



mitp

Elisabeth
Jung

Java Übungsbuch

Für die Versionen Java 8 bis Java 13

Aufgaben mit vollständigen Lösungen

Inhaltsverzeichnis

	Einleitung	13
	Vorkenntnisse	13
	Aufbau des Buches	13
	Benötigte Software	17
	Autorin	18
1	Klassendefinition und Objektinstanziierung	19
1.1	Klassen und Objekte	19
	☆ Aufgabe 1.1: Definition einer Klasse	23
	☆ Aufgabe 1.2: Objekt (Instanz) einer Klasse erzeugen	23
1.2	Das Überladen von Methoden	23
	☆ Aufgabe 1.3: Eine Methode überladen	24
1.3	Die Datenkapselung, ein Prinzip der objektorientierten Programmierung	24
	☆ Aufgabe 1.4: Zugriffsmethoden	25
1.4	Das »aktuelle Objekt« und die »this-Referenz«	25
	☆☆ Aufgabe 1.5: Konstruktordefinitionen	25
1.5	Die Wert- und Referenzübergabe in Methodenaufrufen	26
	☆☆☆ Aufgabe 1.6: Wertübergabe in Methoden (»call by value«)	26
1.6	Globale und lokale Referenzen	27
	☆☆ Aufgabe 1.7: Der Umgang mit Referenzen	27
1.7	Java-Pakete	27
	☆ Aufgabe 1.8: Die package-Anweisung	28
	☆ Aufgabe 1.9: Die import-Anweisung	29
1.8	Die Modifikatoren für Felder und Methoden in Zusammenhang mit der Definition von Paketen	29
	☆☆ Aufgabe 1.10: Pakete und die Sichtbarkeit von Mitgliedern einer Klasse	30
1.9	Standard-Klassen von Java	30
1.10	Die Wrapper-Klassen von Java und das Auto(un)boxing	32
	☆☆ Aufgabe 1.11: Das Auto(un)boxing	34
1.11	Das Paket java.lang.reflect.	34
1.12	Arrays (Reihungen) und die Klassen Array und Arrays	36
	☆☆ Aufgabe 1.12: Der Umgang mit Array-Objekten	37
1.13	Zeichenketten und die Klasse String	37

	☆	Aufgabe 1.13: Der Umgang mit String-Objekten	46
	☆☆	Aufgabe 1.14: Der Umgang mit Textblöcken	47
1.14		Sprachmerkmale, die in den weiteren Beispielen genutzt werden	48
	☆	Aufgabe 1.15: Methoden mit variablen Argumentenlisten	49
1.15		Den Zugriff auf Klassen und Felder minimieren: Unveränderliche (immutable) Klassen und Objekte	49
1.16		Die alte und neue Date&Time-API als Beispiel für veränderliche und unveränderliche Klassen	51
	☆☆☆	Aufgabe 1.16: Die Methoden von Date/Time-Klassen	55
1.17		Private Konstruktoren	56
	☆	Aufgabe 1.17: Objekte mithilfe eines privaten Konstruktors erzeugen	56
1.18		Lösungen	57
		Lösung 1.1	57
		Lösung 1.2	57
		Lösung 1.3	58
		Lösung 1.4	59
		Lösung 1.5	60
		Lösung 1.6	62
		Lösung 1.7	64
		Lösung 1.8	67
		Lösung 1.9	67
		Lösung 1.10	68
		Lösung 1.11	69
		Lösung 1.12	72
		Lösung 1.13	75
		Lösung 1.14	78
		Lösung 1.15	87
		Lösung 1.16	89
		Lösung 1.17	96
2		Abgeleitete Klassen und Vererbung	99
2.1		Abgeleitete Klassen	99
2.2		Die Konstruktoren von abgeleiteten Klassen	99
2.3		Abgeleitete Klassen und die Sichtbarkeit von Feldern und Methoden	99
	☆☆	Aufgabe 2.1: Test von Sichtbarkeitsebenen im Zusammenhang mit abgeleiteten Klassen	102

2.4	Das Verdecken von Klassenmethoden und das statische Binden von Methoden	103
	☆☆ Aufgabe 2.2: Der Aufruf von verdeckten Klassenmethoden	103
2.5	Das Überschreiben von Instanzmethoden und das dynamische Binden von Methoden	104
	☆ Aufgabe 2.3: Das dynamische Binden von Methoden	105
2.6	Vererbung und Komposition	105
	☆ Aufgabe 2.4: Die Komposition	106
	☆ Aufgabe 2.5: Die Vererbung	106
2.7	Kovariante Rückgabetypen in Methoden	107
	☆☆ Aufgabe 2.6: Die Benutzung von kovarianten Rückgabetypen	107
2.8	Verdeckte Felder	108
2.9	Vergrößernde und verkleinernde Konvertierung (»up- und down-casting«)	108
2.10	Der Polymorphismus, ein Prinzip der objektorientierten Programmierung	108
	☆☆ Aufgabe 2.7: Der »Subtyp-Polymorphismus« im Kontext einer Klassenhierarchie	109
2.11	Die Methoden der Klassen <code>java.lang.Object</code> und <code>java.util.Objects</code>	110
	☆ Aufgabe 2.8: Die <code>equals()</code> - und <code>hashCode()</code> -Methoden von <code>Object</code>	117
	☆☆☆ Aufgabe 2.9: Die <code>equals()</code> -Methode und die Vererbung	118
2.12	Das Klonen und die Gleichheit von geklonten Objekten	122
	☆ Aufgabe 2.10: Das Klonen von Instanzen der eigenen Klasse	122
	☆ Aufgabe 2.11: Das Klonen von Instanzen anderer Klassen	122
	☆ Aufgabe 2.12: Das Klonen und der Copy-Konstruktor	123
2.13	Der Garbage Collector und das Beseitigen von Objekten	123
2.14	Lösungen	124
	Lösung 2.1	124
	Lösung 2.2	128
	Lösung 2.3	130
	Lösung 2.4	132
	Lösung 2.5	134
	Lösung 2.6	136
	Lösung 2.7	138

	Lösung 2.8	141
	Lösung 2.9	147
	Lösung 2.10	154
	Lösung 2.11	155
	Lösung 2.12	156
3	Die Definition von abstrakten Klassen, Interfaces und Annotationen	159
3.1	Abstrakte Klassen	159
3.2	Abstrakte Java-Standard-Klassen und eigene Definitionen von abstrakten Klassen	159
	☆ Aufgabe 3.1: Die abstrakte Klasse Number und ihre Unterklassen	159
	☆ Aufgabe 3.2: Definition einer eigenen abstrakten Klasse	160
3.3	Interfaces (Schnittstellen)	161
	☆☆ Aufgabe 3.3: Die Definition eines Interface.	161
3.4	Die Entscheidung zwischen abstrakten Klassen und Interfaces ...	162
	☆☆ Aufgabe 3.4: Paralleler Einsatz von Interfaces und abstrakten Klassen	163
3.5	Implementieren mehrerer Interfaces für eine Klasse	164
	☆☆ Aufgabe 3.5: Das Ableiten von Interfaces	164
3.6	Die Definition von inneren Klassen	165
	☆ Aufgabe 3.6: Ein Beispiel mit anonymer Klasse	167
3.7	Erweiterungen in der Definition von Interfaces	168
	☆☆☆ Aufgabe 3.7: Private Interface-Methoden.	169
3.8	Die Definition von Annotationen	171
3.9	Vordefinierte Annotationstypen	174
	☆ Aufgabe 3.8: Annotationen an Methoden und Parameter von Methoden anheften	175
	☆ Aufgabe 3.9: Eine Klasse annotieren	176
	☆☆ Aufgabe 3.10: Die @Override- und @Inherited-Annotation.	176
3.10	Lösungen	177
	Lösung 3.1	177
	Lösung 3.2	179
	Lösung 3.3	180
	Lösung 3.4	183
	Lösung 3.5	186
	Lösung 3.6	188
	Lösung 3.7	189
	Lösung 3.8	192

	Lösung 3.9	194
	Lösung 3.10	196
4	Generics	201
4.1	Die Generizität	201
4.2	Generische Klassen und Interfaces	202
	☆ Aufgabe 4.1: Generischer Datentyp als Behälter für die Instanzen vom Typ des Klassenparameters	203
	☆ Aufgabe 4.2: Generischer Datentyp als »Über-Typ« für die Instanzen vom Typ des Klassenparameters	203
4.3	Wildcardtypen	204
	☆☆ Aufgabe 4.3: Ungebundene Wildcardtypen	204
	☆☆ Aufgabe 4.4: Obere und untere Schranken für Wildcardtypen	204
4.4	Legacy Code, Erasure und Raw-Typen	205
	☆☆ Aufgabe 4.5: Raw-Typen am Beispiel einer generischen Klasse mit zwei Typparametern	206
	☆☆ Aufgabe 4.6: Brückenmethoden (»bridge methods«)	207
4.5	Generische Arrays	208
	☆☆ Aufgabe 4.7: Erzeugen von generischen Arrays	208
4.6	Generische Methoden	209
	☆☆ Aufgabe 4.8: Generische Methodendefinitionen	209
4.7	Generische Standard-Klassen und -Interfaces	209
4.8	for-each-Schleifen für Collections	212
	☆ Aufgabe 4.9: Das Interface List<E> und die Klasse ArrayList<E>	213
	☆☆ Aufgabe 4.10: Das Interface Collection<E> und die Klasse Vector<E>	213
	☆☆ Aufgabe 4.11: Das Interface Map<K,V> und die Klasse TreeMap<K,V>	214
4.9	Factory-Methoden in Collections	215
	☆ Aufgabe 4.12: Factory-Methoden für List, Set und Map	216
4.10	Die Interfaces Enumeration<E>, Iterable<T> und Iterator<E>	216
4.11	Enumerationen und die generische Klasse Enum<E> extends Enum<E>	217
	☆☆ Aufgabe 4.13: Die Definition von Enumerationen	218
4.12	Die Interfaces Comparable<T> und Comparator<T> und das Sortieren von Objekten	218
	☆☆☆☆ Aufgabe 4.14: Das Comparable<T>-Interface	221
	☆☆☆☆ Aufgabe 4.15: Comparable<T> versus Comparator<T>	224

4.13	Typinferenz für Methoden	227
4.14	Typinferenz beim Erzeugen von Instanzen eines generischen Typs	227
	☆☆ Aufgabe 4.16: Typinferenz beim Instanzieren von generischen Klassen	230
	☆☆ Aufgabe 4.17: Der Diamond-Operator in Java 9	231
4.15	Lösungen	233
	Lösung 4.1	233
	Lösung 4.2	234
	Lösung 4.3	235
	Lösung 4.4	236
	Lösung 4.5	238
	Lösung 4.6	240
	Lösung 4.7	242
	Lösung 4.8	243
	Lösung 4.9	245
	Lösung 4.10	246
	Lösung 4.11	247
	Lösung 4.12	249
	Lösung 4.13	251
	Lösung 4.14	253
	Lösung 4.15	261
	Lösung 4.16	266
	Lösung 4.17	270
5	Exceptions und Errors	275
5.1	Ausnahmen auslösen	275
5.2	Ausnahmen abfangen oder weitergeben	276
	☆☆ Aufgabe 5.1: Unbehandelte RuntimeExceptions	276
	☆☆ Aufgabe 5.2: Behandelte RuntimeExceptions	276
5.3	Das Verwenden von finally in der Ausnahmebehandlung	277
	☆☆ Aufgabe 5.3: Der finally-Block	277
5.4	Ausnahmen manuell auslösen	278
5.5	Exception-Unterklassen erzeugen	278
	☆☆ Aufgabe 5.4: Benutzerdefinierte Ausnahmen manuell auslösen	278
5.6	Multi-catch-Klausel und verbesserte Typprüfung beim Rethrowing von Exceptions	279
	☆☆ Aufgabe 5.5: Disjunction-Typ für Exceptions	280
	☆☆ Aufgabe 5.6: Typprüfung beim Rethrowing von Exceptions	281

