierfehlers ein, z. B. durch den Zugriff auf einen ungültigen Index eines Arrays oder fehlerhafte Typumwandlungen. Grundsätzlich sollte es durch geeignete Tests von Parametern und Variableninhalten sowie eine sorgfältige Programmierung möglich sein, RuntimeExceptions auszuschließen.
Andere	Diese Ausnahmen treten durch nicht kontrollierbare Situationen ein, z. B. durch das Öffnen einer Datei, die nicht existiert, oder durch Interpretationsfehler von Parametern.

- ✓ Beachten Sie, dass die Klasse Exception nur einen Teil des unter dem Begriff Exception zusammengefassten Exceptionhandlings ausmacht.
- ✓ Alle Exceptions sind Laufzeitfehler. Sie treten erst zur Laufzeit (Runtime) des Programms ein. Der Name RuntimeException kennzeichnet demgegenüber nur einen Teil der Exceptions.

Ausnahmen und Fehler unterscheiden

Tritt eine Exception auf, können Sie diese abfangen oder weiterleiten. Es wird in Java zwischen Exception (Ausnahme) und Error (Fehler) unterschieden.

Eine Exception ist ein behebbarer „Fehler“, der abgefangen werden kann bzw. abgefangen werden muss. Ihr Java-Programm kann nach einer Ausnahmebehandlung dennoch stabil weiterarbeiten.

Ein **Error** ist in der Regel ein schwerer Fehler, der einen Program Absturz verursachen kann (z. B. ist das Laufzeitsystem von Java nicht mehr stabil). Ein Java-Programm wird in der Regel dadurch beendet.

Die Ausnahmebehandlung sollte keine Tests auf gültige Werte ersetzen, da der Exception-Mechanismus wesentlich langsamer ist als beispielsweise ein Test mit einer `if`-Anweisung. Wird eine Anweisungsfolge jedoch sehr häufig aufgerufen und ist das Auftreten einer Exception sehr unwahrscheinlich, ist der Exception-Mechanismus effizienter, da die `if`-Bedingung nicht geprüft werden muss.

Kontrollierte und unkontrollierte Exceptions

Exceptions werden in kontrollierte und unkontrollierte Exceptions unterschieden.

- ✓ Ausnahmen vom Typ `Error` und `RuntimeException` sind unkontrollierte Exceptions, alle anderen sind kontrollierte Exceptions.
- ✓ Unkontrollierte Exceptions müssen nicht von Ihnen abgefangen werden.
- ✓ Exceptions des Typs `Error` können Sie im Programm nicht sinnvoll behandeln, da die Fehlerursache in der virtuellen Maschine liegt.
- ✓ `RuntimeExceptions` können Sie durch sorgfältige Programmierung ausschließen. Für unkontrollierte Exceptions existiert eine Standardbehandlung.
- ✓ Kontrollierte Exceptions müssen Sie im Programm behandeln. Anderenfalls meldet der Compiler einen Fehler.

Beispiel für das Auftreten einer Exception: `com\herdt\java9\kap14\ConvertToInt.java`

Im Beispiel wird über die Methode `parseInt` der Wrapper-Klasse `Integer` versucht, eine Zeichenkette in eine Zahl vom Typ `int` zu konvertieren. Dadurch, dass `ABC` keine Dezimalzahl ist, wird auf der Konsole eine Information ausgegeben, dass eine Exception vom Typ `java.lang.NumberFormatException` aufgetreten ist. Das Programm wird beendet.

```
public static void main(String[] args)
{
    String s = "ABC";
    int i = Integer.parseInt(s);
    System.out.println("Die Zahl ist " + i);
}
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "ABC"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    at com.herdt.java9.kap14.ConvertToInt.main(ConvertToInt.java:8)
```

Ausgabe der Fehlermeldung in der Konsole

14.2 Exceptions abfangen und behandeln

Alle Exceptions, die direkt oder indirekt von der Klasse `Exception` abgeleitet sind, müssen Sie behandeln oder weitergeben. Für Klassen, die von der Klasse `RuntimeException` und deren Subklassen abgeleitet wurden (vgl. folgendes Beispiel), gilt dies jedoch nicht. Um festzustellen, welche Exception eine Methode auslösen kann, verwenden Sie die Dokumentation zum JDK.

Hinter dem Methodennamen wird durch das Schlüsselwort `throws` angegeben, welche Exceptions durch die Methode ausgelöst werden können.

Beispielsweise kann durch die Methode `parseInt` ① die Exception `NumberFormatException` ② ausgelöst werden.

```
parseInt
public static int parseInt(String s)
    throws NumberFormatException
```

Die Methode `parseInt` der Klasse `Integer` (`java.lang`)

Vergessen Sie, eine kontrollierte Exception abzufangen, meldet der Compiler einen Fehler der Art `...unreported Exception: java.io.FileNotFoundException; must be caught or declared...`

Syntax für eine Exception-Behandlung

Um eine Exception zu behandeln, verwenden Sie einen `try-catch`-Block. Kann eine Methode eine Exception auslösen, müssen Sie diese in einen solchen Block einschließen.

- ✓ Mit dem Schlüsselwort `try` leitet Sie den `try`-Block ein.
- ✓ Es folgt in einem Anweisungsblock der reguläre Programmcode mit den Methodenaufrufen, die eine Exception auslösen können.
- ✓ Anschließend folgt der `catch`-Block, mit dem Sie eine oder mehrere Exceptions abfangen.
- ✓ Geben Sie hinter dem Schlüsselwort `catch` in Klammern `[]` den Exceptiontyp und den Namen für die Exception (Referenzvariable) ein. Wenn Sie als Typ beispielsweise die Klasse `Exception` verwenden, werden alle Exceptions abgefangen, die von der Klasse `Exception` oder davon abgeleiteten Klassen erzeugt wurden. Mehrere Exceptionstypen trennen Sie durch einen vertikalen Strich.
- ✓ Innerhalb des folgenden Anweisungsblocks können Sie über die Referenzvariable auf das Exception-Objekt zugreifen. Bei Behandlung von mehr als einer Exception in einem `catch`-Block ist der Parameter implizit `final`.
- ✓ Sie können mehrere `catch`-Blöcke hintereinander verwenden. Es wird aber nur einer dieser Blöcke ausgeführt.
- ✓ Wurde eine Exception von einer `catch`-Anweisung behandelt, wird das Exception-Objekt später, wenn keine Referenz mehr darauf existiert, vom Garbage-Collector beseitigt.

```
try
{
    // Regulaerer Programmcode
    ...
}
catch (Exceptiontype1 |
      Exceptiontype2 exc)
{
    //Behandlung für Exceptiontype1
    und Exceptiontype2
}
catch (Exceptiontype3 exc)
{
    //Behandlung für Exceptiontype3
}
...
```

Die Programmausführung wird hinter dem letzten `catch`-Block fortgesetzt.

- ✓ Wird keine Exception ausgelöst, wird die Programmausführung hinter der letzten `catch`-Anweisung fortgesetzt.
- ✓ Wird eine Exception durch keinen `catch`-Block abgefangen, weil sie nicht mit dem abgefangenen Typ zuweisungskompatibel ist, gilt diese Exception als nicht behandelt.
- ✓ `try`- und `catch`-Block müssen **unmittelbar** aufeinander folgen.

Wenn Sie mehrere Exception-Typen abfangen möchten, müssen Sie die `catch`-Blöcke so anordnen, dass sie Superklassen von Exceptions **nach** den jeweiligen Subklassen abfangen. Wenn Sie beispielsweise zuerst Exceptions vom Typ der Klasse `Exception` abfangen würden, dann würden nachfolgende `catch`-Blöcke, die davon abgeleitete Exceptions abfangen (`RuntimeException`, `IOException`), niemals erreicht werden.

Beispiel: `com\herdt\java9\kap14\TryAndCatch.java`

Der folgende Programmcode zeigt eine Modifizierung des Beispiels aus dem vorherigen Abschnitt. Durch das Einschließen der Anweisung in einen `try-catch`-Block wird die Exception abgefangen (`try`) und behandelt (`catch`).

```
public static void main(String[] args)
{
    String s = "ABC";
    ① try
    {
    ②     int i = Integer.parseInt(s);
        System.out.println("Die Zahl ist " + i);
    }
    ③ catch (NumberFormatException e)
    {
    ④     System.out.println(s +
        " konnte nicht in eine Zahl umgewandelt werden.");
    }
}
```

- ① Der `try`-Block beginnt mit dem Schlüsselwort `try`.
- ② Innerhalb der geschweiften Klammern des `try`-Blocks steht der ursprüngliche Programmcode, der die Exception auslöst (eine `NumberFormatException`).
- ③ Mit der `Catch`-Anweisung wird die Exception abgefangen.
- ④ Innerhalb der geschweiften Klammern des `catch`-Blocks wird die Exception behandelt: Ein entsprechender Text wird ausgegeben.

Wichtige Methoden

Wie die meisten anderen Klassen besitzen auch Exceptions Methoden. Durch die Superklasse Throwable werden unter anderem die Methoden `getMessage`, `toString` und `printStackTrace` an alle anderen Exception-Klassen vererbt. In den Subklassen können weitere Methoden definiert sein. Standardmäßig besitzen alle Exception-Klassen zwei Konstruktoren, einen parameterlosen und einen, dem eine Zeichenkette übergeben wird. Diese Zeichenkette kann über `getMessage` abgefragt werden.

<code>String getMessage()</code>	Gibt Antwort auf die Frage, was passiert ist Liefert den Text, der beim Erzeugen des Exception-Objekts im Konstruktor angegeben wurde. Er ist null, wenn der parameterlose Konstruktor verwendet wurde.
<code>String toString()</code>	Liefert einen String, der sich aus dem Namen der Exception-Klasse, einem Doppelpunkt oder Leerzeichen und dem Text, den <code>getMessage</code> liefert, zusammensetzt
<code>printStackTrace()</code>	Liefert eine Antwort auf die Frage, wo etwas passiert ist Gibt den Inhalt des Stacks auf der Standardfehlerausgabe (standardmäßig der Konsole) aus

Mithilfe der Methode `printStackTrace` können Sie den Weg einer Exception durch die Anwendung verfolgen (backtrace):

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "ABC"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    at com.herdt.java9.kap14.ConvertToInt.main(ConvertToInt.java:8)
```

Bedeutung der Bestandteile:

- ✓ `java.lang.NumberFormatException`:
→ Exception-Typ
- ✓ `For input string: "ABC"`
→ Text der Fehlermeldung
- ✓ `java.lang.NumberFormatException.forInputString`
→ Klasse und Methode, in der die Exception ausgelöst wurde
- ✓ `(NumberFormatException.java:65)`
→ Dateiname und Zeilennummer

Falls die Methode wiederum von einer anderen Methode aufgerufen wurde, folgen wie in diesem Beispiel weitere `at`-Meldungen. Wird eine Exception nicht behandelt (`RuntimeException`) oder bis zuletzt (main-Methode) weitergegeben, wird die Methode `printStackTrace` aufgerufen. Im letzteren Fall wird auch das Programm beendet.

Beispiel: `com\herdt\java9\kap14\Except.java`

In diesem zweiten Beispiel wird ein Feld erzeugt und gefüllt. Dabei wird auf einen ungültigen Index zugegriffen und dadurch eine Exception ausgelöst. Die Exception wird über einen `catch`-Block behandelt.

Das Beispiel dient nur zur Veranschaulichung der Fehlerauswertung. Sie müssen im Programmcode sicherstellen, dass der Index im gültigen Wertebereich liegt.

```
public static void main(String[] args)
{
  ① int field[] = new int[10];
  ② try
  {
    ③ for (int j = 0; j <= field.length; j++)
      field[j] = j;
  }
  ④ catch (ArrayIndexOutOfBoundsException ex)
  {
    ⑤ System.out.println("Sie haben die Feldgrenzen verletzt");
    ⑥ System.out.println(ex.getMessage());
    ⑦ System.out.println(ex.toString());
  }
}
```

- ① Ein Array wird dynamisch mit 10 Einträgen erzeugt. Dieses Feld soll im Folgenden mit Zahlenwerten vom Typ `int` gefüllt werden.
- ② Mit dem Schlüsselwort `try` wird der `try`-Block eingeleitet.
- ③ Das Feld wird mit Werten gefüllt. Dabei werden „versehentlich“ (fehlerhafte Angabe von `<=` statt `<`) die Feldgrenzen überschritten, die bei 0 und `field.length-1` liegen. Der Zugriff `field[field.length]` führt zu einer Exception `ArrayIndexOutOfBoundsException`.
- ④ Die Exception `ArrayIndexOutOfBoundsException` wird abgefangen. Über die Variable `ex` können Sie auf das Exception-Objekt zugreifen.
- ⑤ Eine allgemeine Fehlermeldung wird ausgegeben.
- ⑥ Mit der Methode `getMessage` der Klasse `Throwable` wird die Fehlerbeschreibung der Exception ausgelesen und angezeigt. Die Methode gibt den fehlerhaften Indexwert zurück (10).
- ⑦ Die Methode `toString` gibt eine Meldung aus, die sich aus dem vollständigen Klassennamen der Exception (`java.lang.ArrayIndexOutOfBoundsException`) und gegebenenfalls dem Text, den `getMessage` liefert, mit einem Doppelpunkt getrennt, zusammensetzt.