5.7	Lösungen	283
	Lösung 5.1	283
	Lösung 5.2	284
	Lösung 5.3	286
	Lösung 5.4	287
	Lösung 5.5	289
	Lösung 5.6	292
6	Lambdas und Streams	297
6.1	Mittels anonymer Klassen Code an Methoden übergeben	297
6.2	Funktionale Interfaces	297
6.3	Syntax und Deklaration von Lambda-Ausdrücken	298
	☆ Aufgabe 6.1: Lambda-Ausdruck ohne Parameter versus anonymer Klasse	300
	☆ Aufgabe 6.2: Lambda-Ausdruck mit Parameter versus anonymer Klasse	301
6.4	Scoping und Variable Capture	302
	☆☆ Aufgabe 6.3: Die Umgebung von Lambda-Ausdrücken	303
6.5	Methoden- und Konstruktor-Referenzen	304
	☆ Aufgabe 6.4: Methoden-Referenzen in Zuweisungen	306
	☆☆ Aufgabe 6.5: Konstruktor-Referenzen und die neuen funktionalen Interfaces <code>Supplier<T></code> und <code>Function<T,R></code>	307
6.6	Default- und statische Methoden in Interfaces	308
6.7	Das neue Interface <code>Stream</code>	310
6.8	Die <code>forEach</code> -Methoden von <code>Iterator</code> , <code>Iterable</code> und <code>Stream</code>	313
	☆ Aufgabe 6.6: Die funktionalen Interfaces <code>BiConsumer<T,U></code> , <code>BiPredicate<T,U></code> und <code>BiFunction<T,U,R></code>	315
	☆☆ Aufgabe 6.7: Die Methoden des Interface <code>Stream</code> und die Behandlung von <code>Exceptions</code> in <code>Lambda-Ausdrücken</code>	316
6.9	Das Interface <code>Collector</code> und die Klasse <code>Collectors</code> : Reduktion mittels Methoden von <code>Streams</code> und <code>Kollektoren</code>	317
	☆☆ Aufgabe 6.8: Weitere Methoden des Interface <code>Stream</code> : <code>limit()</code> , <code>count()</code> , <code>max()</code> , <code>min()</code> , <code>skip()</code> , <code>reduce()</code> und <code>collect()</code>	319
	☆☆☆☆ Aufgabe 6.9: Das Interface <code>Collector</code> und die Klasse <code>Collectors</code>	322
6.10	Parallele <code>Streams</code>	323
	☆☆☆☆ Aufgabe 6.10: Parallele <code>Streams</code>	325

6.11	Die map()- und flatMap()-Methoden von Stream und Optional.	327
	☆☆ Aufgabe 6.11: map() versus flatMap()	329
6.12	Spracherweiterungen mit Java 10, Java 11, Java 12 und Java 13	331
	☆☆ Aufgabe 6.12: Typinferenz für lokale Variablen in Java 10 und Java 11.	336
	☆☆ Aufgabe 6.13: Switch-Statements und Switch-Expressions	337
6.13	Lösungen	339
	Lösung 6.1	339
	Lösung 6.2	340
	Lösung 6.3	342
	Lösung 6.4	347
	Lösung 6.5	348
	Lösung 6.6	353
	Lösung 6.7	357
	Lösung 6.8	363
	Lösung 6.9	381
	Lösung 6.10	391
	Lösung 6.11	400
	Lösung 6.12	405
	Lösung 6.13	412
7	Die Modularität von Java	419
7.1	Das Java-Modulsystem.	419
	☆☆ Aufgabe 7.1: Eine einfache Modul-Definition	426
	☆☆ Aufgabe 7.2: Eine Applikation mit mehreren Modulen	428
	☆☆ Aufgabe 7.3: Implizites Lesen von Modulen	430
	☆☆ Aufgabe 7.4: Eine modulbasierte Service-Implementierung	432
7.2	Lösungen	433
	Lösung 7.1	433
	Lösung 7.2	435
	Lösung 7.3	441
	Lösung 7.4	447
	Stichwortverzeichnis	453

Einleitung

Das *Java Übungsbuch für die Versionen Java 8 bis Java 13* ist wie alle meine Übungsbücher aus der Erkenntnis entstanden, dass zu umfangreiche Beispiele mit komplizierten Algorithmen beim Lernen von Java am Anfang keine echte Hilfe bieten. Darum liegt der Schwerpunkt des Buches nicht auf der Umsetzung von komplizierten Vorgängen, sondern auf der Art und Weise. Stattdessen konzentriert es sich darauf, die in der Dokumentation nicht immer verständlich formulierten Erläuterungen zu Java-Klassen und -Interfaces mit einfachen Beispielen zu erklären und gleichzeitig die zugrunde liegenden Konzepte zu erörtern.

Dieses Buch wendet sich in erster Linie an Lehrer, Schüler und Studenten als Begleitliteratur zum Lernen der Programmiersprache Java, ist aber auch zum Selbststudium für alle Interessenten an dem Erlernen der Programmiersprache geeignet.

Durch die Einfachheit und Vollständigkeit der Aufgabenlösungen sowie die unterschiedlichen Lösungsmöglichkeiten erhält der Leser ein fundiertes Verständnis für die Aufgabenstellungen und deren Lösungen.

Durch das Lösen von Aufgaben soll der in Referenz- und Lehrbüchern von Java angebotene Stoff vertieft werden, und die dabei erzielten Ergebnisse können anhand der Lösungsvorschläge überprüft werden. Die Beispiele im Buch sind eher selten von zu komplexer Natur, sodass der eigentliche Zweck nicht in den Hintergrund tritt, und alle beschriebenen Themen können tiefgehend und präzise damit eingeübt werden.

Vorkenntnisse

Es ist Voraussetzung, dass der Leser zusätzlich mit einem Lehrbuch zu Java arbeitet bzw. bereits damit gearbeitet hat. Die grundlegenden Erläuterungen zu Java in diesem Buch können lediglich als Wiederholung des bereits vorhandenen Wissens dienen, reichen aber nicht aus, um die Sprache Java erst neu zu lernen.

Als weitere Voraussetzung sind Grundlagen im Bereich der Programmierung und im Umgang mit dem Betriebssystem erforderlich. Ein paralleler Zugriff auf die Java-Online-Dokumentation kann Hilfe zu den Java-Standard-Klassen bieten.

Aufbau des Buches

Jedes Kapitel beginnt mit einer kurzen und knappen Wiederholung des Stoffes, der in den Übungsaufgaben dieses Kapitels verwendet wird. Danach folgen alle

Aufgabenstellungen der Übungen. Am Ende des Kapitels finden Sie gesammelt die Lösungen der Übungsaufgaben mit Kommentaren, Erläuterungen und Hinweisen.

Die Aufgaben haben unterschiedliche Schwierigkeitsgrade. Dieser wird im Aufgabekopf durch ein bis drei Sternchen gekennzeichnet:



1 Sternchen für besonders einfache Aufgaben, die auch von Anfängern leicht bewältigt werden können



2 Sternchen für etwas kompliziertere Aufgaben, die einen durchschnittlichen Aufwand benötigen



3 Sternchen für Aufgaben, die sich an geübte Programmierer richten und einen wesentlich höheren Aufwand oder die Kenntnis von speziellen Details erfordern

Die Programme aus früheren Übungen werden teilweise in späteren Übungen gebraucht und es wird auch immer wieder auf theoretische Zusammenhänge zurückgekommen oder hingewiesen.

Die Lösungsvorschläge haben umfangreiche Kommentare, sodass ein Verständnis für die durchgeführte Aufgabe auch daraus abgeleitet werden kann und dadurch jede einzelne Aufgabe im Gesamtkontext unabhängig erscheint.

In den Kapiteln 1, 2 und 3 liegt das Hauptmerkmal auf den Eigenheiten der objektorientierten Programmierung mit Java. Durch eine Vielzahl von Beispielen wird gezeigt, was die Java-Standard-Klassen und Interfaces an Funktionalitäten bieten und wie diese sinnvoll in die Definition von eigenen Klassen eingebettet werden können. Diese Kapitel enthalten zusätzlich Informationen zur Reflection-API von Java, der Definition von Annotationen und inneren Klassen sowie Neuerungen aus den Versionen 8 bis 13, die sich auf die neue Date&Time-API, Textblöcke, Compact Strings und die Weiterentwicklung von Interfaces beziehen. Mit Java 8 wurden sogenannte Default-Methoden eingeführt. Diese werden in der Literatur auch als »virtual extension«- bzw. »defender«-Methoden bezeichnet und Schnittstellen, die über derartige Methoden verfügen, als erweiterte Schnittstellen. Damit können Interfaces zusätzlich zu abstrakten Methoden konkrete Methoden in Form von Standard-Implementierungen definieren und in Java wird die Mehrfachvererbung von Funktionalität ermöglicht. Neben Default-Methoden können Interfaces in Java

nun auch statische Methoden enthalten. Anders als die statischen Methoden von Klassen werden diese jedoch nicht von abgeleiteten Typen geerbt.

Kapitel 4 beschäftigt sich im Detail mit Generics und dem Collection Framework mit all seinen generischen Klassen und Interfaces sowie mit der Definition von Enumerationen. Die Typinferenz für Methoden und beim Erzeugen von generischen Typen (der Diamond-Operator) sowie das Subtyping von parametrisierten und Wildcard-parametrisierten Typen sind ebenfalls Gegenstand der Themen aus diesem Kapitel.

Kapitel 5 erläutert das Exception-Handling.

Kapitel 6 beschäftigt sich mit den neuen Sprachmitteln von Java 8, Lambdas und Streams sowie mit weiteren Neuerungen aus den Versionen 8 bis 13, wie Switch-Expressions und Local Variable Type Inference.

Mit der Java-Version 8.0 haben sich ganz neue Betrachtungsweisen und Programmieretechniken in der Entwicklung von Applikationen mit Java eröffnet. Eine der wichtigsten Neuerungen in Java 8 sind neue Sprachmittel, die sogenannten Lambda-Ausdrücke, eine Art anonyme Methoden, die auf funktionalen Interfaces basieren. Diese besitzen jedoch eine viel kompaktere Syntax als Methoden. Das resultiert daraus, dass in ihrer Benutzung auf Namen, Modifikatoren, Rückgabety, throws-Klausel und in vielen Fällen auch auf Parameter verzichtet werden kann. Mit ihnen kann Funktionalität ausgeführt, gespeichert und übergeben werden, wie dies bisher nur von Instanzen in Java bekannt war.

Damit verbundene Themen wie die Gegenüberstellung zu anonymen Klassen, Syntax und Semantik, Behandlung von Exceptions, Scoping und Variable Capture, Methoden- und Konstruktor-Referenzen werden in den ersten Unterkapiteln des 6. Kapitels dieses Buches beschrieben und anhand von vielen Beispielen erläutert.

Des Weiteren finden Sie hier die Beschreibung aller neuen funktionalen Interfaces und deren Methoden. Die nachfolgenden Unterkapitel beschäftigen sich im Detail mit der Definition und Nutzung von Streams. Ein Stream besteht aus einer Folge von Werten (in der Literatur wird auch von Sequenzen von Elementen gesprochen), die nur teilweise von mehreren in einer Pipeline dazwischenliegenden Operationen ausgewertet und durch eine abschließende Operation bereitgestellt werden. Diese Operationen werden in Java als Methodenaufrufe formuliert, die Funktionalität in Form von Lambdas und Methoden-Referenzen entgegennehmen können und diese auf alle Elemente der Folge anwenden.

Mit einer Vielzahl von Aufgaben basierend auf Lambdas, Streams und Kollektoren (in denen Stream-Elemente angesammelt und reduziert werden können) werden die neuen Techniken angewandt und alle neuen Begriffe erklärt.

Kapitel 7 präsentiert das neue Java-Modulsystem. Mit dem neuen Modulsystem wurde Java selbst modular gemacht und es können eigene Applikationen und Bibliotheken modularisiert werden.

Java 9 führt das Modul als eine neue Programmkomponente ein. Das Erzeugen von Modulen und deren Abhängigkeiten führt dazu, dass der Zugriffsschutz in Java 9 restriktiver ist. Das Anlegen der erforderlichen Verzeichnisstrukturen für modulbasierte Applikationen, das Packaging von Modul-Code sowie die Implementierung von Services werden ebenfalls im Detail erklärt. Eine Vielzahl von Applikationen mit ausführlichen `.cmd`-Dateien für deren Ausführung ergänzen die theoretischen Erläuterungen aus diesem Kapitel.

Weitere Sprachänderungen und Neuigkeiten aus den Versionen Java 10 bis Java 13 sind in allen Kapiteln integriert. Dazu gehören der Diamond-Operator für innere anonyme Klassen, Default- und private Methoden in Interfaces, Erweiterungen der `switch`-Anweisung und die Einführung von Switch-Expressions, die Typinferenz für lokale Variablen und nicht zuletzt die mit Java 13 neu eingeführten Textblöcke, die bereits im ersten Kapitel im Vergleich zu den klassischen String-Literalen ausführlich und mit vielen Beispielen präsentiert werden.

Weil der Schwerpunkt des Buches nicht auf der Umsetzung von aufwendigen Algorithmen liegen soll, verwende ich einfache Beispiele mit Zahlen, Buchstaben, Wörtern, Büchern, Wochentagen, geometrischen Figuren etc. und teilweise auch mit ganz abstrakten Klassennamen wie `Klasse1`, `Klasse2`, `KlasseA`, `KlasseB` etc.

An dieser Stelle möchte ich auf das dem Buch zugrunde liegende Konzept hinweisen, das parallel zu einfachen Aufgaben, die zu allen eingeführten Definitionen und Begriffen gebracht werden, auch Aufgaben von einem höheren Schwierigkeitsgrad präsentiert werden. Dabei werden anhand von inhaltlichen Zusammenhängen zwischen den Beispielen viele Basiskonzepte von Java erläutert.

Ich habe generell versucht, keine Begriffe, Klassen und Komponenten zu benutzen, die nicht schon in vorangehenden Beispielen und Kapiteln definiert oder erläutert wurden. In den wenigen Fällen, wo es sich nicht vermeiden ließ, wird darauf hingewiesen und auf die entsprechenden Stellen verwiesen.

Das Buch soll möglichst parallel zu einer Vielzahl von Java-Lehrbüchern eingesetzt werden können und einen Beitrag dazu leisten, die große Fülle von Informationen, die auf uns über die API-Dokumentation zukommt, besser einzuordnen und korrekt anwenden zu können.

Schrecken Sie nicht davor zurück, von Anfang an (gerade bei den schwierigeren Aufgaben) die Anforderungen aus dem Aufgabentext mit den Ergebnissen aus dem Lösungsvorschlag zu vergleichen (zumindest anfangs und vielleicht auch nur teilweise). Nahe an der Programmiersprache formuliert, sollen diese Aufgaben in erster Linie dazu dienen, das Programmieren mit Java zu erlernen, ohne sich gleichzeitig auf aufwendige Algorithmen zu konzentrieren. Es ist ja auch mit ein Grund, warum die Bücher vollständige Lösungsvorschläge beinhalten. Sie sind gerade dafür gedacht, die Theorie besser zu verstehen, aber auch gleichzeitig mit Beispielen einzuüben.

Andererseits verpflichten diese Bücher nicht, selbst auf die gleiche Lösung zu kommen, und enthalten nur Vorschläge zu den Lösungen von Aufgaben.

Anhand eines umfangreichen Index können Sie beim selbstständigen Programmieren im Buch nachschlagen, wenn Sie nach einem Lösungsansatz oder Hilfe beim Beseitigen von Fehlern suchen sollten.

Benötigte Software

Das aktuelle Java Development Kit der Java Standard Edition können Sie sich kostenlos von der Java-Homepage von Oracle <http://www.oracle.com/technetwork/java/javase/downloads/index.html> herunterladen. Das JDK umfasst sowohl die Software zur Programmerstellung als auch das JRE (Java Runtime Environment) für die Programmausführung.

Grafische Entwicklungsoberflächen (wie z. B. Eclipse) sind keine Voraussetzung und auch nicht Bestandteil dieses Buches. Die Programme lassen sich grundsätzlich mit einem Texteditor wie z. B. Notepad++ oder auch Wordpad eingeben und über die Kommandozeile durch den Aufruf der Programme javac, jar und java übersetzen, paketieren und starten. Die vollständigen Programmaufrufe sind bei jeder Aufgabe angegeben.

Sollte Ihnen beim Erlernen der Programmiersprache selbst eine Entwicklungsumgebung nicht zu aufwendig oder unübersichtlich erscheinen, steht Ihnen nichts im Wege, die zu den Aufgaben zugehörigen Programm- und Klassendateien zum Testen oder auch Ergänzen in eine solche einzubetten.

Website

Die Website zum Buch unter www.mitp.de/0123 beinhaltet den plattformunabhängigen Quellcode der Lösungsvorschläge und die für Windows kompilierte ausführbare Version als Download-Archiv. Diese Archivdatei enthält alle Java-Quellcodes, übersetzten Klassen und Bilddateien in einer Verzeichnisstruktur, die mit der im Buch beschriebenen übereinstimmt. Zusätzlich finden Sie auf dieser Webseite die Datei `Java9Migration.pdf`, in der die Migration von Anwendungen beschrieben wird, die durch verschiedene Kategorien von Modulen unterstützt werden kann.

Ich wünsche Ihnen viel Erfolg beim Programmieren mit Java.

Elisabeth Jung

Klassendefinition und Objektinstanziierung

1.1 Klassen und Objekte

Klassen und Objekte bilden die Basis in der objektorientierten Programmierung. Eine Klasse ist eine Ansammlung von Attributen, die Eigenschaften definieren und Felder genannt werden, und von Funktionen, die deren Zustände und Verhaltensweisen festlegen und als Methoden bezeichnet werden. Felder und Methoden werden auch Member der Klasse genannt.

Klassen werden mit dem Schlüsselwort `class` eingeleitet und definieren eine logische Abstraktion, die eine Basisstruktur für Objekte vorgibt. Sie sind als eine Erweiterung der primitiven Datentypen zu sehen. Während Klassen Vorlagen (Modelle) definieren, sind Objekte konkrete Exemplare, auch Instanzen der Klasse genannt.

Eine Schnittstelle (Interface) ist eine reine Spezifikation, die definiert, wie eine Klasse sich zu verhalten hat. Sie wird mit dem Schlüsselwort `interface` eingeleitet und konnte zu den Anfangszeiten von Java keine Implementationen von Feldern und Methoden enthalten, mit Ausnahme von Konstantendefinitionen (die als `static` und `final` deklariert werden). Mit der Weiterentwicklung von Java wurden sowohl `public static`-Methoden als auch `public default`-Methoden in Interfaces zugelassen (Java 8). Um redundanten Code zu vermeiden und die Benutzung von Interfaces zu verbessern, wurden mit Java 9 zusätzlich `private`-Methoden zugelassen. Einige dieser Methoden können Implementierungen liefern. Wir werden im weiteren Verlauf darauf näher eingehen.

Ein Objekt einer Klasse wird in Java mit dem `new`-Operator und einem Konstruktor erzeugt. Damit werden auch seine Felder initialisiert und der erforderliche Speicherplatz für das Objekt reserviert. Ein Objekt wird über eine Referenz angesprochen. Eine Referenz entspricht in Java in etwa einem Zeiger in anderen Programmiersprachen und ist einem Verweis auf das Objekt gleichzustellen, womit dieses identifiziert werden kann.

Mit Referenztypen werden Datentypen bezeichnet, die im Gegensatz zu den primitiven Datentypen vom Entwickler selbst definiert werden. Diese können vom Typ einer Klasse, einer Schnittstelle oder eines Arrays sein. Ein Arraytyp identifiziert ein Objekt, das mehrere Werte von ein und demselben Typ speichern kann.

Die mit dem Modifikator `static` deklarierten Felder und Methoden in einer Klasse werden als Klassenfelder bzw. Klassenmethoden bezeichnet. Alle anderen Felder

und Methoden einer Klasse werden auch Instanzfelder bzw. Instanzmethoden genannt.

Die Klassen bilden in Java eine Klassenhierarchie. Jede Klasse hat eine Oberklasse, deren Felder und Methoden sie erbt. Die Oberklasse aller Klassen in Java ist die Klasse `java.lang.Object` (siehe dazu Kapitel 2, *Abgeleitete Klassen und Vererbung*).

Beim Definieren von Klassen ist zu beachten, dass eine Klasse eine in sich funktionierende Einheit darstellt, die alle benötigten Felder und Methoden definiert.

Konstruktoren

Für die Initialisierungen eines Objekts der Klasse werden Konstruktoren genutzt. Diese sind eine spezielle Art von Methoden. Sie haben den gleichen Namen wie die Klasse, zu der sie gehören, und verfügen über keinen Rückgabewert. Weil der `new`-Operator eine Referenz auf das erzeugte Objekt zurückgibt, ist eine zusätzliche Rückgabe von Werten in Konstruktoren nicht mehr erforderlich. Jede Klasse besitzt einen impliziten Konstruktor, der auch als Standard-Konstruktor bezeichnet wird. Dieser hat eine leere Parameterliste und übernimmt das Initialisieren der Instanzfelder mit den Defaultwerten der jeweiligen Datentypen. Eine Klasse kann mehrere explizite Konstruktoren definieren, die sich durch ihre Parameterlisten unterscheiden. Der parameterlose Konstruktor wird nur dann vom Compiler generiert, wenn die Klasse keinen expliziten Konstruktor definiert. Ist dies jedoch der Fall und die Klasse möchte auch den parameterlosen Konstruktor benutzen, so muss dieser ebenfalls explizit definiert werden.

Öffentliche (`public`) Konstruktoren werden von Klassen angeboten, damit auch ihre Benutzer, in der Literatur häufig als Clients bezeichnet, Instanzen davon erzeugen können. In diesem Zusammenhang wird immer öfter darauf hingewiesen (insbesondere wenn es um den produktiven Einsatz von Klassen geht), dass noch eine weitere Möglichkeit besteht, Instanzen von Klassen zu erzeugen, indem die Klasse eine oder mehrere statische Factory-Methoden zur Verfügung stellt. Laut Definition sind dies Klassenmethoden, die eine Instanz der Klasse zurückgeben. Ein Beispiel dafür sind die `valueOf()`-Methoden der Wrapper-Klassen für primitive Datentypen, wie `Integer`, `Double` etc. Diese Methoden müssen nicht unbedingt bei jedem Aufruf ein Objekt zurückliefern. Damit wird das unnötige Erzeugen von identischen Objekten vermieden und Klassen die Möglichkeit gegeben, mit bereits konstruierten Objekten zu arbeiten, indem diese abgespeichert und von den Methoden wiederholt zurückgegeben werden.

Der große Nachteil von derartigen Methoden, der auch beim Erstellen der Beispiellklassen aus diesem Buch berücksichtigt wurde, ist jedoch, dass Klassen, die über keine `public` oder `protected` Konstruktoren verfügen, nicht erweitert werden können. Auch wenn damit Programmierer aufgefordert werden, des Öfteren Komposition anstelle von Vererbung (siehe dazu Kapitel 2) zu benutzen, ist und bleibt das Vererben von Klassen ein wesentlicher Bestandteil der Programmiersprache Java. Um »nahe an der Programmiersprache« alle Einzelheiten von Java mit möglichst einfachen Beispielen zu erlernen, brauchen wir Klassen, die das Ableiten

zulassen und somit ihren Unterklassen den Aufruf der Konstruktoren von Oberklassen ermöglichen.

Klassenfelder und Klassenmethoden

Klassenfelder gehören nicht zu einzelnen Objekten, sondern zu der Klasse, in der sie definiert wurden. Alle durchgeführten Änderungen ihrer Werte werden von der Klasse und allen ihren Objekten gesehen. Jedes Klassenfeld ist nur einmal vorhanden. Darum sollten Klassenfelder benutzt werden, um Informationen, die von allen Objekten der Klasse benötigt werden, zu speichern. Diese Felder können direkt über den Klassennamen angesprochen werden und stehen zur Verfügung, bevor irgendein Objekt der Klasse erzeugt wurde.