```
Sie haben die Feldgrenzen verletzt
10
java.lang.ArrayIndexOutOfBoundsException: 10
```

Ausgabe der Fehlermeldung in der Konsole

Beispiel zur differenzierten Fehlerauswertung: com\herdt\java9\kap14\UsingStackTrace.java

Die Klasse Throwable besitzt eine Methode `getStackTrace`, über die Sie auf die Elemente des Stacks zugreifen können. Auf diese Weise lassen sich Fehler differenziert auswerten.

```
catch (NumberFormatException e)
{
    StackTraceElement[] stea = e.getStackTrace();
    StackTraceElement ste = stea[0];
    System.out.println
        ("Fehler in Datei " + ste.getFileName() + ", Zeile " +
         ste.getLineNumber());
    System.out.println("Klasse: " + ste.getClassName());
    System.out.println("Methode: " + ste.getMethodName());
}
```

Tipps für die Verwendung von **try-catch**-Blöcken

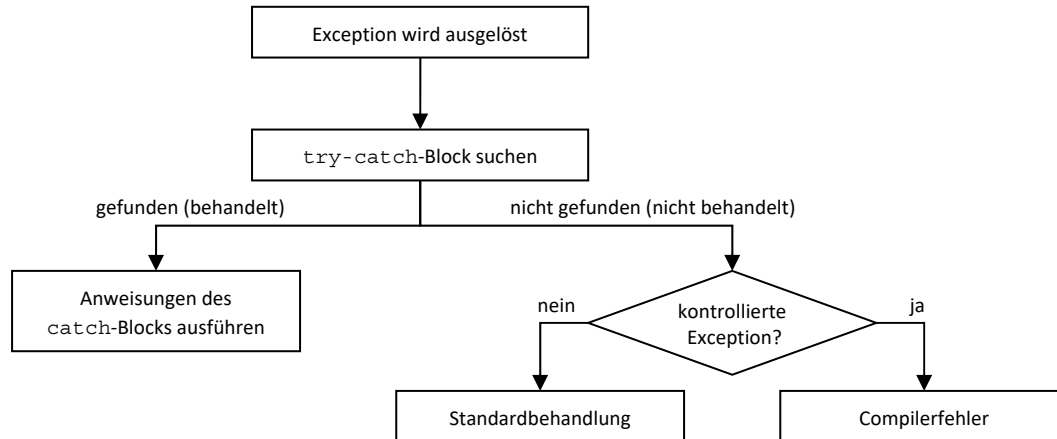
- ✓ Schließen Sie nicht **einzelne** Anweisungen in try-catch-Blöcke ein. Dies vermindert die Lesbarkeit des Programms. Verwenden Sie stattdessen mehrere catch-Blöcke, die die jeweiligen Exception-Typen abfangen, oder behandeln Sie mehrere Exceptions in einem catch-Block.
- ✓ Führen Sie wenn möglich einen einfachen Vergleich bzw. Test der Daten durch, statt im Problemfall auf eine Exception zu reagieren. Falls eine Exception ausgelöst wird, ist der Exception-Mechanismus sehr viel langsamer als beispielsweise eine `if`-Anweisung. Je nach Häufigkeit der Ausführung des Tests bzw. der Möglichkeit des Auftretens einer Exception kann entweder der Test über `if` oder die Exception-Behandlung effizienter sein.

Besser	Schlechter
<pre>if (i == 0) ergebnis = 0; // führe keine Division durch i durch else ergebnis = 100 / i;</pre>	<pre>try { ergebnis = 100 / i; } catch (Exception e) { ergebnis = 0; }</pre>

- ✓ Beachten Sie, dass alle nach einer aufgetretenen Exception folgenden Anweisungen bis zum nächsten catch-Block übersprungen werden.

14.3 Exceptions weitergeben

Java besitzt einen festgelegten Mechanismus, wie Exceptions in einem Programm weitergegeben werden.



- ✓ Eine Exception wird in einer Methode ausgelöst.
- ✓ Es wird nach einem try-catch-Block gesucht, der die betreffende Anweisung „umschließt“ und die ausgelöste Exception behandelt.
- ✓ Eine Methode, in der möglicherweise Exceptions ausgelöst werden, muss die Exceptions nicht selbst behandeln, sondern kann sie an die aufrufende Methode weitergeben. In diesem Fall wird in der aufrufenden Methode nach einem entsprechenden try-catch-Block gesucht.
- ✓ Wird ein passender catch-Block gefunden, der die betreffende Exception abfängt, wird die Exception behandelt, indem die Anweisungen des entsprechenden catch-Blocks ausgeführt werden.
- ✓ Wird die Exception nicht behandelt und handelt es sich um eine unkontrollierte Exception, wird die Standard-Exception-Behandlung durchgeführt.
- ✓ Wird eine kontrollierte Exception nicht behandelt, wird das Programm vom Compiler nicht übersetzt.

Aufgrund dieses Mechanismus müssen Sie alle kontrollierten Exceptions (die von der Klasse `Exception`, nicht aber `RuntimeException` abgeleitet sind) abfangen und behandeln oder weitergeben. Die Behandlung von Exceptions erfolgt über einen try-catch-Block, wobei die catch-Anweisung die betreffende Exception bzw. eine Superklasse dieser Exception abfangen muss.

Eine Methode, die Exceptions weitergibt, behandelt eine oder mehrere Exception-Typen nicht selbst, sondern teilt dem Compiler mit, dass die Exception auftreten kann, diese aber nicht in der Methode abgefangen wird.

Syntax für das Weitergeben von Exceptions

Mithilfe des Schlüsselworts `throws` geben Sie bekannt, dass die Methode die angegebenen Exception-Typen auslösen kann.

```
[modifier] type method([parameters]) throws Exceptiontype [,
Exceptiontype1, ...]
{
    // Programmcode, der obige Exceptions
    // auslösen kann
}
```

- ✓ Die Methode wird wie bisher definiert.
- ✓ Der Methode wird das Schlüsselwort `throws` angehängt.
- ✓ Geben Sie die Exceptiontypen an, die möglicherweise ausgelöst werden. Mehrere Exception-Typen trennen Sie durch Kommata.

Ablauf bei der Weitergabe einer Exception

- ✓ Eine Methode `methodA` kann eine Exception `Ex1` auslösen. Sie behandelt diese nicht selbst, sondern gibt diese weiter.
- ✓ Mit `throw` lösen Sie eine Exception `Ex1` aus.
- ✓ Die Methode `methodB` ruft die Methode `methodA` auf. Den Methodenaufruf können Sie in einen `try-catch`-Block einschließen und die betreffende Exception `Ex1` über `catch` abfangen.
- ✓ Die Methode `methodB` könnte die Exception aber auch wiederum weitergeben.
- ✓ Die Weitergabe kann bis zur `main`-Methode fortgeführt werden. Diese muss die Exception letztendlich abfangen, ansonsten erhalten Sie einen Compiler-Fehler (Ausnahme: `RuntimeException`).

```
methodA() throws
Ex1
{
    ...
    throw new Ex1();
}
methodB()
{
    try
    {
        methodA();
    }
    catch (Ex1 e)
    {
        // Behandlung
    }
}
```

Beispiel: `com\herdt\java9\kap14\PassException.java`

Im folgenden Programm wird eine Exception `ClassNotFoundException` zuerst weitergeleitet und danach behandelt. Diese Exception kann beim Erzeugen von Klassenobjekten ausgelöst werden.

```
public class PassException
{
    ① public PassException() throws ClassNotFoundException
    {
    ②    Class specialClass;
```

```

③    specialClass = Class.forName("xxx");
        //Die Klasse xxx existiert nicht
    }
④    public static void main(String[] args)
    {
⑤        try
        {
⑥            PassException w = new PassException();
        }
⑦        catch (ClassNotFoundException e)
        {
            System.out.println("Klasse nicht gefunden");
        }
    }
}

```

- ① Im Konstruktor der Klasse `PassException` wird durch die Methode `forName` ③ möglicherweise eine Exception vom Typ `ClassNotFoundException` ausgelöst. Über `throws` gibt der Konstruktor die Exception weiter.
- ② Die Variable `specialClass` speichert die Eigenschaften einer Klassendefinition (Klassenobjekt). Über diese Variable können dann z. B. Objekte der Klasse erzeugt werden.
- ③ Die Methode `forName` der Klasse `Class` (Package `java.lang`) gibt ein Objekt der Klasse mit dem als Parameter übergebenen Namen (`xxx`) zurück. Die angegebene Klasse mit dem Namen `xxx` existiert in diesem Fall nicht. Es wird eine Exception ausgelöst.
- ④ In der Methode `main` wird die Exception über einen `try-catch`-Block abgefangen. Sie hätten aber auch hier die Exception über `throws` weiterleiten können. In diesem Fall wird das Programm beendet und automatisch die Methode `printStackTrace` aufgerufen.
- ⑤ Der `try`-Block beginnt.
- ⑥ Es wird ein neues Objekt der Klasse `PassException` erzeugt.
- ⑦ Die Exception `ClassNotFoundException` wird abgefangen.

Ohne den `try-catch`-Block in der `main`-Methode hätte der Compiler die Übersetzung des Programms abgebrochen.

Tipps für die Weitergabe von Exceptions

- ✓ `RuntimeExceptions` sollten Sie nicht weitergeben. Diese Exceptions werden in der Regel durch unsachgemäße Programmierung hervorgerufen. Besser ist es in diesen Fällen, das Auftreten der Exception durch Tests und gewissenhafte Programmierung zu unterbinden.
- ✓ Methoden in abgeleiteten Klassen können nicht mehr Exceptions weitergeben als in der Basisklasse. Aus diesem Grund sollten Sie in bestimmten Fällen in den Methoden der Basisklasse auch Exception-Typen angeben, die in der Methode in abgeleiteten Klassen auftreten können.
- ✓ Warum sollten Sie überhaupt Exceptions weitergeben? Wenn Sie beispielsweise eine eigene Klasse entwickeln und ein Zustand eintritt, den Sie dem Anwender Ihrer Klasse übermitteln wollen, können Sie z. B. eine Exception auslösen. Diese muss der Anwender abfangen, d. h., Sie müssen die Exception zunächst weitergeben.

14.4 Abschlussarbeiten in einem **finally**-Block ausführen

Die Anweisungen in einem `catch`-Block werden nur im Falle einer Exception vom betreffenden Typ ausgeführt. Um einen Programmcode in jedem Fall auszuführen, unabhängig davon, ob eine Exception aufgetreten ist oder nicht, können Sie die `try-catch`-Struktur um einen sogenannten `finally`-Block erweitern.

Diese Anweisungen können z. B. Dateien schließen oder Daten in eine Datenbank schreiben (Ressourcenschutz). Die Anweisungen im `finally`-Block werden ausgeführt, wenn ...

- ✓ keine Exception ausgelöst wurde,
- ✓ eine Exception ausgelöst und in einem `catch`-Block behandelt wurde,
- ✓ eine Exception ausgelöst und nicht im `catch`-Block behandelt wurde,
- ✓ wenn der `try`-Block durch die Sprunganweisungen `return`, `break`, `continue` verlassen wird.

Beachten Sie, dass bei einer Programmbeendigung über `System.exit` der `finally`-Block nicht mehr ausgeführt wird.

Syntax für den **finally**-Block

- ✓ Die `try`-Anweisung leitet einen Exception-Block ein.
- ✓ Es folgt in einem Anweisungsblock der Programmcode, der Exceptions auslösen kann.
- ✓ Sie können dem `finally`-Block optional `catch`-Blöcke voransetzen. Der `finally`-Block muss hinter dem letzten `catch`-Block stehen.
- ✓ Die Anweisungen im `finally`-Block werden ausgeführt, unabhängig davon, ob im `try`-Block eine Exception ausgelöst wurde.