Klassenmethoden können ebenfalls über den Klassennamen angesprochen werden.

Innerhalb der eigenen Klasse können alle Klassenfelder und Klassenmethoden auch ohne Klassennamen angesprochen werden, sollten aber, um den Richtlinien der objektorientierten Programmierung zu genügen, möglichst mit diesem verwendet werden.

Instanzfelder und Instanzmethoden

Instanzfelder sind mehrfach vorhanden, da für jedes Objekt eine Kopie von allen Instanzfeldern einer Klasse erstellt wird. Die Instanzen einer Klasse unterscheiden sich voneinander durch die Werte ihrer Instanzfelder. Innerhalb einer Klasse kann der Zugriff darauf direkt über ihren Namen erfolgen oder in der Form `this.name` bzw. `obj.name` (wobei `obj` eine Referenz auf ein Objekt der Klasse ist).

Das Schlüsselwort `this` bezeichnet die Referenz auf das »aktuelle Objekt« der Klasse, auch das »aufrufende Objekt« genannt. Damit ist ein Zugriff auf Objekteigenschaften (in Instanzfeldern gespeichert) jederzeit möglich. Konstruktoren werden mit dem Schlüsselwort `new` aufgerufen und innerhalb eines anderen Konstruktors über das Schlüsselwort `this`, gefolgt von der Parameterliste.

Java-Programme

Die Java-Programmtechnologie basiert auf die Zusammenarbeit von einem Compiler und einem Interpreter. Die Programme werden zuerst kompiliert, was einer syntaktischen Prüfung und der Erstellung von Bytecode entspricht. Es entsteht dadurch noch kein ausführbares Programm, sondern ein plattformunabhängiger Code, der an einen Interpreter, die virtuelle Java-Maschine (JVM), übergeben wird. Die JVM ist ein plattformspezifisches Programm, das Bytecode lesen, interpretieren und ausführen kann.

Ein Java-Programm manipuliert Werte, die durch Typ und Name gekennzeichnet werden. Über die Festlegung von Name und Typ wird eine Variable definiert. Man spricht von Variablen in Zusammenhang mit einem Programm und von Feldern in Zusammenhang mit einer Klassendefinition.

Eine Variable ist im Grunde genommen ein symbolischer Name für eine Speicheradresse. Während für primitive Variablen der Typ des Werts, der an dieser Adresse gespeichert wird, gleich dem Typ des Namens der Variablen ist, wird im Falle einer Referenzvariablen nicht der Wert von einem bestimmten Objekt an dieser Adresse gespeichert, sondern die Angabe, wo das Programm den Wert (das Objekt) von diesem Typ finden kann.

Im Gegensatz zu lokalen Variablen, die keinen Standardwert haben und deswegen nicht verwendet werden können, bevor sie nicht explizit initialisiert wurden, werden alle Felder in einer Klassendefinition automatisch mit Defaultwerten initialisiert (mit 0, 0.0, false primitive Typen und mit null Referenztypen).

Die Definition einer Referenzvariablen besteht aus dem Namen der Klasse bzw. eines Interface gefolgt vom Namen der Variablen. Eine so definierte Referenzvariable kann eine Referenz auf ein beliebiges Objekt der Klasse oder einer Unterklasse oder den Defaultwert null aufnehmen. Weil Arrays als Objekte implementiert werden, müssen Arrays mit einem Array-Initialisierer oder mit dem new-Operator erzeugt werden.

Bevor ein Programm Objekte von einer Klasse bilden kann, wird diese mit dem Java-Klassenlader (Klasse `java.lang.ClassLoader`) geladen und mit dem Java-Bytecode-Verifier geprüft.

Nach der Art der Ausführung existieren mehrere Arten von Java-Programmen:

- Ein Java-Applet ist ein Java-Programm, das im Kontext eines Webbrowsers mit bestimmten Sicherheitseinschränkungen abläuft. Es wird mittels einer HTML-Seite gestartet und kann im Browser oder mithilfe des Appletviewers ausgeführt werden. Applets wurden mit der Version 9 von Java als deprecated gekennzeichnet und werden nicht mehr weiterentwickelt.
- Ein Servlet ist ein Java-Programm, das im Kontext eines Webserver abläuft.
- Eine Java-Applikation ist ein eigenständiges Programm, das direkt von der JVM gestartet wird. Alle nachfolgenden Beispiele werden als Java-Applikationen präsentiert.

Jede Java-Applikation benötigt eine `main()`-Methode, die einen Eingangspunkt für die Ausführung des Programms durch die JVM definiert. Diese Methode muss für alle Klassen der JVM zugänglich sein und deshalb mit dem Modifikator `public` definiert werden. Sie muss auch mit dem Modifikator `static` als Klassenmethode deklariert werden, da ein Aufruf dieser Methode möglich sein muss, ohne dass eine Instanz der Klasse erzeugt wurde. Von hier aus werden alle anderen Programmabläufe gesteuert.

Auf die Definition der Parameterliste der `main()`-Methode wird in nachfolgenden Programmbeispielen eingegangen.

Gleich in den ersten Beispielen werden für Bildschirmausgaben die Methoden `System.out.print(...)` und `System.out.println(...)` der Klasse `java.io.PrintStream` verwendet. Mit der Methode `System.out.print(...)` wird in der

gleichen Bildschirmzeile weitergeschrieben, in der eine vorangehende Ausgabe erfolgte. Die Methode `System.out.println()` ohne Parameter schließt eine vorher ausgegebene Bildschirmzeile ab und bewirkt, dass danach in eine neue Zeile geschrieben wird. Ein Aufruf der Methode `System.out.println(...)` mit Parameter ist äquivalent mit dem Aufruf von `System.out.print(...)` gefolgt von einem Aufruf von `System.out.println()` ohne Parameter, das heißt, dieser Aufruf führt immer zu einem Zeilenende. Auf die Definition von diesen Methoden kommen wir, in der Beschreibung der Java-Standard-Klasse `java.lang.System` noch einmal zurück.

Ein Programm wird als Quelltext in einer oder mehreren `.java`-Dateien und als übersetztes Programm in einer oder mehreren `.class`-Dateien abgelegt.

Aufgabe 1.1



Definition einer Klasse

Definieren Sie eine Klasse `KlassenDefinition`, die die `main()`-Methode als einzige Klassenmethode implementiert. Aus dieser soll die Bildschirmanzeige »Dies ist eine einfache Klassendefinition« erfolgen.

Hinweise für die Programmierung

Ein Erzeugen von Instanzen der Klasse ist nicht erforderlich.

Achten Sie auf den richtigen Abschluss der Ausgabezeile.

Java-Dateien: `KlassenDefinition.java`

Programmaufruf: `java KlassenDefinition`

Aufgabe 1.2



Objekt (Instanz) einer Klasse erzeugen

Definieren Sie eine Klasse `ObjektInstanzierung`, die in einem parameterlosen Konstruktor die Bildschirmanzeige »Instanz einer Java-Klasse erzeugen« vornimmt und in ihrer `main()`-Methode eine Instanz der Klasse erzeugt.

Java-Dateien: `ObjektInstanzierung.java`

Programmaufruf: `java ObjektInstanzierung`

1.2 Das Überladen von Methoden

Eine Klasse kann mehrere Methoden mit gleichem Namen besitzen, wenn diese eine verschiedene Anzahl von Parametern bzw. Parameter von unterschiedlichen Typen im Methodenkopf definieren. Dabei ist ohne Bedeutung, ob es sich um Klassen- oder Instanzmethoden handelt.

Parallel zur Parameterliste unterscheidet sich auch die Aufrufsyntax der Methode. Dieses Konzept ist unter dem Namen »Überladen von Methoden« bekannt.

Aufgabe 1.3



Eine Methode überladen

Definieren Sie eine Klasse `QuadratDefinition`, die ein Instanzfeld `a` vom Typ `int` besitzt, das die Seitenlänge eines Quadrats angibt. Im Konstruktor der Klasse wird ein `int`-Wert zum Initialisieren des Instanzfelds übergeben.

Implementieren Sie zwei Methoden für die Berechnung des Flächeninhalts eines Quadrats mit der Formel $f = a \cdot a$. Definieren Sie eine parameterlose Instanzmethode `flaeche()` und eine Klassenmethode, die die Instanzmethode überlädt und eine Referenz vom Typ der eigenen Klasse übergeben bekommt.

Die Klasse `QuadratDefinitionTest` erzeugt eine Instanz der Klasse `QuadratDefinition`, berechnet auf zwei Arten deren Flächeninhalt über den Aufruf der Methoden der Klasse und zeigt die errechneten Ergebnisse am Bildschirm an.

Java-Dateien: `QuadratDefinition.java`, `QuadratDefinitionTest.java`
Programmaufruf: `java QuadratDefinitionTest`

1.3 Die Datenkapselung, ein Prinzip der objektorientierten Programmierung

Den Feldern und Methoden einer Klasse können über Modifikatoren verschiedene Sichtbarkeits Ebenen zugeordnet werden.

Der bereits erwähnte Modifikator `public` sagt aus, dass der Zugriff auf Member einer Klasse von überall aus erfolgen kann, von wo aus auch die Klasse erreichbar ist.

Sind die Felder oder Methoden mit `private` definiert, können sie nur innerhalb der eigenen Klasse direkt angesprochen werden. Felder sollten immer als `private` definiert werden, wenn die Zuweisung von unzulässigen Werten verhindert werden soll. Dies ist der Fall, wenn sie von einer eigenen Methode der Klasse, die diesen Wert auch ändern kann, verwendet oder weitergegeben werden.

Definiert die Klasse keine Einschränkungen diesbezüglich oder einen zugelassenen Wertebereich für Felder innerhalb, von dem auch andere Klassen Werte setzen können, sollte sie über Zugriffsmethoden (»accessor-methods«) verfügen, die die Werte dieser Felder zurückgeben und ggf. setzen können. Dies entspricht dem sogenannten Prinzip der Datenkapselung: Auf die Felder einer Klasse soll nur mithilfe von Methoden der Klasse zugegriffen werden können.

Aufgabe 1.4



Zugriffsmethoden

Definieren Sie eine Klasse `Punkt` mit zwei Instanzfeldern vom Typ `double`, die die Koordinaten `x` und `y` eines Punkts im zweidimensionalen kartesischen Koordinatensystem beschreiben. Sie sollen von außerhalb der Klasse nur über die von Ihnen definierten Zugriffsmethoden `setX()`, `setY()`, `getX()` und `getY()` zugänglich sein und werden im Konstruktor der Klasse übergeben. Fügen Sie der Klasse eine zusätzliche Instanzmethode `anzeige()` für eine Punktanzeige am Bildschirm in der Form `(x, y)` hinzu.

Definieren Sie zum Testen der Klasse `Punkt` eine zweite Klasse `PunktTest`, die in ihrer `main()`-Methode eine Instanz der Klasse `Punkt` erzeugt und an dieser die Methoden der Klasse aufruft.

Java-Dateien: `Punkt.java`, `PunktTest.java`
 Programmaufruf: `java PunktTest`

1.4 Das »aktuelle Objekt« und die »this-Referenz«

In jedem Konstruktor und in jeder Instanzmethode kann das aktuelle (aufrufende) Objekt der Klasse in Form einer `this`-Referenz angesprochen werden. Ein Konstruktoraufruf aus einem anderen Konstruktor erfolgt über `this(parameterListe)` und muss der zuerst erreichte übersetzte Programmcode in diesem Konstruktor sein. Aus anderen Methoden kann ein Konstruktor nicht über `this` aufgerufen werden, sondern nur mit dem `new`-Operator.

Aufgabe 1.5



Konstruktordefinitionen

Erstellen Sie eine Java-Klasse mit dem Namen `Vektor`, die drei Instanzfelder `x`, `y` und `z` definiert, die die Komponenten eines Vektors bezeichnen. Die Klasse definiert drei Konstruktoren:

- den parameterlosen Konstruktor,
- einen Konstruktor, der drei Argumente vom Typ `int` mit den gleichen Namen wie die der Instanzfelder übergeben bekommt
- und den sogenannten Copy-Konstruktor, der als Parameter eine Referenz vom Typ der eigenen Klasse besitzt.

Der parameterlose Konstruktor soll über den Aufruf des zweiten Konstruktors alle Instanzfelder der Klasse auf 0 setzen.

Die Klasse soll über eine Methode für die Bildschirmanzeige eines `Vektor`-Objekts in der Form `(x, y, z)` verfügen.

Definieren Sie zwei weitere Methoden, die sich überladen, zum Erzeugen eines neuen `Vektor`-Objekts, das als Summe der aktuellen Instanz und einer übergebenen berechnet wird und deren Rückgabewert die aktuelle Instanz ist. Die erste Methode soll drei Parameter vom Typ `int` besitzen, die zweite Methode einen Parameter vom Typ `Vektor`.

Soll das ursprüngliche Objekt nicht verloren gehen, kann eine Kopie davon erzeugt werden. Eine dritte Methode im Lösungsvorschlag der Aufgabe berechnet die gleiche Summe, ohne dass die Instanz, an der die Methode aufgerufen wird, abgeändert wird. Bei gleicher Parameterliste muss die Methode über einen neuen Namen verfügen.

Zum Testen der Klasse `Vektor` soll eine zweite Klasse `VektorTest` erstellt werden, die in ihrer `main()`-Methode Instanzen der Klasse mithilfe ihrer Konstruktoren erzeugt und ihre Methoden aufruft.

Java-Dateien: `Vektor.java`, `VektorTest.java`

Programmaufruf: `java VektorTest`

1.5 Die Wert- und Referenzübergabe in Methodenaufrufen

In Java-Methoden werden alle Argumente, ob es Werte von primitiven Typen oder Referenzen sind, als Kopie per Wert übergeben. Der Mechanismus der Wertübergabe wird auch »call by value« bzw. »copy per value« genannt. Wenn ein Argument übergeben wird, wird dessen Wert an eine Speicheradresse in den Stack der Methodenaufrufe (»method call stack«) kopiert. Egal ob dieses Argument eine Variable von einem primitiven oder Referenztyp ist, wird der Inhalt der Kopie als Parameterwert übergeben und nur diese kann innerhalb der Methode abgeändert werden, nicht der Wert selbst. Das heißt, eine Parametervariable wird als lokale Variable betrachtet, die zum Zeitpunkt des Methodenaufrufs mit dem entsprechenden Argument initialisiert wird und nach dem Beenden der Methode nicht mehr existiert.

Eine Argumentübergabe per Referenz, auch »call by reference« genannt, wie sie in anderen Programmiersprachen verwendet wird, gibt es in Java nicht. Für die Übergabe von Objekten werden zwar Referenzen vom Typ der Objekte als Parameter für Methoden definiert, doch werden diese, wie vorher beschrieben, kopiert. Aus diesem Grund ist in der Java-Literatur oft zu lesen: »In Java werden Objekte per Referenz und Referenzen per Wert übergeben.«

Aufgabe 1.6



Wertübergabe in Methoden (»call by value«)

Die Klasse `MethodenParameter` definiert drei Klassenmethoden mit den Signaturen `public methode1(int x, int[] y)`, `public methode2(Punkt x, Punkt[] y)` und `public methode3(Punkt x)`, wobei `Punkt` die Klasse aus der Aufgabe 1.4 bezeichnet.

Rufen Sie aus der `main()`-Methode der Klasse alle drei Methoden auf und zeigen Sie die Werte der von Ihnen übergebenen primitiven, Array- und Referenz-Typen vor und nach den Methodenaufrufen am Bildschirm an.

Hinweise für die Programmierung

Um festzustellen, wie die Übergabe in Methodenaufrufen erfolgt, soll durch Zuweisungen und den Aufruf von Zugriffsmethoden der Klasse `Punkt` ein Teil der im Methodenaufruf übergebenen Werte verändert werden.

Java-Dateien: `MethodenParameter.java`
Programmaufruf: `java MethodenParameter`

1.6 Globale und lokale Referenzen

Alle bisherigen Programme haben Referenzvariablen als lokale Referenzen in Methoden oder als deren Parametervariablen definiert. Instanz- und Klassenfelder von einem Referenztyp werden auch als globale Referenzen bezeichnet.

Aufgabe 1.7



Der Umgang mit Referenzen

Definieren Sie eine Klasse `GlobalReferenzen`, die anstelle der lokalen Variablen aus den Methoden der Klasse `MethodenParameter` globale Programmvariablen definiert und die Methoden selbst ohne Parametervariablen.

Hinweise für die Programmierung

Referenzparameter von Methoden können im Prinzip durch globale Referenzen der Klasse ersetzt werden, nur sind die darauf durchgeführten Änderungen innerhalb von Methoden auch nach außen sichtbar. Dabei macht es kein Unterschied, ob die globalen Referenzen als Klassen- bzw. Instanzfelder definiert wurden.

Referenzparameter von Methoden können im Prinzip durch globale Referenzen der Klasse ersetzt werden, nur sind die darauf durchgeführten Änderungen innerhalb von Methoden auch nach außen sichtbar. Dabei macht es kein Unterschied, ob die globalen Referenzen als Klassen- bzw. Instanzfelder definiert wurden.

Java-Dateien: `GlobalReferenzen.java`
Programmaufruf: `java GlobalReferenzen`

1.7 Java-Pakete

Die erstellten Java-Klassen können in Pakete (»packages«) zusammengefasst werden, die als eigene Klassenbibliotheken dienen. Jedes Paket definiert eine eigene Umgebung für die Namensvergabe von Klassen, um Konflikte zu unterbinden, die bei einer Vergabe von gleichen Namen auftreten könnten.

Ein Programm wird in ein Paket oder dessen Subpakete über eine `package paketname1[.paketname2...]`-Anweisung integriert, die am Anfang des Sourcecodes stehen muss. Paketnamen sind im Grunde genommen Bezeichnungen von Dateiverzeichnissen, in die die Java-Dateien hinterlegt werden.

Immer wenn ein Klassenname in einem Programm auftritt, muss der Compiler das Paket identifizieren können, in dem sich diese Klasse befindet. Dazu dient die Anweisung `import paketname.klassenname;` von Java.

Die Namen von Klassen und deren Paketen werden vom Compiler in die bereits vorher erwähnte Klassendatei, die mit dem Suffix `.class` gespeichert wird, eingetragen. Diese Datei ist eine Unterstützung für den JVM-Klassenlader beim Auffinden der Klasse. Eine zusätzliche Hilfe ist auch die Umgebungsvariable `CLASSPATH`, die eine Liste von Dateiverzeichnissen und Namen von Archivdateien für die Suche zur Verfügung stellen kann. Archivdateien sind Dateien, die selbst andere Dateien beinhalten, und werden in Java mit dem Suffix `.jar` abgeschlossen. Unter Windows wird die `CLASSPATH`-Variable über das Betriebssystemkommando: `set classpath = c:\pfadname1;pfadname2;archivname1;...` gesetzt und mit `set classpath = .;` gelöscht.

Ist die Umgebungsvariable nicht gesetzt, so sucht der Klassenlader nach einer Klasse im aktuellen Verzeichnis oder in einem Verzeichnis, das den ersten Paketnamen in einer angegebenen `import`-Anweisung trägt, danach in einem Verzeichnis, das den zweiten Paketnamen trägt, etc. Ist eine Umgebungsvariable gesetzt, werden ihre Einträge von links nach rechts nach einem Verzeichnis oder einer Archivdatei, die entweder die Datei oder den ersten Paketnamen enthalten, durchsucht.

Beide Arten der Suche werden so lange fortgesetzt, bis eine Klasse gefunden und geladen wird, ansonsten wird die Fehlermeldung: `"no class definition found"` ausgegeben.

Die meisten bis jetzt verwendeten Klassen wurden ohne Modifikator definiert. Eine Klasse ohne `public` ist nicht uneingeschränkt öffentlich, es können nur Klassen aus dem gleichen Paket, in dem sich die Klasse befindet, Instanzen davon erzeugen. Darum wird dieser Zugriffsschutz auch als »package private« bezeichnet. Eine Klasse hat nur zwei Zugriffsebenen: `standard` (ohne Modifikator) und `public`. Eine mit `public` definierte Klasse ist für alle anderen Klassen zugänglich und muss immer in einer Java-Datei mit gleichem Namen gespeichert werden.

Aufgabe 1.8



Die package-Anweisung

Für ein Dateiverzeichnis `kapitel1` wird ein Unterverzeichnis `paket1` definiert. Erstellen Sie eine Klasse `PackageTest`, die in ihrer `main()`-Methode die Zeichenkette `"Test der package-Anweisung"` am Bildschirm ausgibt, und speichern Sie diese als die Java-Datei `PackageTest.java` im Verzeichnis `paket1` ab. Wie muss

die `package`-Anweisung in dieser Klassendefinition lauten, damit das Programm im Verzeichnis `kapitel1` übersetzt und ausgeführt werden kann?

Hinweise zu den Programmaufrufen

Ist das Verzeichnis `paket1` nicht mit der `CLASSPATH`-Umgebungsvariablen gesetzt, so muss es beim Übersetzen als Dateiverzeichnisname angegeben werden: `javac paket1\PackageTest.java`. Wird im Sourcecode die Anweisung `package paket1;` angegeben, kann für die Programmausführung der Paketname dem Klassennamen vorangestellt werden: `java paket1.PackageTest`.

Java-Dateien: `kapitel1\paket1\PackageTest.java`

Programmaufrufe im Verzeichnis `kapitel1`: `javac paket1\PackageTest.java` und `java paket1.PackageTest` oder `java paket1/PackageTest`

Aufgabe 1.9



Die import-Anweisung

Im Verzeichnis `paket1` wird ein weiteres Unterverzeichnis `paket2` hinterlegt. Definieren Sie eine Klasse mit dem Namen `Klasse`, die die Anweisung `package paket2;` beinhaltet und in ihrer `main()`-Methode die Zeichenkette "Definition einer Klasse im Verzeichnis `paket2`" am Bildschirm ausgibt. Soll diese Klasse aus einem externen Paket angesprochen werden, muss sie als `public` definiert werden. Speichern Sie diese Klasse als Java-Datei im Verzeichnis `paket2` ab.

Definieren Sie eine weitere Klasse `KlassenTest`, die als Java-Datei im Verzeichnis `paket1` gespeichert ist und eine Instanz der Klasse `Klasse` erzeugt.

Das mit der Klasse `KlassenTest` erstellte Java-Programm soll im Verzeichnis `paket1` übersetzt und ausgeführt werden.

Die Verwendung des Klassennamens `Klasse` kann entweder über eine `import`-Anweisung erfolgen oder es muss das Präfix `paket2.` beim Übersetzen und Ausführen des Programms angegeben werden.

Java-Dateien: `kapitel1\paket1\KlassenTest.java`,

`kapitel1\paket1\paket2\Klasse.java`

Programmaufrufe im Verzeichnis `kapitel1`: `javac paket1\KlassenTest.java` und `java paket1/KlassenTest`

1.8 Die Modifikatoren für Felder und Methoden in Zusammenhang mit der Definition von Paketen

Auf ein Member einer Klasse, das ohne Modifikator definiert wurde, kann von außerhalb eines Pakets nicht zugegriffen werden. Nur mit `public` deklarierte Member sind uneingeschränkt öffentlich. Ein mit `protected` definiertes Member ist außerhalb eines Pakets nur für abgeleitete Klassen einer Klasse sichtbar. Weitere

Ergänzungen zu diesen Aussagen können in Kapitel 2, *Abgeleitete Klassen und Vererbung* (Abschnitt 2.3), gelesen werden.

Kapitel 7 aus diesem Buch beschäftigt sich mit dem neuen Modulsystem von Java und kommt nochmals in Abschnitt 7.2 auf die Sichtbarkeits Ebenen innerhalb von Paketen im Zusammenhang mit Modulen zurück.