```
try
{
    //Anweisungen
}
catch (type var)
{
    //Behandlung
}
finally
{
    //wird immer
    ausgeführt
}
```

Sie können in der `main`-Methode ein `try-finally`-Konstrukt verwenden, um zum Programmende Aufräumarbeiten durchzuführen.



Besonderheiten bei **try-catch-finally**-Blöcken

- ✓ Achten Sie bei der Verwendung von geschachtelten `try-catch-finally`-Blöcken darauf, in welcher Reihenfolge die Anweisungen des `finally`-Blocks und des eigentlichen Exceptionhandlings ausgeführt werden. Behandelt keiner der inneren `catch`-Blöcke die aufgetretene Exception, werden zuerst die Anweisungen im inneren `finally`-Block ausgeführt, bevor die Fehlerbehandlung des umschließenden `try-catch`-Blocks ausgeführt wird.
- ✓ Wenn Sie einen `try-finally`-Block (ohne `catch`-Blöcke) verwenden, werden die Exceptions nicht behandelt. Dies muss in einem zusätzlichen `try-catch`-Block erfolgen, der den `try-finally`-Block umschließt.

14.5 Exceptions auslösen

Sie haben in Java die Möglichkeit, selbst Exception-Objekte zu erzeugen und auszulösen. Dadurch können Sie für Ihren eigenen Programmcode das gleiche Prinzip zur Ausnahmebehandlung einsetzen wie das JDK.

- ✓ Da Exceptions Objekte einer Exception-Klasse sind, müssen sie zuerst mit `new` erzeugt werden.
- ✓ Das Auslösen einer Exception erfolgt mit der `throw`-Anweisung.

Beispiel

Im folgenden Programmausschnitt wird überprüft, ob der Wert der Variablen `i` im Intervall 1 bis 100 liegt. Befindet er sich nicht darin, wird eine Exception erzeugt und ausgelöst. Durch den Aufruf von `throw` wird der aktuelle Programmblock verlassen und deshalb die Ausgabe ① auf der Konsole nicht mehr ausgeführt.

```
int i = -1;
if ((i < 1) || (i > 100))
    throw new IndexOutOfBoundsException
        ("i liegt nicht im Intervall 1...100");
System.out.println("Diese Zeile wird nicht angezeigt"); ①
```

Syntax für das Auslösen einer Exception

- ✓ Erzeugen Sie in einer Anweisung ein Exception-Objekt und lösen Sie die Exception über `throw` aus ①. Sie benötigen in diesem Fall keine zusätzliche Referenzvariable auf das Exception-Objekt.

```
throw new Exceptiontype(); ①
throw new Exceptiontype("..."); ②
...
Exceptiontype e; ③
e = new Exceptiontype(); ④
throw e; ⑤
```
- ✓ Zusätzlich können Sie im Konstruktor der Exception-Klasse einen Text angeben, der die Exception näher beschreibt ②. Einige Exception-Klassen können weitere Konstruktoren besitzen.
- ✓ Sie können das Erzeugen und Auslösen einer Exception auch in einzelnen Schritten durchführen ③–⑤.
- ✓ Die Anweisung `throw` löst die Exception aus.
- ✓ Der aktuelle Block des Programms wird verlassen und das Programm so lange durchlaufen, bis ein `catch`-Block gefunden wird, der die betreffende Exception abfängt, oder im Falle einer `RuntimeException` wird die Standardbehandlung durchgeführt.

Exception behandeln und erneut auslösen (Weitergabe von Exceptions)

In einigen Fällen ist es sinnvoll, eine Exception zu behandeln und dann erneut auszulösen. Auf diese Weise ist es möglich, eine eigene Exception-Behandlung mit einer Standardbehandlung oder Behandlung in einem umschließenden `catch`-Block zu verbinden.

Beispiel

```
...
catch (Exception e)
{
    //eigene Behandlung
    throw e; //das Objekt e existiert bereits
}
```

Mögliche Anwendungen der Weitergabe von Exceptions sind beispielsweise:

- ✓ Ausgabe einer Fehlermeldung zu Testzwecken, um die konkrete Position im Programm zu kennzeichnen, die die Exception ausgelöst hat. Die eigentliche Behandlung erfolgt an anderer Stelle.
- ✓ Sie können eine andere Exception erzeugen und weitergeben, als Sie ursprünglich abgefangen haben.
- ✓ Bei der Ausgabe von Daten in eine Datei wird eine Exception erzeugt. Sie schließen in der Exception-Behandlung die Datei und leiten die Exception weiter, um zu signalisieren, dass bei der Ausgabe ein Fehler aufgetreten ist.

Wenn Sie eine Exception weiterleiten, wird die Information, die durch die Methode `printStackTrace` ausgegeben wird, nicht geändert. Zum Beispiel wird die neue Zeilennummer, die der Auslöser der Exception war, nicht angepasst. Wenn Sie diese Information aktualisieren möchten, verwenden Sie die folgende Anweisung:

```
throw e.fillInStackTrace();
```

Dadurch wird die Information im Laufzeit-Stack aktualisiert. Sie müssen außerdem die Exception `Throwable` abfangen.

14.6 Eigene Exceptions erzeugen

Um eigene Exceptions zu erzeugen, leiten Sie eine neue Klasse von einer bereits vorhandenen Exception-Klasse ab und erweitern diese gegebenenfalls. Neue Exception-Klassen sollten nur von der Klasse `Exception` und davon abgeleiteten Klassen erzeugt werden, da diese Exceptions behandelt werden müssen. Die neue Exception-Klasse sollte mindestens zwei Konstruktoren (parameterlos und mit Meldungstext) besitzen. Sie können die neue Klasse jedoch beliebig erweitern, z. B. um weitere Informationen zum Grund der Exception weiterzugeben.

- ✓ Erstellen Sie eine neue Klasse, die Sie von der Klasse `Exception` oder einer davon abgeleiteten Klasse ableiten.
- ✓ Diese Klasse sollte einen Standardkonstruktor und einen Konstruktor, der einen `String` als Parameter entgegennimmt, besitzen.
- ✓ Die eigene `Exception` kann anschließend wie eine vordefinierte `Exception` ausgelöst und behandelt werden.

```
class NewExceptionType extends
Exception
{
    public NewExceptionType()
    {
        ...
    }
    public NewExceptionType(String msg)
    {
        super(msg); ...
    }
    //Weitere Anweisungen
}
```

Einen Namen für eine neue `Exception`-Klasse festlegen

Bei der Vergabe eines Namens für eine neue `Exception`-Klasse sollten Sie die folgenden dafür üblichen Konventionen beachten.

- ✓ Der Name der neuen Klasse sollte sich aus einer Beschreibung der neuen Funktionalität und dem Namen der alten Klasse zusammensetzen, z. B.
- ✓ `class RealNumberFormatException extends NumberFormatException`
- ✓ Der Text `Exception` sollte immer am Ende des Klassennamens angefügt werden. Damit ist die Klasse eindeutig als `Exception`-Klasse identifizierbar.
- ✓ Des Weiteren gelten die Regeln für Klassennamen.

Beispiel für einen neu erstellten Exceptiontyp:

com\herdt\java9\kap14\WrongIndex.java

und com\herdt\java9\kap14\IndexOutOfMinMaxException.java

Im Programm wird eine `Exception`-Klasse `IndexOutOfMinMaxException` definiert und verwendet. Diese `Exception` soll ausgelöst werden, wenn ein `Index` nicht in einem definierten Bereich liegt. Der Klasse wird außerdem ein weiterer Konstruktor hinzugefügt, dem die einzuhaltenden Grenzwerte übergeben werden.

```
① public class IndexOutOfMinMaxException extends Exception
  {
    ② public IndexOutOfMinMaxException(String msg)
      {
        ③ super(msg);
      }
    ④ public IndexOutOfMinMaxException()
      {
        super();
      }
  }
```

```
⑤ public IndexOutOfRangeException(int min, int max)
    {
⑥     super("Der Index liegt nicht zwischen " + min + " und " +
max);
    }
}
```


- ① Die neue Klasse `IndexOutOfRangeException` erweitert die Klasse `Exception`.
- ② Dieser Konstruktor erhält als Parameter einen String für weitere Informationen zur `Exception`.
- ③ Der Konstruktor der Superklasse wird aufgerufen.
- ④ Der Standardkonstruktor der Superklasse wird ohne Parameter aufgerufen.
- ⑤ Es wird ein neuer Konstruktor hinzugefügt, dem als Parameter die einzuhaltenden Grenzwerte übergeben werden.
- ⑥ Die Grenzwerte werden mit weiterem Text zu einer Fehlermeldung zusammengesetzt und dem Standardkonstruktor, der einen String als Parameter erwartet, übergeben.

In dem folgenden Beispiel `WrongIndex` wird ein `Exception`-Objekt der neuen Klasse erzeugt. Die `Exception` wird dann ausgelöst und abgefangen.

```
...
try
{
    ...
    throw new IndexOutOfRangeException(1, 100);
}
catch (IndexOutOfRangeException e)
{
    ...
}
...
```

14.7 Übung

Exceptions

Level		Zeit	ca. 30 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Exception erstellen und anwenden ✓ Exception abfangen und behandeln 		
Übungsdateien	<i>ConvertBinary.java, ConvertBinaryTest.java</i>		
Ergebnisdateien	<i>NoBinaryNumberException.java, BinaryStringToNumber.java, Exercise3.java, Exercise4.java, Exercise5.java</i>		

1. Erzeugen Sie eine neue Exception `NoBinaryNumberException`, die von der Klasse `RuntimeException` abgeleitet ist. Sie soll neben den beiden Standardkonstruktoren einen weiteren besitzen, der einen String (eine Binärzahl in der Form "10001110") sowie die Position, die die Exception ausgelöst hat, entgegennimmt.

Die bereits vorhandene Klasse `ConvertBinary` wandelt eine im String-Format übergebene Binärzahl in einen `int`-Typ um. Wird keine korrekte Zahl zur Umwandlung übergeben, so wird die Position des fehlerhaften Zeichens als negative Zahl zurückgegeben. Die Klasse `ConvertBinaryTest` verwendet die Klasse `ConvertBinary`).

Beispiel einer Binärzahl:

$$\begin{aligned}
 10111011 &= 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
 &= 187
 \end{aligned}$$

2. Kopieren Sie die Datei *ConvertBinary.java* in eine neue Datei und entwickeln Sie daraus eine Klasse `BinaryStringToNumber`:
 - ✓ Bei einem fehlerhaften Zeichen soll die Exception aus Übung ① mit dem neuen Konstruktor ausgelöst werden.
 - ✓ Leiten Sie in der Methode, die die Exception auslöst, die Exception an die umschließende Methode weiter.
3. Kopieren Sie das Beispielprogramm `ConvertBinaryTest` in `Exercise3`. Passen Sie die Klassennamen entsprechend an und testen Sie damit die Klasse `BinaryStringToNumber`.
4. Kopieren Sie das Beispielprogramm `Exercise3` in `Exercise4`. Behandeln Sie in einer Methode die Exception und lösen Sie sie danach erneut aus.
5. Kopieren Sie das Beispielprogramm `Exercise4` in `Exercise5`. Erstellen Sie in der `main`-Methode einen `try-finally`-Block, der zum Programmende einen Text ausgibt.

15

Nützliche Klassen und Packages

15.1 Zufallszahlen

Pseudozufallszahlen

In der Programmierung werden für Testzwecke, Simulationen oder in Spielen häufig Zufallszahlen eingesetzt. Sie werden nach einem bestimmten Algorithmus (in Java nach dem Linear-Kongruenz-Algorithmus) berechnet.

In der Klasse `java.lang.Math` ist die Methode `random` implementiert, die als Ergebnis einen `double`-Wert zwischen `0.0` und `1.0` liefert. Die Klasse `java.util.Random` bietet wesentlich mehr Möglichkeiten. Die Klasse stellt zwei verschiedene Konstruktoren zur Verfügung, mit denen Sie ein Zufallszahlengenerator-Objekt erzeugen können:

<code>Random()</code>	Nutzen Sie den parameterlosen Konstruktor, wird der Zufallszahlengenerator auf der Basis der aktuellen Systemzeit (in Millisekunden) initialisiert.
<code>Random(long seed)</code>	Wenn Sie den <code>seed</code> -Wert (engl. für Samen) als Parameter angeben, wird dieser als Initialwert verwendet und es wird unter gleichen Bedingungen immer wieder die gleiche „Pseudo“-Zufallszahlenreihe erzeugt.

In der folgenden Übersicht werden einige wichtige Methoden dieser Klasse vorgestellt:

<code>setSeed(long seed)</code>	Die Methode ändert den <code>seed</code> -Wert des Generator-Objekts auf den als Parameter übergebenen Wert.
<code>boolean nextBoolean()</code> <code>double nextDouble()</code> <code>float nextFloat()</code> <code>long nextLong()</code> <code>int nextInt()</code>	Diese Methoden liefern nahezu gleichmäßig verteilte Zufallszahlen des entsprechenden Typs.
<code>int nextInt(int n)</code>	Der Methode <code>nextInt</code> können Sie einen Parameter <code>n</code> vom Typ <code>int</code> übergeben. Die Methode liefert eine Zufallszahl im Bereich zwischen <code>0</code> und <code>n</code> .

<code>nextBytes(byte[] bytes)</code>	Mit der Methode <code>nextBytes</code> füllen Sie ein <code>byte</code> -Array mit Zufallswerten. Als Parameter muss das Array übergeben werden. Die Anzahl der erzeugten <code>byte</code> -Werte entspricht der Anzahl der Arrays-Elemente (das Array wird komplett gefüllt).
<code>double nextGaussian()</code>	Mithilfe der Methode <code>nextGaussian</code> können Sie eine Reihe von Zufallszahlen generieren, die einer gaußschen Normalverteilung unterliegen. Dabei wird von einem Mittelwert von 0.0 und einer Standardabweichung von 1.0 ausgegangen.

Beispiel: `com\herdt\java9\kap15\Gauss.java`

Im folgenden Beispiel soll eine Messreihe simuliert werden.

- ✓ Es werden 10000 Zufallszahlen nach der gaußschen Normalverteilung erzeugt.
- ✓ Für diese Werte werden das arithmetische Mittel und die Standardabweichung berechnet.
- ✓ Initialisieren Sie den Zufallszahlengenerator mit verschiedenen Werten (②) und werten Sie die Ergebnisse bei wiederholter Programmausführung aus.