Aufgabe 1.10



Pakete und die Sichtbarkeit von Mitgliedern einer Klasse

Die Klassen aus diesen Programmbeispielen sollen als Test der `import`-Anweisung für Pakete dienen, die Subpakete beinhalten. Definieren Sie zu diesem Zweck eine Klasse `PackageTest1`, deren Programmdatei im Verzeichnis `kapitel1` abgelegt ist und Instanzen von zwei weiteren Klassen, `Klasse1` und `Klasse2` erzeugt, die in den Unterverzeichnissen `paket1` und `paket2` von `kapitel1` in Programmdateien mit gleichem Namen abgelegt werden.

Die Klasse `Klasse1` definiert drei Klassenfelder vom Typ `int`: `privatesFeld`, `geschuetztesFeld`, `oeffentlichesFeld` mit den Modifikatoren `private`, `protected`, `public` und ein weiteres Klassenfeld `feld` ohne Modifikator. Sie soll die Zeichenkette "Instanz der Klasse1" am Bildschirm anzeigen und den Paketnamen über die Anweisung: `package paket1`; angeben. Die Klasse `Klasse2` soll die Zeichenkette "Instanz der Klasse2" am Bildschirm anzeigen und die Anweisung: `package paket2`; für die Angabe des Paketnamens definieren.

In der Klasse `PackageTest1` sollen beide Paketnamen über eine `import`-Anweisung bekannt gegeben werden und soweit möglich die Werte der in `Klasse1` definierten Felder am Bildschirm angezeigt werden.

Das Java-Programm `PackageTest1.java` soll im Verzeichnis `kapitel1` übersetzt und ausgeführt werden.

Java-Dateien: `kapitel1\PackageTest1.java`, `kapitel1\paket1\Klasse1.java`, `kapitel1\paket1\paket2\Klasse2.java`

Programmaufrufe im Verzeichnis `kapitel1`: `javac PackageTest1.java` und `java PackageTest1`

1.9 Standard-Klassen von Java

Von großer Bedeutung in der Programmierung mit Java sind seine Standard-Klassen, die die sogenannte Java-API bilden. Die Standard-Klassen von Java sind in Pakete gebündelt, wie z.B. `java.lang`, `java.io`, `java.util` etc. Das Java-Paket `java.lang` beinhaltet die Klassen, die die Basis der Java-Programmiersprache bilden, wie `Object`, `System`, `Process`, `ProcessBuilder`, `Runtime`, `Math`, `Class<T>` etc. Diese Klassen werden automatisch vom Compiler importiert, dafür ist keine `import`-Anweisung nötig.

Zur Identifikation einer Klasse muss bei allen Paketen außer `java.lang` der Paketname dem Klassennamen vorangestellt werden, wie z.B. mit `import java.util.List`. Sollen alle Klassen eines Pakets importiert werden, geschieht dies über einen Stern statt über den Klassennamen, wie z.B. `import java.util.*`.

Mit der Version 9 von Java wurde die Modularisierung als Spracherweiterung eingeführt, mit der Pakete in Module zusammengefasst werden. Gleichzeitig wurden Aktivitäten rund um die Modularisierung der Java-Plattform selbst gestartet und die JDK und JRE modularisiert. Wie bereits erwähnt, können die Neuerungen, die das Modulsystem von Java mit sich bringt, in Kapitel 7 gelesen und eingeübt werden.

Die Klasse System

Die Klasse `System` kann nicht instanziiert werden, da sie keinen Konstruktor besitzt. Sie besitzt aber eine Vielzahl von nützlichen Klassenfeldern und Klassenmethoden. Dazu zählen Felder, die den Zugriff auf die Standardein- und Standardausgabe erlauben, und andere, die Systemeigenschaften und Umgebungsvariablen definieren.

Diese Klasse definiert als Klassenfelder die globalen Referenzen `in`, `out` und `err`, die auf Objekte vom Typ der Klassen `InputStream` und `PrintStream` aus dem Paket `java.io` verweisen, die Ein- und Ausgaben zu den Standardgeräten (in der Regel die Konsole) leiten. Das Objekt, auf das die Referenz `err` der Klasse `PrintStream` zeigt, wird für die Ausgabe von Fehlermeldungen benutzt. Alle drei Instanzen werden beim Programmaufruf erzeugt, mit den Standardgeräten verbunden und stehen jederzeit dem Programmierer zur Verfügung.

Die Instanz, auf die die Referenz `out` zeigt, wird auch mit den Methoden `System.out.print(...)` und `System.out.println(...)` der Klasse `PrintStream` genutzt, um Bildschirmausgaben zu realisieren. Wir haben diese bereits in den vorangegangenen Aufgaben verwendet.

Eine Liste mit allen Systemeigenschaften kann mit der Methode `getProperties()` und die Liste der im System gesetzten Umgebungsvariablen mit der Methode `getenv()` der Klasse `System` abgerufen werden.

Die Klasse File

Die Verwaltung von Dateien und deren Verzeichnissen wird in Java u.a. von der Klasse `File` aus dem Paket `java.io` übernommen. Ein `File`-Objekt ist eine abstrakte Repräsentation einer Datei oder eines Verzeichnisses. Die Klasse `java.nio.Files` (Java 7) stellt zusätzliche Methoden zur Verfügung, mit denen Informationen über Dateien und Verzeichnisse geholt werden können.

Mit der Methode `write()` der `java.io.FileWriter`-Klasse kann ein Text (als Stream von `Characters`) in eine Datei geschrieben werden. Dabei werden die `Characters` in `Bytes` decodiert. Um direkt `Bytes` in eine Datei zu schreiben, kann ein Stream vom Typ `FileOutputStream` benutzt werden.

Die Klassen `Runtime`, `ProcessBuilder` und `Process`

Beim Ausführen eines Programms mithilfe des `java`-Kommandos wird eine JVM gestartet und vom Betriebssystem dazu ein eigener Prozess erzeugt. Es können auch mehrere JVMs gestartet werden, die entsprechenden Prozesse laufen dann parallel, wobei jeder Prozess seinen eigenen Adressraum besitzt.

Eine Instanz der Klasse `Runtime` repräsentiert die Laufzeitumgebung einer Java-Anwendung und kann über den Aufruf der Methode `getRuntime()` der Klasse ermittelt werden. Über dieses Objekt kann die aktuell laufende JVM mit der Methode `exit()` beendet werden. Dies ist auch über die gleichnamige Methode der Klasse `System` möglich und wird über deren Aufruf `System.exit()` eingeleitet.

Über den Aufruf der Methode `Process exec(String name)` der Klasse, wobei `name` den Namen einer `.exe`-Datei spezifiziert und der Rückgabewert vom Typ `Process` ist, kann eine andere Anwendung gestartet werden.

Die Klasse `ProcessBuilder` wurde mit Java 5.0 eingeführt und kann alternativ zur Klasse `Runtime` zum Starten eines Prozesses des Betriebssystems genutzt werden.

Ein Objekt der Klasse `Process` kann durch einen der Methodenaufrufe `Runtime.exec` bzw. `ProcessBuilder.start()` erzeugt werden und repräsentiert einen Prozess des Betriebssystems.

Die Klassen `Exception` und `Error`

Exceptions sind Ausnahmesituationen, die zur Laufzeit eines Programms auftreten und seinen Ablauf unterbrechen können. Diese können behandelt werden, sodass ein Programmabbruch dabei vermieden wird. Errors sind schwerwiegende Fehler, eine weitere Ausführung des Programms ist bei deren Auftreten meistens nicht mehr gerechtfertigt.

Exceptions werden über das Schlüsselwort `throw` ausgelöst und können von einem `catch`-Block aufgefangen werden, in dem deren Verarbeitung erfolgen kann. Dazu werden die Anweisungen, die Exceptions auslösen, zu einem `try`-Block zusammengefasst und diesem wird ein `catch`-Block nachgestellt. Exceptions kann man zur Behandlung auch weitergeben, indem sie mit dem Schlüsselwort `throws` im Methodendefinitionskopf durch Komma getrennt aufgelistet werden. Wird eine Exception auch von der `main()`-Methode mit `throws` weitergeworfen, so wird diese nicht mehr von der Applikation behandelt und die Applikation wird mit einer entsprechenden Fehlermeldung beendet.

Diese Klassen und ihre Unterklassen befinden sich auch im Paket `java.lang` und werden in Kapitel 5, *Exceptions und Errors*, ausführlicher behandelt.

1.10 Die Wrapper-Klassen von Java und das Auto(un)boxing

Sowohl primitive Datentypen als auch Referenztypen legen die Eigenschaften ihrer Werte innerhalb von Klassen in Form von Felddefinitionen fest. Für die Manipulation der Werte stehen für primitive Datentypen Operatoren zur Verfügung, wäh-

rend Klassen Methoden benutzen. Operatoren sind von der Programmiersprache her vordefiniert und können in Java vom Programmierer nicht überladen werden. Methoden bringen den Vorteil mit sich, dass diese ein Überladen erlauben, ohne dass eine bestimmte Anzahl davon vorgegeben wird (siehe dazu die Aufgabe 1.3).

Weil höhere Datenstrukturen wie z. B. Collections nur Objekte aufnehmen können, wurden Hüllenklassen, auch Wrapper-Klassen genannt, für alle primitiven Datentypen definiert: `Boolean`, `Byte`, `Integer`, `Float`, `Double`, `Long`, `Short` und `Character`. Während die primitiven Datentypen Bestandteil der Java-Programmiersprache sind, gehören die Wrapper-Klassen zur Java-API. Eine Hüllenklasse definiert ein Instanzfeld vom entsprechenden Datentyp und Methoden, mit denen dieses manipuliert werden kann. Dies sind z. B. Konvertierungsmethoden wie `toBinaryString()`, `toHexString()` oder Methoden für die Rückgabe des primitiven Werts des Wrapper-Objekts, wie `intValue()`, `floatValue()` etc. Alle Hüllenklassen definieren einen Konstruktor, der ein Argument vom Typ des primitiven Datentyps besitzt und einen Konstruktor mit einem Argument vom Typ der Klasse `String`. Konstruktoren, in denen ein Argument vom Typ des zugehörigen primitiven Datentyps übergeben werden kann, wurden mit Java 9 als deprecated gekennzeichnet, um das Erzeugen einer Vielzahl von unnötigen Objekten zu vermeiden. An deren Stelle wird in der Dokumentation empfohlen, die `valueOf()` von Wrapper-Klassen aufzurufen, durch die, wie bereits erwähnt, nicht immer ein neues Objekt erzeugt wird.

Wrapper-Klassen definieren auch eine Reihe von nützlichen Konstanten wie z. B. `MIN_VALUE` oder `MAX_VALUE`, die den kleinsten bzw. größten Wert des entsprechenden arithmetischen Datentyps bezeichnen. Sie definieren jedoch keine Methoden für arithmetische oder logische Operationen zwischen Objekten einer Klasse oder zur Änderung des innerhalb einer Klasse gespeicherten Werts. Diese Klassen sind als `final` deklariert, das heißt, sie sind nicht erweiterbar und werden alle von der abstrakten Klasse `Number` abgeleitet, deren Methoden sie implementieren.

Mit der Version 5.0 von Java erübrigt sich weitgehend das manuelle Umwandeln von primitiven Datentypen in Objekttypen und umgekehrt, da diese Version eine automatische Konvertierung (Auto(un)boxing) für diese Art von Datentypen implementiert und deren Übergabe in Konstruktoren und Methoden damit wesentlich vereinfacht.

Damit wurde ein weiterer wesentlicher Beitrag zur Typsicherheit von Daten gewährleistet, wie auch mit den in Java 5.0 eingeführten generischen Datentypen, die in Kapitel 4, *Generics*, beschrieben werden.

Autoboxing ist der Vorgang, durch den ein primitiver Datentyp wie `int`, `boolean` etc. automatisch in seinen entsprechenden Wrapper-Typ `Integer`, `Boolean` etc. eingehüllt (boxed) wird, das Erzeugen eines neuen Objekts wird automatisch von Java übernommen.

Mit Auto-unboxing wird der Vorgang bezeichnet, durch den der Wert eines Wrapper-Objekts extrahiert wird, ohne dass Methoden wie `intValue()`, `booleanValue()` etc. der entsprechenden Wrapper-Klassen explizit aufgerufen werden müssen.

In der Literatur wird jedoch darauf hingewiesen, dass das Auto(un)boxing, durch das in der Programmierung möglich gemacht wurde, die elementaren numerischen Typen mit ihren Wrapper-Typen zu mischen, mit Bedacht angewandt werden sollte. Um ein Erzeugen von unnötigen Objekten zu vermeiden, sollten vorrangig elementare Typen benutzt werden und immer darauf geachtet werden, dass ein Autoboxing durch unüberlegte Zuweisungen nicht ungewollt ausgeführt wird.

Aufgabe 1.11



Das Auto(un)boxing

Definieren Sie in einer Klasse mit dem Namen `WrapperKlassenmitAutoBoxing` globale Referenzen vom Typ aller Hüllklassen von Java. Übergeben Sie im Konstruktor dieser Klasse die korrespondierenden primitiven Werte und erzeugen Sie mithilfe der `valueOf()`-Methoden je eine Instanz für jede dieser Klassen.

Prüfen Sie, ob einfache Zuweisungen von primitiven Werten für das Bilden der Instanzen von Hüllklassen ausreichend sind und ob im Methodenaufruf von `System.out.println()` ein Unboxing bei der Angabe dieser Instanzen durchgeführt wird. Definieren Sie eine Methode mit der Signatur `public boolean konvert(Boolean b)` und zeigen Sie, dass diese mit einem primitiven Wert vom Typ `boolean` aufgerufen werden kann.

Erstellen Sie anhand des Lösungsvorschlags ein eigenes Beispiel mit Umwandlungen zwischen primitiven Typen und Typen von Wrapper-Klassen im gleichen Ausdruck und nutzen Sie die erweiterte Schreibweise von `if`-Anweisungen für einen Vergleich der Instanzen von Wrapper-Klassen mit dem `»==-Operator«`.

Zum Testen soll eine weitere Klasse `WrapperKlassenmitAutoBoxingTest` erstellt werden.

Hinweise für die Programmierung

Ein Vergleich mit `==` bleibt weiterhin ein Vergleich von Referenzen und dabei wird kein Unboxing auf primitive Datentypen durchgeführt, sodass ein solcher Vergleich zwischen zwei Wrapper-Objekten trotz gleichem numerischem Wert im Allgemeinen das Ergebnis `false` liefert.

Java-Dateien: `WrapperKlassenmitAutoBoxing.java`, `WrapperKlassenmitAutoBoxingTest.java`

Programmaufruf: `java WrapperKlassenmitAutoBoxingTest`

1.11 Das Paket `java.lang.reflect`

Über das Paket `java.lang.reflect` wird die sogenannte Reflection-API als integraler Bestandteil der Java-Klassenbibliothek bereitgestellt. Damit können Informationen zu Klassen- und Objektstrukturen zur Laufzeit ermittelt werden.

Die Klasse `Class<T>` ist nicht Bestandteil des Pakets `java.lang.reflect`. Das einer Klasse zugehörige Klassenobjekt vom Typ der Klasse `Class` bildet jedoch die Basis der Reflection-API, weil viele der reflektiven Betrachtungen das Erzeugen einer Instanz vom Typ dieser Klasse voraussetzen.

Um ein einheitliches Konzept zur Darstellung von Typen in Java zu erlangen, wurden mit der Version 1.1 der Reflection-API auch `Class`-Objekte für primitive Datentypen eingeführt. Diese werden, wie bereits erwähnt, mit `int.class`, `char.class` etc. bezeichnet. Auch für `void` steht ein entsprechendes Klassenobjekt `void.class` zur Verfügung.

Für jede geladene Klasse wird von der JVM genau ein Klassenobjekt erzeugt. Damit ist gewährleistet, dass dieses nur einmal vorhanden ist. Das Klassenobjekt darf nicht mit der Klassendatei (`className.class`), die beim Übersetzen erzeugt wird und den Bytecode der Klasse enthält, verwechselt werden, auch wenn die Möglichkeit besteht, das Klassenobjekt für Referenztypen auf »statische Art« mithilfe des `Class`-Literal zu ermitteln: `Class<String> klsObjekt = String.class`.

Um Objekte von Klassen »dynamisch« zu erzeugen, kann am Klassenobjekt die `newInstance()`-Methode aufgerufen werden.

Eigentlich wird das Erzeugen von Objekten einer Klasse mithilfe des `new`-Operators in Java auch erst zur Laufzeit durchgeführt. Der Compiler muss dabei den Namen der Klasse kennen, um den passenden Konstruktoraufruf zu erzeugen.

Wird der Name einer Klasse erst zur Laufzeit bekannt gegeben, kann der `new`-Operator nicht mehr verwendet werden und es muss auf die `newInstance()`-Methode zurückgegriffen werden. So gesehen ist das Erzeugen eines Objekts in Java immer »dynamisch«, auch wenn von der Ausdrucksweise »Objekte dynamisch erzeugen« eher in Verbindung mit dem Erzeugen von Objekten mithilfe einer `newInstance()`-Methode während des Ladevorgangs von Klassen zur Laufzeit Gebrauch gemacht wird. In der Java-Literatur wird dieser Vorgang auch »Objekte via Reflection erzeugen« genannt.

Das Klassenobjekt liefert mithilfe seiner Methodendefinitionen zusätzlich zu allgemeinen Informationen zu einer Klasse (wie den eigenen Namen, den Namen der Oberklasse und den Namen der implementierten Interfaces) auch alle Felder und Methoden (darunter auch die Konstruktoren) der Klasse.

Auf die Felder und Methoden von so erzeugten Objekten kann während der Laufzeit zugegriffen werden. Dazu können Methoden wie `getField(String name)` oder `getDeclaredField(String name)` bzw. `getMethod(String name, Class<?>... parameterTypes)` am Klassenobjekt der entsprechenden Klasse aufgerufen werden. An einem so ermittelten `java.lang.reflect.Field`-Objekt kann die Methode `get(InstanzKlasse)` aufgerufen werden, die den im Feld gespeicherten Wert zurückgibt. Verschiedene Methoden der Klasse `java.lang.reflect.Method` können deren Rückgabety, für die Methode deklarierte Annotationen etc. liefern.

1.12 Arrays (Reihungen) und die Klassen `Array` und `Arrays`

Ein Array ist ein Objekt, das mehrere Werte von ein und demselben Typ speichern kann. Diese Werte werden auch Elemente oder Komponenten des Arrays genannt. Arrays können sowohl Werte von primitiven als auch von Referenztypen aufnehmen. Die aufeinanderfolgenden Elemente werden in zusammenhängende Speicherbereiche hinterlegt und ein Array hat immer eine feste Länge. Einem Arrayelement wird ein Index zugeordnet, über den es direkt angesprochen werden kann. Der Index des ersten Elements hat in Java den Wert 0.

Mit Referenz vom Typ eines Arrays ist die Referenz auf ein Array-Objekt gemeint. Dieses Array-Objekt enthält alle Elemente des Arrays. Der Typ kann ein primitiver, Klassen- oder Interface-Typ sein.

Ein Array kann in Java mithilfe eines Array-Initialisierers erzeugt werden. So definiert `int[] iArray = {1, 5, 3}`; ein Array vom Typ `int` und `char[] cArray = {'A', 'B'}`; ein Array vom Typ `char`. Die Elemente der so definierten Arrays werden mit den angegebenen Werten initialisiert.

Eine zweite Möglichkeit besteht darin, den `new`-Operator zu benutzen, `int[] iArray = new int[3]`; definiert ein Array von 3 Werten vom Typ `int`, `char[] cArray = new char[2]`; ein Array von 2 Werten vom Typ `char` und `Punkt[] pArray = new Punkt[2]`; ein Array von 2 Objekten vom Typ der Klasse `Punkt` aus der Aufgabe 1.4.

Mit diesen Definitionen wird Speicher für die angegebene Anzahl von Elementen alloziert und alle Bits von Elementen werden auf 0 gesetzt. Der `new`-Operator gibt eine Referenz auf das Array-Objekt zurück.

Im Unterschied zu Arrays von primitiven Typen wird für Arrays von Referenztypen erst einmal nur das Array-Objekt erzeugt und nicht auch Objekte von den einzelnen Elementen. Diese müssen im Nachhinein einzeln mit dem Konstruktor erzeugt werden: `Punkt[0] = new Punkt(1,1)`; und `Punkt[1] = new Punkt(2,2)`;

Beide Definitionsarten können auch kombiniert werden. So definiert `int[] iArray = new int[3]{1, 5, 3}`; ein Array vom Typ `int` und `char[] cArray = new char[2]{A, B}`; ein Array vom Typ `char`.

Die Anzahl der Elemente eines Arrays wird auch Dimension genannt. Java unterstützt auch mehrdimensionale Arrays. Ein zweidimensionales Array wird als Array von eindimensionalen Arrays gebildet und ist somit eine Aneinanderreihung von mehreren eindimensionalen Arrays.

Die Java-Standard-Klasse `Array` ist eine Utility-Klasse, die nur Klassenmethoden enthält, daher kann kein Objekt vom Typ dieser Klasse instanziiert werden. Sie definiert Methoden, die zur Manipulation von beliebigen Array-Objekten und deren Elemente genutzt werden können, und eine Methode `newInstance()`, die ein Objekt vom Typ der Klasse `Object` zurückgibt, auf das alle anderen Methoden der Klasse angewandt werden können. Die Klasse `Array` befindet sich im Paket `java.lang.reflect`.

Die Java-Standard-Klasse `Arrays` ist im Paket `java.util` enthalten und definiert nützliche Funktionen zum Vergleichen, Sortieren und Füllen von Arrays. Mit der Version Java 5.0 wird mit der Methode `toString()` dieser Klasse eine `String`-Repräsentation für eindimensionale Arrays geliefert.

Aufgabe 1.12



Der Umgang mit Array-Objekten

Definieren Sie eine Klasse `ArrayTest1`, die ein- und zweidimensionale Arrays von primitiven Typen und Referenztypen deklariert und initialisiert. Wird ein Array-Objekt mit dem `new`-Operator erzeugt, muss dies über die Angabe einer festen Größe erfolgen. Denken Sie sich sowohl für Deklarationen als auch beim Initialisieren von Arrayelementen alternative Lösungen aus und vergleichen Sie Ihre Ergebnisse mit denen aus dem Lösungsvorschlag für diese Aufgabe. Benutzen Sie für die Ausgabe am Bildschirm von eindimensionalen Arrays die Methode `toString()` der Klasse `Arrays` und definieren Sie eine Klassenmethode `anzeige()`, die eine Referenz von einem zweidimensionalen Array übergeben bekommt und dessen Elemente am Bildschirm anzeigt.

Primitive Datentypen können als Objekte der Klasse `Class<T>` von Java (wurde mit Java 5 generifiziert) über Standard-Namen wie `int.class`, `long.class` etc. angesprochen werden. Benutzen Sie die Methode `newInstance()` der Klasse `Array`, um ein Array-Objekt dynamisch zu erzeugen, und deren Methoden `setInt()` und `getInt()`, um die Elemente des so erzeugten Arrays zu manipulieren. Zeigen Sie am Beispiel der Klasse `Vektor` aus der Aufgabe 1.5, dass Arrayelemente von Referenztypen immer einzeln instanziiert werden müssen, und rufen Sie für deren Anzeige am Bildschirm die Methode `anzeige()` der Klasse `Vektor` auf. Achten Sie auf die Java-Klassen bzw. Pakete, die von der Klasse `ArrayTest1` importiert werden müssen.