```

public static void main(String[] args)
{
    ① int n = 10000;
    ② Random rd = new Random();
    ③ double[] x = new double[n];
    ④ double mittel = 0.0;
    double streuung = 0.0;
    for (int i = 0; i < n; i++)
    {
        ⑤ x[i] = rd.nextGaussian();
        ⑥ mittel += x[i];
    }
    ⑦ mittel /= n;
    for (double g : x)
    ⑧ streuung += (g - mittel) * (g - mittel);
    ⑨ streuung = Math.sqrt(streuung / (n - 1));
    ⑩ System.out.println("arithmetischer Mittelwert: " + mittel);
    System.out.println("Streuung: " + streuung);
}

```

- ① Die Anzahl der Messwerte wird mit 10000 Werten festgelegt.
- ② Es wird ein Zufallszahlengenerator-Objekt ohne `Seed`-Wert erzeugt.
- ③ Ein Array `x` soll der Abspeicherung der ermittelten Zufallszahlen dienen.
- ④ Die Variablen für die Berechnung von Mittelwert und Streuung werden mit 0 initialisiert.

- ⑤ Durch wiederholten Aufruf der Methode `nextGaussian` werden die Zufallszahlen erzeugt und im Array `x` gespeichert.
- ⑥ Für die Berechnung des Mittelwertes werden die Zufallszahlen aufsummiert.
- ⑦ Die Berechnung des Mittelwertes erfolgt.
- ⑧ Die Streuung (mittlere quadratische Abweichung) wird nach der Formel

$$\sqrt{\frac{1}{(n-1)} * \sum (x_i - \text{mittel})^2}$$

berechnet. In der `foreach`-Schleife wird die Summenberechnung durchgeführt. Der Rest der Berechnung erfolgt in Zeile ⑨.

- ⑩ Das berechnete Ergebnis wird ausgegeben.

15.2 Grundlagen zu Datum und Zeit

Entwicklung der Klassen zur Datums- und Zeitberechnung

Bereits seit der Version 1.0 ist die Klasse `Date` zur Darstellung und Berechnung von Datums- und Zeitwerten im JDK enthalten. Viele Methoden dieser Klasse sind als **deprecated** gekennzeichnet, d. h., sie sollten nicht mehr verwendet werden. Die Klasse wird daher in der aktuellen Version nur noch selten verwendet, beispielsweise wenn eine Methode den Datentyp `Date` fordert.

Die Nachfolgerklasse von `Date` heißt `Calendar` und ist seit der Version JDK 1.1 verfügbar. Von dieser wurde als einzige die Klasse `GregorianCalendar` abgeleitet, der beispielsweise auch der in Europa übliche gregorianische Kalender zugrunde liegt. Diese Klassen zur Datums- und Zeitberechnung gehören zum Package `java.util`.

Mit der Version 8 wurde in Java eine neue API `java.time` mit einer Vielzahl von Klassen für die Berechnung von Datumswerten und Zeitangaben eingeführt.

Die Klassen des Package `java.util` sind in Java 9 nicht als **deprecated** gekennzeichnet. Mit der Einführung der neuen API ist es möglich, dass dies in einer der kommenden Versionen erfolgt. Verwenden Sie deshalb möglichst bereits jetzt ausschließlich die neuen Klassen des Package `java.time`.

Merkmale und Vorteile der Klassen der neuen Datums- und Zeit API

Die neue API zeichnet sich unter anderem durch folgende Merkmale aus:

- ✓ Sie unterstützt den international genutzten Zeitstandard ISO 8601.
- ✓ International häufig verwendete Kalender, wie beispielsweise der japanische, der aktuell in China verwendete (Minguo), der islamische (Hijriah) sowie der buddhistische werden unterstützt.
- ✓ Alle Klassen dienen einem speziellen Zweck, ohne dabei Seitenwirkung auf andere Datums- und Zeitobjekte zu haben.
- ✓ Die API kann mit weiteren Klassen, beispielsweise für spezielle Kalender, erweitert werden.
- ✓ Alle `java.time` Objekte sind immutable.

Objekte der bis Java 8 zur Verfügung stehenden Klassen waren mutable, durch Berechnung änderten sich also die Zeitwerte. Durch die Verwendung von immutablen Objekten seit Java 8 erzeugt jede Berechnung einen neuen Wert. Damit ist es möglich, einen Wert mit einem parallel laufenden Task oder einer Bibliothek zu teilen, ohne dass das Risiko besteht, dass sich der Wert unerwartet ändert.

Die Umsetzung der Zeit in den einzelnen Klassen erfolgt nach diesen Regeln:

- ✓ Ein Tag hat 86.400 Sekunden (Schaltsekunden werden nicht berücksichtigt).
- ✓ Die Genauigkeit der Berechnung erfolgt in Nanosekunden.
- ✓ Um Mitternacht entspricht die in Java berechnete Zeit exakt der öffentlichen Zeit.
- ✓ Zu allen anderen Zeitpunkten entspricht die Java Zeit so genau wie möglich der offiziellen Zeit, zur Berechnung wird ein exakt definierter Algorithmus verwendet.

Das Package `java.time` umfasst z. B. Klassen für

- ✓ die Berechnung von Zeitpunkten und die Differenz zwischen zwei Zeitpunkten (Klassen `Instant`, `Duration`),
- ✓ die Arbeit mit Datumsangaben mit (`ZonedDateTime`) und ohne (`LocalDate`) Zeitzoneverwendung,
- ✓ die Beschreibung von Teilen eines Datums (z. B. `YearMonth`),
- ✓ typische Kalendermanipulationen (`TemporalAdjuster`),
- ✓ Berechnungen mit Zeiten (`LocalTime`),
- ✓ die Formatierung und Darstellung von Datum- und Zeit (`DateTimeFormatter`).

Die folgenden Abschnitte stellen einige wichtige Klassen vor.

15.3 Zeitpunkte – Klassen `Instant` und `Duration`

Als Grundlage für die Arbeit mit Zeitpunkten wird eine Zeitreihe verwendet, welche auf Nanosekunden-Basis berechnet wird. Diese ist durch drei Konstanten gekennzeichnet:

Konstante (<code>static final</code>)	Erläuterung	Wert
EPOCH	willkürlich gewählter Ausgangswert (Mitternacht des 01. Januar 1970 am 0-Meridian in Greenwich)	1970-01-01T00:00:00Z
MIN(<code>int</code>)	der kleinste unterstützte Wert (ca. eine Milliarde Jahre zurück)	'-1000000000-01-01T00:00Z'
MAX(<code>int</code>)	der größte unterstützte Wert (31. Dezember des Jahres 1.000.000.000)	'1000000000-12-31T23:59:59.999999999Z'

Die Dauer zwischen zwei Zeitpunkten wird mit einem Objekt der Klasse `Duration` berechnet.

Die Methode `now()` liefert den aktuellen Zeitpunkt. Daneben existiert eine Vielzahl von Methoden, u. a. zur Ausführung von arithmetischen Berechnungen.

Beispiel für die Verwendung der Klassen `Instant` und `Duration`: `com\herdt\java9\kap15\TimeDifference.java`

In diesem Beispiel wird die Zeitdauer für eine Berechnung gemessen.

```
package com.herdt.java9.kap15;
① import java.time.*;
class TimeDifference
{
    ② static void runAdd()
    {
        int i = 0;
        while (i < 1_000_000_000)
            i++;
    }
    public static void main(String[] args)
    {
        ③ Instant start = Instant.now();
        ④ runAdd();
        ⑤ Instant end = Instant.now();
        ⑥ Duration timeElapsed = Duration.between(start, end);
        ⑦ long millis = timeElapsed.toMillis();
        System.out.println
            ("Zeitdauer in Millisekunden: " + millis);
    }
}
```

- ① Das Package `java.util` wird importiert.
- ② Die Methode `runAdd` führt eine begrenzte Anzahl von Additionen durch.
- ③ Für den Beginn der Zeitdauer wird ein Zeitpunktobjekt der Klasse `Instant` erstellt.
- ④ Die Berechnung wird aufgerufen und ausgeführt.
- ⑤ Ein zweites Zeitpunktobjekt wird zum Setzen des Zeitpunktes des Abschlusses der Berechnungen erstellt.
- ⑥ Die Methode `between` der Klasse `Duration` ermittelt aus zwei übergebenen Zeitpunkten die Dauer.
- ⑦ Die ermittelte Zeitdauer wird in Millisekunden umgerechnet und anschließend ausgegeben.

15.4 Datumsangaben – Klassen `LocalDate`, `ZonedDateTime` und `Period`

Datumsangaben ohne und mit Zeitzone

Während Zeitpunkte der Klasse `Instant` die absolute Zeit darstellen, stehen Objekte der Klassen `LocalDate` und `ZonedDateTime` für Datumsangaben als Darstellungsweise der menschlichen Sicht auf die Tage. Erstere nutzen sie für ein einfaches Datum, beispielsweise einen Geburtstag, bei dem die Angabe einer Zeitzone keine Rolle spielt. Zeitzonenspezifische Angaben treten oft bei Anwendungen im internationalen Umfeld auf. Sollen beispielsweise die Termine von Besprechungen in einer weltweit genutzten Konzernanwendung verarbeitet werden, müssen diese die verschiedenen Zeitzonen berücksichtigen.

Datumsangaben ohne Zeitzone

Objekte der Klasse `LocalDate` können ein Datum innerhalb der Grenzen 01.01. des Jahres -999.999.999 (`LocalDate.MIN`) und 31.12. des Jahres 999.999.999 (`LocalDate.Max`) darstellen. Analog zur Klasse `Instant` nutzt `LocalDate` als Ausgangstag für Berechnungen den 01.01.1970, den `EPOCH_DAY`.

Zur Erstellung eines Objektes der Klasse `LocalDate` nutzen Sie die Methoden `now()` oder `of()`.

Die Methode	... liefert ein Objekt der Klasse <code>LocalDate</code> ...
<code>now()</code>	für das aktuelle Datum, basierend auf der Systemzeit und der Standardzeitzone.
<code>of</code> (<code>int year</code> , <code>int month</code> , <code>int dayOfMonth</code>)	entsprechend den für Jahr, Monat und Tag übergebenen Angaben; der Monat wird als Zahl angegeben.
<code>of</code> (<code>int year</code> , <code>Month month</code> , <code>int dayOfMonth</code>)	entsprechend den für Jahr, Monat und Tag übergebenen Angaben; der Monat wird als Element der Enumeration <code>Month</code> angegeben.
<code>ofEpochDay</code> (<code>long epochDay</code>)	für ein Datum, basierend auf dem Datum des <code>EPOCH_DAY</code> und der angegebenen Anzahl von Tagen (negativ oder positiv).
<code>ofYearDay</code> (<code>int year</code> , <code>int dayOfYear</code>)	für das angegebene Jahr und den festgelegten Tag des Jahres.

Weitere typische Methoden im Umgang mit der Klasse `LocalDate` sind:

Die Methode	liefert...
<code>getDayOfWeek()</code>	den Wochentag als Element der Enumeration <code>DayOfWeek</code> .
<code>getDayOfMonth()</code>	den Tag des Monats (zwischen 1 und 31).
<code>getDayOfYear()</code>	den Tag des Jahres (zwischen 1 und 366).
<code>getMonth()</code> <code>getMonthValue()</code>	den Monat des Jahres als Wert (zwischen 1 und 12) oder als Element der Enumeration <code>Month</code> .
<code>getYear()</code>	das Jahr.

Ein Vorteil der mit Java 8 eingeführten API für Datum und Zeit ist die Verwendung der Werte 1 bis 12 für die Monate. In der Vorgängerklass `GregorianCalendar` wurde dafür der Wertebereich 0 bis 11 verwendet. Die notwendigen Umrechnungen waren eine häufige Fehlerquelle.

Als Äquivalent zur Klasse `Duration`, welche bei der Ermittlung von Differenzen zwischen zwei Zeitpunkten genutzt wird, dient im Kontext von `LocalDate` die Klasse `Period`.

Die Klasse `LocalDateTime` verwaltet neben dem Datum auch die Uhrzeit, ebenfalls ohne Angabe einer Zeitzone.

Beispiel für die Verwendung der Klassen `LocalDate` und `Period`: *com\herdt\java9\kap15\SpecialDate.java*

Die Anwendung ermittelt die Anzahl der vergangenen Zeitperiode in Jahren, Monaten und Tagen seit dem Tag der offiziellen Ankündigung der Verfügbarkeit von Java (erfolgte am 23.05.1995).

```

package com.herdt.java9.kap15;
import java.time.*;

class SpecialDate
{
    public static void main(String[] args)
    {
        ① LocalDate today = LocalDate.now();
        ② LocalDate javaBirthday = LocalDate.of(1995, 5, 23);
        ③ Period liveTime = Period.between( javaBirthday, today);
        ④ System.out.println("Java wurde veröffentlicht vor: " +
            liveTime.getYears() + " Jahren, " +
            liveTime.getMonths() + " Monaten und " +
            liveTime.getDays() + " Tagen." );
    }
}

```

- ① Mit der Methode `now` der Klasse `LocalDate` wird das aktuelle Datum ermittelt.
- ② Ein zweites Objekt der Klasse `LocalDate` wird auf Basis des Datums '23.5.1995' erstellt.

- ③ Die Methode `between` der Klasse `Period` ermittelt aus zwei übergebenen Daten die Periode.
- ④ Die Werte der ermittelten Periode werden getrennt nach Jahren, Monaten und Tagen auf der Konsole ausgegeben.

Datumsangaben mit Zeitzone

Da es weltweit viele Zeitzonen gibt, reicht oft die Verwendung eines Objekts der Klasse `LocalDate` im Programm nicht aus. In diesem Fall nutzen Sie die Klasse `ZonedDateTime`, welche Datums- und Zeitwerte unter der Berücksichtigung einer Zeitzone verwalten kann. Auch für die in vielen Ländern genutzte Sommerzeit können Sie diese Klasse einsetzen.

Die Datenbank der Internet Assigned Authority (IANA) listet die Zeitzonen auf, siehe <https://www.iana.org/time-zones>.

Java nutzt die Datenbank der IANA, welche mehrfach im Jahr aktualisiert wird. Jede Zeitzone besitzt eine ID. Aktuell gibt es nahezu 600 Zeitzonen-IDs.

Ein Beispiel für die Anwendung einer Zonen-ID finden Sie hier:

`com\herdt\java9\kap15\DateTimeZone.java`.

Zur Erstellung eines Objektes der Klasse `ZonedDateTime` können Sie ...

ein vorhandenes <code>LocalDateTime</code> -Objekt unter Angabe einer Zonen-ID konvertieren	<pre>LocalDateTime now = LocalDateTime.now(); ZoneId zone = ZoneId.of("Europe/Berlin"); ZonedDateTime zdt = now.atZone(zone);</pre>
es über die statische Methode <code>of</code> der Klasse <code>ZonedDateTime</code> erstellen	<pre>ZonedDateTime zdt2 = ZonedDateTime.of(1995, 5, 23, 15, 0, 0, 0, ZoneId.of("America/New_York"));</pre>

Die Klassen `ZonedDateTime` und `LocalDateTime` besitzen viele identische Methoden, z. B.:

Methoden	
<code>plusDays()</code> , <code>plusWeeks()</code> , <code>plusMonth()</code> , <code>plusYears()</code> , <code>plusHours()</code> , <code>plusMinutes()</code> , <code>plusSeconds()</code> , <code>plusNanos()</code>	um eine Zeit-/Datumseinheit zu dem Objekt zu addieren.
<code>minusDays()</code> , <code>minusWeeks()</code> , <code>minusMonth()</code> , <code>minusYears()</code> , <code>minusHours()</code> , <code>minusMinutes()</code> , <code>minusSeconds()</code> , <code>minusNanos()</code>	um eine Zeit-/Datumseinheit von dem Objekt zu subtrahieren.
<code>plus()</code> , <code>minus()</code>	um eine Dauer (<code>Duration</code> oder <code>Period</code>) zu addieren/subtrahieren.
<code>withDayOfMonth()</code> , <code>withDayOfYear()</code> , <code>withMonth()</code> , <code>withYear()</code> , <code>withHour()</code> , <code>withMinute()</code> , <code>withSecond()</code> , <code>withNano()</code>	um ein neues Objekt zu liefern, welches einen bestimmten Wert für eine Datums-/Zeitwerteinheit besitzt.

Ermitteln eines speziellen Datums – die Klasse **TemporalAdjusters**

Häufig werden spezielle Datumsangaben benötigt, welche in einem bestimmten logischen Zusammenhang zu einer Woche oder zu einem Monat stehen, beispielsweise „der erste Dienstag jeden Monats“ oder „der letzte Werktag in einem Monat“. Die Klasse `TemporalAdjusters` besitzt einige statische Methoden, die Ihnen bei derartigen Berechnungen helfen.

Die Methode	liefert ...
<code>next(wochentag)</code> , <code>previous(wochentag)</code>	das nächste oder vorige Datum, das auf einen bestimmten Wochentag fällt.
<code>nextOrSame(wochentag)</code> <code>previousOrSame(wochentag)</code>	das nächste oder vorige Datum, das auf einen bestimmten Wochentag fällt (Ausgangspunkt: aktuelles Datum).
<code>dayOfWeekInMonth</code> (<code>n</code> , <code>wochentag</code>)	den <code>n</code> .ten Wochentag in einem Monat.
<code>lastInMonth(wochentag)</code>	den letzten Wochentag im Monat.
<code>firstDayInMonth()</code> <code>firstDayOfNextMonth()</code> <code>firstDayOfNextYear()</code> <code>lastDayOfMonth()</code> <code>lastDayOfPreviousMonth()</code> <code>lastDayOfYear()</code>	den Tag entsprechend dem Namen der Methode.

Das Ergebnis der Methoden wird der Methode `with()` eines `LocalDate`-Objektes übergeben.

Beispiel `com\herdt\java9\kap15\DateAdjuster.java`

Im Beispiel wird das Datum des ersten Montags nach dem 1.1.2015 ermittelt.

```

package com.herdt.java9.kap15;
import java.time.*;
① import java.time.temporal.*;

class DateAdjuster
{
    public static void main(String[] args)
    {
        ② LocalDate firstMonday = LocalDate.of(2017, 9, 1). with(
            TemporalAdjusters.nextOrSame(DayOfWeek.MONDAY));
        System.out.println(firstMonday.toString());
    }
}

```

- ① Zusätzlich zum Package `java.time` muss das Package `java.time.temporal` importiert werden.

- ② Die Angabe des Ausgangsdatums. Der Methode `with` wird als Parameter die Versetzungsmethode `nextOrSame` mit dem Element `MONDAY` der Enumeration `DayOfWeek` übergeben.

15.5 Zeiten – die Klasse `LocalTime`

Für die Verarbeitung von Zeiten stellt die API die Klasse `LocalTime` bereit.

Bei Anwendungen, die verschiedene Zeitzonen oder die Sommerzeit berücksichtigen müssen, sollten Sie die Klasse `ZonedDateTime` verwenden.

Ein Objekt der Klasse `LocalTime` erstellen Sie mit einer der Methoden `now` bzw. `of`. Beide Methoden sind überladen und ermöglichen damit die Angabe verschiedener Parameter bei der Objekterstellung:

Methode	erstellt ein <code>LocalTime</code> -Objekt basierend auf
<code>now()</code>	der Systemzeit und der Standardzeitzone
<code>now(Clock clock)</code>	einem speziellen Objekt der Klasse <code>Clock</code>
<code>now(ZoneId zone)</code>	der Systemzeit und der per ID angegebenen Zeitzone
<code>of(int hour, int minute)</code>	den angegebenen Werten für Stunde und Minute
<code>of(int hour, int minute, int second)</code>	den angegebenen Werten für Stunde, Minute und Sekunde
<code>of(int hour, int minute, int second, int nanoOfSecond)</code>	den angegebenen Werten für Stunde, Minute, Sekunde und Nanosekunde

Wichtige Methoden zur Manipulation der Zeit im Programm sind:

Methode	Funktion
<code>plusHours()</code> , <code>plusMinutes()</code> , <code>plusSeconds()</code> , <code>plusNanos()</code>	fügt dem jeweiligen Zeitelement den übergebenen Wert hinzu
<code>minusHours()</code> , <code>minusMinutes()</code> , <code>minusSeconds()</code> , <code>minusNanos()</code>	subtrahiert von dem jeweiligen Zeitelement den übergebenen Wert
<code>plus()</code> , <code>minus()</code>	verändert die Zeitangabe um eine Dauer
<code>withHour()</code> , <code>withMinute()</code> , <code>withSecond()</code> , <code>withNano()</code>	liefert ein neues <code>LocalTime</code> -Objekt unter Berücksichtigung des gesetzten Wertes
<code>getHour()</code> , <code>getMinute()</code> , <code>getSecond()</code> , <code>getNano()</code>	liefert den Wert für das jeweilige Zeitelement des <code>LocalTime</code> -Objektes
<code>toSecondOfDay()</code> , <code>toNanoOfDay()</code>	gibt die Anzahl an Sekunden/Nanosekunden seit Mitternacht für das <code>LocalTime</code> -Objekt an
<code>isBefore()</code> , <code>isAfter()</code>	vergleicht zwei <code>LocalTime</code> -Objekte

Beispiel `com\herdt\java9\kap15\Times.java`

Im Beispiel werden einige Methoden zur Manipulation der Zeit verwendet.

```
package com.herdt.java9.kap15;
import java.time.*;
① import java.time.temporal.*;
class Times
{
    - public static void main(String[] args)
    {
②      LocalDateTime rightNow = LocalDateTime.now();
③      LocalDateTime bedTime = LocalDateTime.of(23, 00);
④      LocalDateTime standUp = bedTime.plusHours(8);
⑤      if (rightNow.isAfter(bedTime) &&
          rightNow.isBefore(standUp))
          System.out.println("Es ist Schlafenszeit...");
      else
      {
          long hoursUntilBed =
              rightNow.until(bedTime, ChronoUnit.HOURS);
          System.out.println("Es sind noch ca. " + hoursUntilBed +
              " Stunden bis zur Schlafenszeit.");
      }
    }
}
```

- ① Um die Enumeration `ChronoUnit` verwenden zu können, wird das Package `java.time.temporal` importiert.
- ② Ein `LocalTime`-Objekt wird auf Basis der aktuellen Systemzeit erstellt.
- ③ Erstellung eines weiteren `LocalTime`-Objekts auf Basis der Zeitangabe 23:00 Uhr.
- ④ Das dritte `LocalTime`-Objekt wird durch das Hinzufügen von 8 Stunden zum zweiten Objekt erstellt. Die 24-hr-Grenze um Mitternacht wird dabei vom System berücksichtigt.
- ⑤ Vergleich zweier `LocalTime`-Objekte mit der Methode `isBefore`. Abhängig von der aktuellen Uhrzeit wird ein Hinweistext ausgegeben oder die Methode `until` unter Verwendung des Elements `HOURS` der Enumeration `ChronoUnit` verwendet, um eine Zeitdifferenz zu bestimmen.

Die Darstellung der Zeitwerte im 12- (AM/PM) oder 24-Stunden-Format erfolgt durch die Formatierungsklasse `DateTimeFormatter`.

15.6 Datums- und Zeitangaben formatiert ausgeben – die Klasse `DateTimeFormatter`

Mithilfe der Klasse `DateTimeFormatter` bestehen drei Möglichkeiten, einen Datums- bzw. Zeitwert darzustellen:

- ✓ mithilfe eines vordefinierten Standardformats
- ✓ in einem lokalspezifischen Format
- ✓ unter Verwendung eines selbst definierten Musters

Vordefinierte Standardformate dienen in erster Linie der Darstellung in maschinenlesbarer Form, beispielsweise für Zeitstempel bei der Protokollierung. Sie werden über die Methode `format` des jeweiligen Formates verwendet:

```
formatted = DateTimeFormatter.ISO_DATE_TIME.format(nowComplete);
```

Die für Menschen gebräuchliche Darstellung erfolgt über die lokalspezifischen Formatierer. Für Datum und Zeit gibt es jeweils 4 Darstellungsformen:

Stil	Datum	Zeit
SHORT	13.5.17	10:41
MEDIUM	13.05.2017	10:41:31
LONG	13. Mai. 2017	10:41:31 MESZ
FULL	Dienstag, 13. Mai 2017	10:41 Uhr MESZ

Zur Erstellung eines lokalspezifischen Formatierers nutzen Sie eine der Methoden `ofLocalizedDate`, `ofLocalizedTime` bzw. `ofLocalizedDateTime` unter Angabe des Formatierungsstils:

```
formatter =
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL);
formatted = formatter.format(nowComplete);
```

Eigene Darstellungsformen können durch die Angabe eines Musters bestimmt werden:

```
formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");
```

Die obigen drei Formatierungen erzeugen folgende Ausgabe auf der Konsole:

```
2017-05-13T10:41:31.78+02:00 [Europe/Berlin]
Dienstag, 13. Mai 2017 10:41 Uhr MESZ
Di 2017-05-13 10:41
```

Formatierte Ausgabe eines `ZonedDateTime`-Objekts

Das komplette Beispielprogramm finden Sie unter `com\herdt\java8\kap15\FormattedTimes.java`.

15.7 Die Klasse `System`

Die Klasse `System` ist im Package `java.lang` enthalten. Sie stellt nützliche Systeminformationen und Methoden zur Verfügung.

Systeminformationen ermitteln

Aufgrund seiner Plattformunabhängigkeit bietet Java keine direkte Möglichkeit, auf Umgebungsvariablen des Betriebssystems zuzugreifen. Dafür wird aber eine `Properties`-Liste zur Verfügung gestellt, in der Informationen zum Java-Laufzeitsystem enthalten sind. `Properties`-Listen sind von der Klasse `HashTable` abgeleitet. In Form einer Tabelle sind die Systemeigenschaften und ihre Werte als Schlüssel-Werte-Paare abgespeichert.

- ✓ Die Klasse `System` verfügt über die Methode `getProperties`, die ein `Properties`-Objekt liefert.

`Properties getProperties()`
- ✓ Den jeweiligen Wert für eine Eigenschaft erhalten Sie über die Methode `getProperty`, die mit zwei unterschiedlichen Parameterlisten aufgerufen werden kann.

`String getProperty(String key)`
`String getProperty(String key, String def)`
- ✓ Beide Methoden liefern den Wert zu der übergebenen Eigenschaft (`key`) als `String` zurück. Ist diese Eigenschaft nicht gesetzt, liefert die erste Variante `null` zurück. Die zweite Variante gibt den als zweiten Parameter übergebenen `default`-Wert zurück. In der folgenden Tabelle sind die Eigenschaften aufgelistet, die immer verfügbar sind.

<code>java.home</code>	Installationsverzeichnis
<code>java.version</code>	Version der Laufzeitumgebung
<code>java.vendor</code>	Hersteller (z. B. Sun Microsystems Inc.)
<code>java.vendor.url</code>	URL des Herstellers
<code>java.class.path</code>	Klassenpfad
<code>java.class.version</code>	Versionsnummer der Klasse
<code>java.specification.name</code>	Name der Spezifikation der Laufzeitumgebung
<code>java.specification.version</code>	Version der Spezifikation der Laufzeitumgebung
<code>java.specification.vendor</code>	Hersteller der Spezifikation der Laufzeitumgebung
<code>java.vm.name</code>	Name der implementierten Java Virtual Machine (VM)
<code>java.vm.version</code>	Version der Java VM
<code>java.vm.vendor</code>	Hersteller der Java VM
<code>java.vm.specification.name</code>	
<code>java.vm.specification.version</code>	Name der Spezifikation der Java VM

<code>java.vm.specification.vendor</code>	Version der Spezifikation der Java VM
<code>line.separator</code>	Hersteller der Spezifikation der Java VM
<code>os.arch</code>	Zeilenendekennzeichen (meist \n)
<code>os.name</code>	Betriebssystemarchitektur
<code>os.version</code>	Betriebssystemname
<code>path.separator</code>	Trennzeichen mehrerer Pfadangaben, z. B. das Semikolon in C:\;C:\Windows;C:\java\ C:\Programme\java\jdk-9
<code>file.separator</code>	Trennzeichen zwischen Verzeichnis und Unterverzeichnis bzw. Dateiname
<code>user.dir</code>	Arbeitsverzeichnis des angemeldeten Nutzers
<code>user.home</code>	Home-Verzeichnis
<code>user.name</code>	Angemeldeter Nutzer

Über ein Enumeration-Objekt können Sie diese Properties-Liste sequenziell durchlaufen und so auf alle Eigenschaften zugreifen.

Beispiel: `com\herdt\java9\kap15\SystemInfo.java`

Die Methode `printSysInfo` zeigt alle verfügbaren Eigenschaften des Systems an:

```

static void printSysInfo()
{
① Enumeration props = System.getProperties().propertyNames();
  System.out.println("Systemeigenschaften:\n");
② while (props.hasMoreElements())
  {
③   String prop = props.nextElement().toString();
④   System.out.print( prop + ":" );
⑤   for (int i = prop.length(); i < 30; i++)
     System.out.print(" ");
⑥   System.out.println(System.getProperty(prop));
  }
}

```

- ① Es wird ein Enumeration-Objekt `props` erzeugt. Die Methode `System.getProperties` liefert ein `Properties`-Objekt und die Methode `propertyNames` eine Liste mit den Namen der Systemeigenschaften als Enumeration-Objekt.
- ② Die Aufzählung wird sequenziell durchlaufen. Die `while`-Schleife bricht ab, wenn kein weiteres Objekt vorhanden ist.

- ③ Die Methode `nextElement` liefert das nächste Objekt (den Namen der nächsten Eigenschaft) der Aufzählung. Dieses wird in einen String umgewandelt.
- ④ Der Name der Eigenschaft wird ausgegeben.
- ⑤ Zur übersichtlichen Ausgabe werden weitere Leerzeichen ausgegeben.
- ⑥ Den Wert zu der Eigenschaft erhalten Sie über die Methode `getProperty` der Klasse `System`. Der Name der Eigenschaft `prop` wird als Parameter übergeben.

Beispiel: `com\herdt\java9\kap15\SystemInfoList.java`

Eine einfache Liste, in der die Schlüssel (Key) und mit einem Gleichheitszeichen getrennt die entsprechenden Werte aufgeführt werden, können Sie mit der Methode `list` der Klasse `Properties` erstellen:

```
static void printSysInfo()
{
    Properties props = System.getProperties();
    System.out.println("Systemeigenschaften:\n");
    props.list(System.out);
}
```

15.8 Weitere Methoden der Klasse `System`

Programmbeendigung

Die Methode `exit` beendet das laufende Programm und gibt den als Parameter übergebenen Wert an das aufrufende Programm (z. B. eine Programmierumgebung oder ein Betriebssystem) zurück. Den Statuswert können Sie selbst festlegen und so Informationen über den Grund des Programmabbruchs nach außen geben. Die Konvention legt fest, dass der Wert 0 bei normaler Beendigung des Programms zurückgegeben wird. Ansonsten übergeben Sie einen positiven Wert größer als 0.

```
System.exit(int status);
```

Im `catch`-Block einer `try-catch`-Anweisung kann die `exit`-Methode dafür sorgen, dass das Programm im Fehlerfall verlassen wird. Beachten Sie, dass in diesem Fall keine `finally`-Blöcke zur Ausführung kommen.

```
catch (XYException e)
{
    // Fehlerbehandlung
    System.exit(1);
}
```

Standerdeingabe und -ausgabe umleiten

Die Objekte `out`, `in` und `err` der Klasse `System` werden für die Ein- und Ausgabe auf den Standardgeräten verwendet. Mit den nebenstehenden Methoden lässt sich die Standerdeingabe und -ausgabe vom Programm aus umleiten.

```
setErr(PrintStream err)
setIn(InputStream in)
setOut(PrintStream out)
```

Sie können beispielsweise der Methode `setOut` einen `PrintStream` übergeben, der die Daten an einen `FileOutputStream` weiterleitet und somit die Standardausgabe in eine Datei umleitet.

Beispiel für die Umleitung der Ausgabe und die Verwendung der Methode `exit`: `com\herdt\java9\kap15\SystemInfoToFile.java`

In dem Beispiel wird die Ausgabe der Systemeigenschaften in die Datei `SystemTest.txt` umgeleitet. Falls die Datei nicht erstellt werden kann, wird eine Exception `FileNotFoundException` ausgelöst und abgefangen. Dabei wird das Programm über die Methode `System.exit` mit dem Rückgabewert 1 beendet.

```
public static void main(String[] args)
{
    try
    {
        System.setOut(new PrintStream(new
        FileOutputStream("SystemTest.txt")));
    }
    catch (FileNotFoundException fnfex)
    {
        System.out.println(fnfex);
        System.exit(1);
    }
    printSysInfo();
}
```

Feldinhalte kopieren

Die Methode `arraycopy` kopiert Arrays oder Teile davon in ein anderes Array. Die Elemente des Arrays können primitive Datentypen oder Objekte sein. Die ersten beiden Parameter sind der Name des Quell-Arrays (Source – `src`) und die Position, ab welcher kopiert werden soll. Der dritte und vierte Parameter legt das Ziel-Array (Destination – `dst`) und die Position fest, an die das erste Element kopiert werden soll. Die Anzahl der zu kopierenden Elemente wird durch den fünften Parameter angegeben. Beim Aufruf dieser Methode wird eine `IndexOutOfBoundsException` ausgelöst, wenn auf Elemente außerhalb der Array-Grenzen zugegriffen wird. Passen die Typen der Arrays nicht zusammen, wird eine Ausnahme vom Typ `ArrayStoreException` ausgelöst.

```
arraycopy(Object src, int src_position, Object dst,
int dst_position, int length)
```

Beispiel: `com\herdt\java9\kap15\FieldCopy.java`

In diesem Beispiel wird eine Benutzereingabe in zwei Teile aufgesplittet, indem diese durch die Methode `arraycopy` in zwei weitere Felder kopiert wird.

```
static void copyArray()
{
    ① byte array1[] = new byte[20];
    byte array2[] = new byte[10];
    byte array3[] = new byte[10];
    System.out.print("Geben Sie 20 Zeichen ein: ");
    try
    {
        ② System.in.read(array1);
        ③ System.arraycopy(array1, 0, array2, 0, 10);
        System.out.print("Die ersten 10 Zeichen sind: ");
        ④ System.out.write(array2);
        ⑤ System.arraycopy(array1, 10, array3, 0, 10);
        System.out.print("\nDie letzten 10 Zeichen sind: ");
        ④ System.out.write(array3);
    }
    catch (IOException io)
    {
        System.out.print(io.getMessage());
    }
}
```

- ① Es werden drei `byte`-Arrays definiert. Für das erste Array werden 20 Elemente vorgesehen, für die anderen beiden je 10 Elemente.
- ② Die Methode `read` liest maximal 20 Zeichen in das Array `array1` ein.
- ③ Die ersten 10 Zeichen des Arrays `array1` werden in das Array `array2` kopiert.
- ④ Der Methode `write` kann ein `byte`-Array übergeben werden, welches auf dem Bildschirm angezeigt wird.
- ⑤ 10 Zeichen ab Position 10 werden aus Array `array1` in das Array `array3` kopiert.

Eine Array-Variable ist eine Referenzvariable. Bei der Zuweisung von einem Array an ein zweites Array mit einer Wertzuweisung (=) werden nicht die Elemente der Arrays kopiert, sondern es wird nur die Adresse des Source-Arrays zugewiesen. Beide Referenzvariablen verweisen auf dasselbe Arrays-Objekt.

`array2 = array1;`

Garbage-Collector aufrufen

Während der gesamten Programmausführung läuft im Hintergrund der sogenannte Garbage-Collector, der alle nicht mehr referenzierten Objekte an das Laufzeitsystem zurückgibt. Da der Garbage-Collector mit niedriger Priorität läuft, ist der Zeitpunkt der Speicherfreigabe nicht voraussehbar. Über die folgende Methode können Sie den Zeitpunkt für den Aufruf des Garbage-Collectors selbst festlegen.

```
System.gc();
```

Beachten Sie, dass der Aufruf einige Zeit in Anspruch nehmen kann.

Zeitmessung

Um die Zeitdauer bestimmter Prozesse zu ermitteln, wird eine möglichst genaue Zeitdifferenz zwischen Beginn und Ende des Prozesses benötigt. Für diese Aufgabe können Sie zwei Methoden mit unterschiedlicher Genauigkeit verwenden:

Methode	Beschreibung
<code>long System.currentTimeMillis()</code>	Gibt die Zeit (in ms) zurück, die seit dem 1.1.1970 (00:00 Uhr) und dem Zeitpunkt des Methodenaufrufs vergangen ist
<code>long System.nanoTime()</code>	Gibt den Wert eines internen Zeitgebers (in ns) zurück. Die Methode kann nur zur Messung von Zeitdifferenzen auf einem System verwendet werden, da die Basis der Zeitmessung nicht festgelegt ist.

15.9 Die Klasse `Console`

Einfache Eingabe und Ausgabe über die Konsole

Die Klasse `Console` des Packages `java.io` ist speziell für die Ein- und Ausgabe von Textinformationen über die Konsole vorgesehen.

Mit der Methode `System.console` erzeugen Sie ein Objekt der Klasse `Console`, sofern eine Konsole existiert. Anderenfalls wird `null` zurückgegeben.

Auswahl der Methoden der Klasse **Console**



Die folgende Tabelle zeigt einige Methoden der Klasse `Console` im Überblick:

<code>Console printf(String fmt, Object... args)</code>	Gibt wie bei der Methode <code>System.out.printf</code> die Daten formatiert aus. Dabei enthält der <code>String fmt</code> sowohl feststehende Texte und Formatierungszeichen als auch Platzhalter für die nachfolgend als Argumente (<code>object...</code>) übergebenen Werte.
<code>String readLine()</code>	Liest eine einzelne Eingabezeile als <code>String</code> ein
<code>String readLine(String fmt, Object... args)</code>	Liest ebenfalls eine einzelne Eingabezeile als <code>String</code> ein. Beispielsweise als Aufforderung zur Eingabe können Sie entsprechend der Methode <code>printf</code> als Argumente einen Formatstring und gegebenenfalls Werte übergeben.
<code>String readPassword()</code>	Liest eine einzelne Eingabezeile verdeckt als <code>String</code> ein. D. h., bei der Eingabe werden die eingegebenen Zeichen nicht auf dem Bildschirm angezeigt. Dies wird beispielsweise bei der Abfrage eines Kennworts verwendet.
<code>String readPassword(String fmt, Object... args)</code>	Auch bei der Methode <code>readPassword</code> können Sie beispielsweise als Aufforderung zur Eingabe einen entsprechenden formatierten Text ausgeben.

Beispiel zur Verwendung der Klasse **Console**:
com\herdt\java9\kap15\UseConsole.java


<pre> package com.herdt.java9.kap15; import java.util.*; import java.io.*; class UseConsole { public static void main(String[] args) { ① Console con = System.console(); ② if (con != null) { ③ con.printf("Login:%n"); ④ String s = con.readLine("Geben Sie bitte Ihren Namen ein: "); con.printf("Guten Tag %s,%n", s); ⑤ char[] pwchars = con.readPassword ("geben Sie bitte Ihr Passwort ein: "); </pre>	
---	--

```
⑥      String pw = new String(pwchars);  
        con.printf("Passwortprüfung... '%s'", pw);  
    }  
    else  
⑦      System.out.println("Keine Console vorhanden");  
    }  
}
```

- ① Mit der Methode `console` wird – sofern die Anwendung über die Konsole ausgeführt wird – ein Objekt dieser Konsole erzeugt.
- ② Es wird geprüft, ob ein Objekt erzeugt wurde (anderenfalls enthält die Variable `con` den Wert `null`).
- ③ Mit der Methode `printf` können Sie entsprechend der Methode `System.out.printf` Text formatiert auf der Konsole ausgeben.
- ④ Alle Zeichen, die bis zum Betätigen der -Taste eingegeben werden, lassen sich mit der Methode `readLine` einlesen und in einem `String` speichern. Ein entsprechender Text, der zur Eingabe auffordert, wird gegebenenfalls als Argument übergeben.
- ⑤ Mit der Methode `readPassword` werden ebenfalls alle Zeichen bis zum Betätigen der -Taste eingelesen. Die Zeichen werden bei der Eingabe jedoch nicht angezeigt. Die Methode liefert ein `char`-Array zurück.
- ⑥ Das `char`-Array wird in einen `String` konvertiert.
- ⑦ Sofern das Programm nicht über die Konsole ausgeführt wird, wird ein entsprechender Hinweis ausgegeben.

15.10 Übungen

Übung 1: Nützliche Klassen verwenden 1

Level		Zeit	ca. 30 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Arbeiten mit Zufallszahlen und Arrays ✓ Verwenden der Klassen zur Berechnung und Anzeige von Kalenderdaten 		
Übungsdatei	--		
Ergebnisdateien	<i>Exercise1.java, Exercise2.java, Exercise3.java, measurements.txt</i>		

1. Für die Berechnung von Tages-Durchschnittstemperaturen in einer chemischen Versuchsanlage sind für 14 Tage je 10 Messwerte zu erfassen und in einem zweidimensionalen Array zu speichern. Die zu ermittelnden Werte sollen für die Simulation als Zufallszahlen generiert werden und zwischen 20 und 35 (°C) liegen. Die Temperaturwerte für einen Tag und die errechnete Durchschnittstemperatur sind in je einer Zeile auszugeben. Am Schluss ist der Gesamtdurchschnitt der Temperatur in den 14 Tagen anzuzeigen.

Temperaturen in Grad Celsius

22 32 33 22 34 20 Durchschnitts-Temperatur: 27

21 20 32 23 32 22 Durchschnitts-Temperatur: 25

...

26 34 34 30 23 33 Durchschnitts-Temperatur: 30

Gesamt-Durchschnitts-Temperatur: 26

2. Erstellen Sie in diesem Übungsteil ein Programm, das den Kalender für einen Monat anzeigt. Der Monat und das Jahr sind als Parameter dem Programm zu übergeben. Die Ausgabe soll den folgenden Aufbau besitzen:

Kalender für Oktober 2017

Wo Mo Di Mi Do Fr Sa So

39 1

40 2 3 4 5 6 7 8

41 9 10 11 12 13 14 15

42 16 17 18 19 20 21 22

43 23 24 25 26 27 28 29

44 30 31

3. Ermitteln Sie für die Ausgabe den Monatsnamen und das Jahr und geben Sie beides aus. Danach folgt die Ausgabe der Kopfzeile. Die erste Spalte enthält die Kalenderwoche, die folgenden Spalten enthalten die Wochentage. Achten Sie bei der Ausgabe darauf, dass der erste Tag dem richtigen Wochentag zugeordnet wird.

Für die Einstellung des Abstands zwischen den Tagen kann die Escape-Sequenz `\t` für den Tabulator verwendet werden.

Für den ersten Parameter (Monat) kann eine Sicherheitsabfrage eingebaut werden, sodass nur die Werte von 1 bis 12 angenommen werden.

Hinweis zum Wochentag: Für die Berechnung des Wochentags können Sie ein Objekt der Klasse `WeekFields` verwenden. Die Woche liefert deren Methode `weekOfYear()`.

Die Erstellung des Objekts erfolgt über die Methode `of()`, beispielsweise so:


```
WeekFields week = WeekFields.of(Locale.getDefault());
```

Damit Sie `Locale` nutzen können, importieren Sie das Package `import java.util.*`.

4. Erweitern Sie das Programm aus Übungsteil ①, indem Sie die Ausgabe in eine Datei mit dem Namen *measurements.txt* umleiten. Speichern Sie die Angaben des aktuellen Datums, der aktuellen Uhrzeit und des verwendeten Betriebssystems ebenfalls in der Datei *measurements.txt*.

```
...
25 32 25 33 23 21 23 33 31 25 Durchschnitts-
Temperatur: 27
Gesamt-Durchschnitts-Temperatur: 26
Erstellt am 13.05.2014 um 11:11:34
unter Windows 8.1 Version 6.3
```

Übung 2: Nützliche Klassen verwenden 2

Level		Zeit	ca. 15 min
Übungsinhalte	<ul style="list-style-type: none"> ✓ Zeitoperationen ✓ Manipulation von Zeichenketten mit der Klasse <code>StringBuffer</code> 		
Übungsdatei	--		
Ergebnisdatei	<i>TimeMeasurement.java</i>		

1. Erstellen Sie einen String `s1` mit der Zeichenkette "X" und einen `StringBuffer`.
2. Messen Sie die Zeit, die bei der Programmausführung benötigt wird, um 10000-mal einen String "X" an den String `s1` anzuhängen.
3. Geben Sie das Ergebnis der Zeitmessung aus.
4. Messen Sie die Zeit, die bei der Programmausführung benötigt wird, um den `StringBuffer` 10000-mal um einen String "X" zu erweitern.
5. Geben Sie auch das Ergebnis dieser Zeitmessung aus.



Beachten Sie, dass das Programm je nach Hardwareausstattung mehrere Minuten benötigt. Verringern oder erhöhen Sie gegebenenfalls die Anzahl der Durchläufe.

Symbole

!	39
!=	38
&&	39
*.class-Datei	15
*.java-Datei	11, 15
...	164
{	15
	39
<	38
<=	38
<>	172
==	38, 147
>	38
>=	38

A

abstract	107
AbstractList	175
Abstrakte Klassen	106, 107, 132
Abstrakte Methoden	107
Accessoren	80
Adapterklassen	134
Adapterklassen anwenden	135
add()	173, 175, 177
Alphanumerischer Datentyp	28
Android	10
Anonyme Arrays	164
Anweisungen	14, 25
append()	150
Archive	120
Arithmetische Operatoren	34
arraycopy()	158, 214
ArrayIndexOutOfBoundsException	156, 188
ArrayList	171, 175, 176
Array-Literale	155
Arrays	162
Arrays schachteln	160, 161
Arrays, anonyme	164
Arrays, Felder	154
Arrays, mehrdimensionale	160
ArrayStoreException	214
Assoziativität, Operatoren	34, 37
Attribute	14, 62, 64, 67, 79
Attribute definieren	64
Aufzählungstypen	165
Aufzählungstypen definieren	166
Ausdruck, Expression	30, 34
Ausnahmebehandlung	182
Ausnahmen	183
Auswahl, mehrseitige	50, 51
Autoboxing	152

B

Basisklasse	89, 93, 102
Basisklasse, Methoden	96
Bedingungen, Kontrollstrukturen	45
Bezeichner	21, 64
Beziehung, 'has a'	98
Beziehung, 'is a'	98
Bindung, Operatoren	34
Boolean	13
boolean	27
Boolescher Datentyp	27, 28
Boxing	152
break	51, 59, 193
byte	26

C

Calendar	201
capacity()	150
case	51
catch	182, 185, 186, 189, 193
char	27
charAt()	147, 150
class	63
ClassNotFoundException	191
CLASSPATH	110, 112
clear()	174
clone()	103, 104
Collections	171, 172
compareTo()	146
Compiler	11, 15
Console	42, 216
contains()	146, 174
continue	59, 193
copyOf()	158
copyOfRange()	158
currentTimeMillis()	216

D

Date	201
Daten formatiert ausgeben	40
Datentypen überprüfen	133
Datentypen, alphanumerische	27, 28
Datentypen, boolesche	27, 28
Datentypen, generische	173
Datentypen, Gleitkomma-	26
Datentypen, Integer-	26
Datentypen, numerische	26, 27
Datentypen, primitive	25
Datum, Darstellung und Berechnung	201
default	51, 129
Default-Zugriffsrecht, Package-	113

Dekrement	36
Deque	172
Dokumentation, Quelltexte	24
Doppelt verkettete Listen	176
double	26
do-while	46, 55, 59
Duration	202

E

Eclipse	13
Editionen, Java	9
Editoren, Übersicht	13
Eigenschaften von Objekten	67
Eingabe, verdeckte	217
else	48
Endlosschleife	54
endsWith()	147
Entwicklungsumgebungen	12
enum	165
Enumeration	165, 212
Enumeration definieren	166
Enums	165
equals()	103, 104, 145, 174
equalsIgnoreCase()	146
Error	183, 184
Exception	183, 185
Exceptionhandling	182
Exceptions	182
Exceptions abfangen	185
Exceptions auffangen	182
Exceptions auslösen	182, 194, 195
Exceptions behandeln	185
Exceptions weitergeben	190, 191
Exceptions, (un)kontrollierte	184
Exceptions, eigene erzeugen	195
Exceptions, Klassifizierung	182
exit()	213
exports	124
exports to	124
Expression, Ausdruck	30
extends	90, 130

F

false	27
Fehler	183
Fehlermeldungen	18
Felder, Arrays	154
FileNotFoundException	214
fill()	162
final	33, 105, 107
Finale Klassen	105
finally	193
float	26

for 46, 56, 59, 160, 179, 201
 foreach-Schleifen 160, 179, 201
 Funktionalität, Klassen 62, 69, 79
 Fußgesteuerte Schleifen 53, 55

G

Garbage-Collector aufrufen 216
 gc() 216
 Generische Klassen/Datentypen 173
 get() 175
 getMessage() 187
 getProperty() 211
 getStackTrace() 189
 Getter-Methoden 80
 GregorianCalendar 201
 Gültigkeitsbereich 28

H

hashCode() 103, 174
 HashTable 211
 hasNext() /
 hasPrevious() 176, 177
 Hauptprogramm, main() 77

I

Identifizier 21
 if 45, 46, 47, 48, 59
 if-else 48, 50
 implements 131
 import 115, 116, 117
 import static 117
 Importieren, Besonderheiten 119
 indexOf() 148, 175
 IndexOutOf-
 BoundsException 214
 Initialisierung 30
 Inkrement 36
 insert() 150
 instanceof 100, 133
 Instant 202
 Instanziierung 65
 Instanzvariablen 64
 IntelliJ IDEA 14
 interface 129
 Interface Methoden,
 Default 129, 136
 Interface Methoden,
 statische 129, 137
 Interfaces 129, 132
 Interfaces erstellen 129
 Interfaces implementieren 130
 Interfaces nutzen 132
 Interpreter 17

intValue() 151
 isEmpty() 148, 174
 Iterable 176
 Iterator 176, 177
 iterator() 174

J

J2SE 10
 Java 2 10
 Java 9 Platform Module System 121
 Java Development Kit, JDK 7, 9
 Java Runtime Environment,
 JRE 10, 12
 Java Web Start Technik 12
 Java, Enterprise Edition 9
 Java, Entwicklungsgeschichte 7
 java, Java-Interpreter 17
 Java, Micro Edition 9
 Java, Standard Edition 9
 Java, Vorteile/Nachteile 7
 java.time 201
 javac 15
 Java-Compiler 11, 15
 javadoc 24
 Java-Interpreter 11, 17
 Java-Laufzeitumgebung 11
 JDK, Java Development Kit 7, 9
 join() 148
 JPMS 121
 JRE 12
 JRE, Java Runtime
 Environment 10, 12

K

Kapselung 79
 Kennwort-Eingabe 217
 Klassen 14, 62, 63, 64
 Klassen importieren 115
 Klassen, abgeleitete 89, 97
 Klassen, abstrakte 106, 132
 Klassen, finale 105
 Klassen, Funktionalität 62, 69, 79
 Klassen, generische 173
 Klassen, qualifizierter Name 111
 Klassenhierarchie 98
 Klassenvariablen 75
 Kommentare erstellen 15, 23
 Kompatibilität 98
 Konstanten verwenden 33, 34
 Konstruktoraufruf 84
 Konstruktoren 83, 92
 Konstruktoren anwenden 84, 85
 Konstruktoren erstellen 83, 84
 Konstruktoren überladen 74

Konstruktoren verketteten 85
 Konvertierung 152
 Kopfgesteuerte Schleifen 53, 56

L

Laufzeitfehler 182
 length 156
 length() 148
 LinkedList 176
 List 171, 172, 174
 Listen 174
 Listen erstellen/erweitern 175
 Listen sequenziell durchlaufen 176
 Listen, Iterator 176
 Listen, Positionszeiger 176
 ListIterator 176, 177
 Literale 22, 27
 LocalDate 204
 LocalDateTime 205
 LocalTime 208
 Logischer Datentyp 27
 Lokale Variablen 28, 64, 66
 long 26

M

main() 15, 18, 77, 163
 Map 172
 Math 117, 199
 Mehrdimensionale Arrays 160
 Mehrseitige Auswahl 50, 51
 Mehrseitige Verzweigung 51
 Methode main 14
 Methoden 14, 62, 79
 Methoden anwenden 70
 Methoden erben 94
 Methoden erstellen 69, 70
 Methoden mit Parametern 71, 72
 Methoden mit Rückgabewert 74
 Methoden redefinieren 95
 Methoden überladen 74
 Methoden überschreiben 95, 135
 Methoden, abstrakte 107
 Methoden, Basisklasse 96
 Methoden, Signatur 71, 95
 Methoden, statische 75, 77
 Modifizierer 63, 64, 113
 Modularisierung 121
 Modulaufbau 123
 Modulbeschreibung 123
 Module 121
 Modulo-Operator 35
 Modulpfad 124
 Mutatoren 81

N

nanoTime()	216
NetBeans	13
new	65, 83, 143, 155
next()	176
nextIndex()	177
NoSuchElementException	177
NumberFormatException	185, 187
Numerische Datentypen	27

O

Object	102, 103, 119
Objekte	62
Objekte erzeugen	65, 66
open	124
opens	124
opens to	124
Operatoren	34
Operatoren, arithmetische	34
Operatoren, binäre	34
Operatoren, Dekrement-	36
Operatoren, Inkrement-	36
Operatoren, logische	39
Operatoren, unäre	37
Operatoren,	
verkürzte Schreibweise	35
Operatoren, Vorzeichen-	37
Overloading	74
Overriding	95

P

package	113
Package-Namen	111
Packages	110
Packages einrichten	110
Packages identifizieren	111
Packages, Klassen zuordnen	112
Packages, Standard-	119
Packages, Übersicht	120
Packages, Zugriffsrechte	113
Package-Zugriffsrecht, Default-	113
Parameter	71, 164
Parameter, Programm-	41
Parameterübergabe	163
parseInt()	151, 185
Path-Variable	16
Period	204, 205
PI	117
Polymorphie	101
Positionszeiger	176
Präfix-/Postfix-Notation	36
previous()	177

previousIndex()	177
print()	144
printf()	144, 217
println()	144
printSysInfo()	212
private	80, 90, 107
Programm ausführen	11, 17, 18
Programm kompilieren	15
Programm starten	17, 18
Programm übersetzen	11, 15
Programm, Dateneingaben	42
Programmbeendigung	213
Programmparameter übergeben	41
Properties	211
propertyName()	212
protected	91
provides with	124
public	113, 129

Q

Qualifizierter Klassenname	111
Quellcode	11
Quelltext	11
Queue	172, 174

R

Random	199
random()	199
readLine()	217
readPassword()	217
Referenzvariable this	81
Referenzvariablen	67, 68, 132, 157
remove()	174, 176
replace()	148
requires	124
requires transitive	124
Reservierte Wörter	22
return	73, 74, 135, 193
Rückgabewert	74
Runtime-	
Exception	183, 184, 185

S

Scanner	42
Schleifen verwenden	52, 53
Schleifen, foreach-	160, 179
Schleifen, fußgesteuerte	53, 55
Schleifen, kopfgesteuerte	53, 56
Schleifenrumpf	53
Schleifensteuerung	53, 59
Schlüsselwörter	22
Schnittstellen	129
Selektor	50

Semikolon	15
Set	172
set()	175, 177
setIn()/setOut()/	
setErr()	213
setSeed()	199
Setter-Methoden	81, 82
short	26
Signatur	71, 95
signum()	151
size()	174
sort()	162
Sourcecode	11
sqrt()	117
Standardeingabe/-ausgabe	
umleiten	213
Standardkonstruktor	83, 92
Standard-Packages	119
static	76, 106, 117, 129
Statische Methoden	75, 77
Statische Variablen	75, 76
String	143, 147
StringBuffer	149
Strings ausgeben	144
Strings vergleichen	145
Strings verketteten	145
Strings, Methoden	147
Strings, Zeilenumbruch	144
Subklasse	89
substring()	148
Sun Microsystems	7
super	93
Superklasse	89, 102
switch	45, 50, 51, 59
Syntax	21
System	158, 211
System.out.println()	15
Systemeigenschaften	211

T

TemporalAdjusters	207
this	81
throw	182, 194
Throwable	183, 187, 195
throws	191
toArray()	174
toLowerCase()	148
toString()	103, 149, 150,
	151, 162, 180, 187
toUpperCase()	148
trim()	148
true	27
try	185, 189, 193
Typargumente	172

Type cast	99	Variablen	28, 29, 64, 68	while	46, 53, 54, 55, 56, 59
Typkompatibilität	31, 98, 100	Variablen benennen	29	Wrapper-Klassen	
Typkonversion	31, 32, 99	Variablen erzeugen	66	verwenden	150, 151
U		Variablen initialisieren	30	Z	
Überladen, Konstruktoren	74	Variablen, lokale	28, 64, 66	Zahlen	25
Überladen, Methoden	74	Variablen, Referenz-	68, 157	Zeichen	25
Überschreiben, Methoden	95	Variablen, statische	75	Zeichenkette	144
Unäre Operatoren	37	Variablen, unveränderliche	33	Zeilenumbruch	144
Unboxing	152	Vererbung	89, 138	Zeitmessung	216
Unicode	22	Vererbungsketten	98	ZonedDateTime	204, 206
Unmutable	143	Vergleichsoperatoren	38	Zufallszahlen	199
Unterklasse	89	Verkettete Listen erstellen	175	Zugriffsmethoden	80
Unveränderliche Variablen	33	Verzweigung, mehrseitige	51	Zugriffsmodifizierer	80, 90, 91, 113
uses	124	void	69, 73, 83	Zugriffsrechte	95, 113
V		Vorzeichenoperatoren	37	Zuweisung	68, 157
valueOf ()	148, 151	W		Zuweisungskompatibilität	98
Varargs	163	Webcode	6	Zuweisungsoperator	30, 38
Variable, Path	16	Webcode für Anhangdateien	5		
		Wertemengen	165		
		Wertzuweisungen	30		

Impressum

Matchcode: JAV9

Autor: Elmar Fuchs

Produziert im HERDT-Digitaldruck

1. Ausgabe, 1. Aktualisierung, Mai 2021

HERDT-Verlag für Bildungsmedien GmbH

Am Kümmerling 21–25

55294 Bodenheim

Internet: www.herdtd.com

E-Mail: info@herdtd.com

© HERDT-Verlag für Bildungsmedien GmbH, Bodenheim

Alle Rechte vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlags reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Dieses Buch wurde mit großer Sorgfalt erstellt und geprüft. Trotzdem können Fehler nicht vollkommen ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Wenn nicht explizit an anderer Stelle des Werkes aufgeführt, liegen die Copyrights an allen Screenshots beim HERDT-Verlag. Sollte es trotz intensiver Recherche nicht gelungen sein, alle weiteren Rechteinhaber der verwendeten Quellen und Abbildungen zu finden, bitten wir um kurze Nachricht an die Redaktion.

Die in diesem Buch und in den abgebildeten bzw. zum Download angebotenen Dateien genannten Personen und Organisationen, Adress- und Telekommunikationsangaben, Bankverbindungen etc. sind frei erfunden. Eventuelle Übereinstimmungen oder Ähnlichkeiten sind unbeabsichtigt und rein zufällig.

Die Bildungsmedien des HERDT-Verlags enthalten Verweise auf Webseiten Dritter. Diese Webseiten unterliegen der Haftung der jeweiligen Betreiber, wir haben keinerlei Einfluss auf die Gestaltung und die Inhalte dieser Webseiten. Bei der Bucherstellung haben wir die fremden Inhalte daraufhin überprüft, ob etwaige Rechtsverstöße bestehen. Zu diesem Zeitpunkt waren keine Rechtsverstöße ersichtlich. Wir werden bei Kenntnis von Rechtsverstößen jedoch umgehend die entsprechenden Internetadressen aus dem Buch entfernen.

Die in den Bildungsmedien des HERDT-Verlags vorhandenen Internetadressen, Screenshots, Bezeichnungen bzw. Beschreibungen und Funktionen waren zum Zeitpunkt der Erstellung der jeweiligen Produkte aktuell und gültig. Sollten Sie die Webseiten nicht mehr unter den angegebenen Adressen finden, sind diese eventuell inzwischen komplett aus dem Internet genommen worden oder unter einer neuen Adresse zu finden. Sollten im vorliegenden Produkt vorhandene Screenshots, Bezeichnungen bzw. Beschreibungen und Funktionen nicht mehr der beschriebenen Software entsprechen, hat der Hersteller der jeweiligen Software nach Drucklegung Änderungen vorgenommen oder vorhandene Funktionen geändert oder entfernt.