Java-Dateien: `ArrayTest1.java`

Programmaufruf: `java ArrayTest1`

1.13 Zeichenketten und die Klasse String

Objekte, die von der Java-Standard-Klasse `String` instanziiert werden, repräsentieren eine Folge von `char`-Werten. Um das Arbeiten mit der Klasse zu vereinfachen, wurde in Java eine abgekürzte Form definiert, mit der Objekte der Klasse ohne einen `new`-Operator gebildet werden können, und zwar durch die einfache Zuweisung einer Zeichenkette (Folge von `char`-Werten zwischen Anführungszeichen, auch Literal genannt): `String s = "Java";`. Das zugewiesene Literal wird vom Compiler in einen sogenannten Konstantenpool eingetragen, eine Liste, die alle anderen Literale und Konstanten aus der Klassendefinition enthält. Der Stringvariablen wird eine Referenz auf dieses Literal zugewiesen. Dies hat als Konsequenz, dass bei einer Objektinstanzierung in der Form: `String s = new String("Java ");`

zwei `String`-Objekte entstehen, das Literal und eine Kopie davon, und sollte aus diesem Grund vermieden werden.

Compact Strings

Die Weiterentwicklung von Java wird durch JDK Enhancements Proposals (JEPs), die die Features für neue Versionen und deren Implementierung im JDK im Detail beschreiben, im Voraus festgelegt. Diese werden auch als »Arbeitspakete für die Erweiterung von Java« in der Literatur bezeichnet.

Ein wichtiger Beitrag zur Einsparung von Hauptspeicher gibt der JEP 254 (Compact Strings) zu Java 9 vor. Obwohl die meisten Strings kein UTF-16-Format benötigen und nur aus Zeichen aus den Latin1- (8 Bit) oder ASCII- (7 Bit) Zeichentabellen bestehen, wurde bis einschließlich Java 8 jeder String als `char[]`-Array abgespeichert, sodass meistens die Hälfte des vorgesehenen Speicherplatzes ungenutzt blieb.

ISO 8859-1, auch bekannt als Latin-1, ist ein von der ISO zuletzt 1998 aktualisierter Standard für die Informationstechnik zur Zeichencodierung mit 8 Bit. Die davon mit 7 Bit codierbaren Zeichen entsprechen US-ASCII mit einem führenden Null-Bit. Zusätzlich zu den 95 darstellbaren ASCII-Zeichen ($20_{16}-7E_{16}$) codiert ISO 8859-1 96 weitere ($A0_{16}-FF_{16}$), also insgesamt 191 von theoretisch möglichen $256(=2^8)$ Zeichen.

Der JEP 254 ändert die interne Repräsentation von Strings, indem ein Flag für das Encoding (ISO-Latin1 oder UTF-16) und ein `byte`-Array, in dem der String dann entweder mit einem oder zwei Byte pro Zeichen gespeichert wird, vorgesehen wurden (siehe dazu die Aufgabe 1.13).

HTML-Dateien und JavaScripts

Mit der HyperText Markup Language (HTML) wird einem Webbrowser mitgeteilt, wie er Inhalte in Form von Webseiten für den Benutzer anzuzeigen hat. Die HTML-Sprache besteht aus einer Vielzahl von Tags, über die durch Auszeichnungen von Textteilen einem Dokument eine Struktur verliehen wird:

- `<html> </html>` definiert den Rahmen eines HTML-Dokuments.
- `<head> </head>` definiert den Rahmen für den Header eines HTML-Dokuments.
- `<body> </body>` definiert den Rahmen für den Inhalt eines HTML-Dokuments.

Ein HTML-Dokument besteht aus drei Bereichen:

- der Dokumententypdeklaration (DOCTYPE) ganz am Anfang der Datei, die die verwendete Dokumententypdefinition angibt,
- dem HTML-Kopf (Head), der hauptsächlich technische oder dokumentarische Informationen enthält, die nicht unbedingt direkt im Browser sichtbar sind, und
- dem HTML-Body, der die anzuzeigenden Informationen enthält.

JavaScript (kurz JS) ist eine Skriptsprache, die ursprünglich in Webbrowsern entwickelt wurde, um die Möglichkeiten von HTML zu erweitern. Damit entwickelte Scripts können in Java mittels Instanzen vom Typ des Interface `javax.script.ScriptEngine` interpretiert und ausgeführt werden. Dazu kann die `eval()`-Methode aufgerufen werden.

Über URI (»uniform resource identifier«)-Referenzen können Ressourcen im Netzwerk eindeutig identifiziert werden. Der einfachste Weg, eine korrekte URI-Instanz für eine `File`-Instanz zu erzeugen, ist der Methodenaufruf `toURI()` an einem `File`-Objekt. Eine so spezifizierte URI kann mithilfe der Methode `browse()` der `java.awt.Desktop`-Klasse im Browser angezeigt werden.

Textblöcke

Wie der JEP 355 (Text Blocks Preview) zum Ausdruck bringt, definiert ein Textblock »eine neue Art von multi-line String-Literalen in Java, die den Gebrauch der meisten Escape-Sequenzen unnötig machen und den String automatisch formatieren«.

Textblöcke sind als alternative String-Repräsentation in Java zu sehen und sollten benutzt werden, um Sourcecode und Textsequenzen einzuschließen, die über mehrere Zeilen eine Vielzahl von Zeilenumbrüchen, String-Konkationen und Trennzeichen erwarten.

Ein Textblock besteht aus 0 oder mehreren Characters, die zwischen einem öffnenden und einem abschließenden Begrenzungszeichen eingeschlossen sind, die jeweils aus drei Anführungszeichen (""") bestehen (»opening« bzw. »closing delimiter«). Alternativ zu:

```
final static String myString =
    "Da steh' ich nun, ich armer Tor!\n" +
    "    Und bin so klug als wie zuvor;\n" +
    "    Heiße Magister, heiße Doctor gar,\n" +
    "Und ziehe schon an die zehen Jahr',\n" +
    "    Herauf, herab und quer und krumm,\n" +
    "    Meine Schüler an der Nase herum -\n" +
    "Und sehe, daß wir nichts wissen können!\n";
```

kann der String als Textblock um einiges übersichtlicher und einfacher lesbar dargestellt werden:

```
final static String myString = """
    Da steh' ich nun, ich armer Tor!
        Und bin so klug als wie zuvor;
        Heiße Magister, heiße Doctor gar,
    Und ziehe schon an die zehen Jahr',
        Herauf, herab und quer und krumm,
```

```
Meine Schüler an der Nase herum -  
Und sehe, daß wir nichts wissen können!  
""";
```

Der Textblock definiert keinen neuen Java-Typ, ist wie alle Literale in Java vom Typ `String` und im Bytecode ist nach der Übersetzung kein Unterschied zu `String`-Literalen zu sehen. `String`-Literalen und Textblöcke können gleich, mehr noch, identisch sein, wie z.B. in den Definitionen mit:

```
String textBlock = ""  
Hallo Java 13!"";
```

und

```
String literal = "Hallo Java 13!";
```

Genauer formuliert, ist das öffnende Begrenzungszeichen in einem Textblock eine Sequenz von drei Anführungszeichen (""") gefolgt von 0 oder mehreren Whitespaces und einem zwingend notwendigen Zeilenumbruch (»line terminator« – eine aus 1 bis 2 Characters bestehende Character-Sequenz, die das Ende einer Zeile markiert). Mit dem ersten Character, das dem Zeilenumbruch folgt, beginnt der Inhalt des Textblocks und dieser endet vor dem ersten Anführungszeichen des abschließenden Begrenzungszeichens (""").

Der Verständlichkeit halber möchte ich erwähnen, dass je nach Kontext »verschiedene Zeichen als Whitespaces (Leerraum) angesehen werden, fast immer zumindest Leerzeichen und Tabulatoren, meist auch Zeilenumbrüche«. Leerzeichen (spaces) und Tabulatoren (Tabs) haben verschiedene Repräsentationen in ASCII: 0x20 bzw. 0x09. Der in Unix benutzte `newline`-Character (»line terminator«) ist `\n` (LF »Line Feed«) und in Windows `\r\n` (CRLF »Carriage return« und »Line Feed«).

Der Inhalt eines Textblocks kann anders als `String`-Literalen das Anführungszeichen (") direkt beinhalten. Die Benutzung von (\") ist ebenfalls gestattet, wird aber nicht unbedingt empfohlen. Wie in der Literatur vermerkt, wurde das »fette« Begrenzungszeichen (""") so gewählt, dass der Character (") ohne Entwertung (\) benutzt werden kann und so den Textblock von einem gewöhnlichen `String`-Literal syntaktisch unterscheidet. Enthält der Textblock selbst drei Anführungszeichen, die nacheinander folgen, muss eines davon entwertet (»escaped«) werden und anstelle von \\"\"\" wird die Schreibweise \""" empfohlen.

Darüber hinaus darf der Textblock anders als ein `String`-Literal Zeilenumbrüche direkt beinhalten. Die Benutzung von `\b`, `\r`, `\n`, `\t`, `\'`, `\"` und `//`, sowie von oktaalen Escape-Sequenzen wird nicht unterbunden, ist aber weder notwendig noch zu empfehlen.

Wie im Text Blocks Preview gelesen werden kann, ist der Textblock:

```
""  
line 1  
line 2  
line 3  
""
```

equivalent mit dem String-Literal:

```
"line 1\nline 2\nline 3\n"
```

bzw. der Zusammensetzung (»concatenation«) von Strings:

```
"line 1\n" +  
"line 2\n" +  
"line 3\n"
```

Wird das abschließende Begrenzungszeichen direkt der letzten Zeile des Inhalts hinzugefügt:

```
""  
line 1  
line 2  
line 3""
```

so erscheint das String-Literal ohne Zeilenumbruch:

```
"line 1\nline 2\nline 3"
```

Die automatische Formatierung eines Textblocks bezieht sich auf das Einrücken (»indentation«) von Zeilen, das Hinzufügen von relevanten (»essential«) und das Löschen von nicht-relevanten (»incidental«) Whitespaces sowie das Interpretieren von Zeilenumbrüchen und anderen Escape-Sequenzen. Diese Aufgaben werden beim Kompilieren durchgeführt.

Das Behandeln von Textblöcken beim Kompilieren erfolgt immer in der gleichen Reihenfolge:

- Weil die Zeilenumbrüche in Sourcecode-Dateien von Plattform zu Plattform verschieden sind, werden in einem ersten Schritt alle Zeilenumbrüche betriebssystemunabhängig von CR (`\u000D`) und CRLF (`\u000D\u000A`) in LF (`\u000A`) umgesetzt. Die Escape-Sequenzen `\n` (LF), `\f` (FF) und `\r` (CR) werden während dieser Umsetzung nicht interpretiert.

- Danach wird der Whitespace, der durch die Code-Formatierung entstanden ist, entfernt.
- Als Letztes werden alle Escape-Sequenzen interpretiert und aufgelöst.

Wie in der weiterführenden Literatur vermerkt, benutzt der Compiler für das Löschen von »nicht relevanten« (»incidental«) Whitespaces einen nicht gerade einfachen Algorithmus, der im Groben für Leerzeichen wie folgt beschrieben werden kann:

- Alle Leerzeichen, die sich am Ende von Zeilen befinden, werden gelöscht.
- Für führende Leerzeichen werden alle Zeilen, die nicht nur Leerzeichen enthalten, geprüft:
 - Die Anzahl der führenden Leerzeichen-Characters wird in allen Zeilen gezählt.
 - Die kleinste Anzahl dieser Characters wird ermittelt und genau diese Anzahl Characters wird in jeder Zeile gelöscht.
 - Als Ergebnis wird zumindest eine der Zeilen keine führenden Leerzeichen mehr beinhalten.
- Wichtig ist auch zu wissen, dass die Zeile, die das abschließende Begrenzungszeichen (""") beinhaltet (diese wird auch als »significant trailing line policy« bezeichnet), in die Prüfung miteinbezogen wird (auch wenn diese leer ist, das heißt, dass (""") das einzige Zeichen ist, das sie beinhaltet).

Interpretieren wir die nachfolgenden Beispiele nach diesen Regeln:

```
String hallo = ""
    Hallo Java 13!
    ""

// >Hallo Java 13!
String hallo = ""
    Hallo Java 13!
    ""

// >   Hallo Java 13!
String hallo = ""
    Hallo Java 13!
    ""

// >   Hallo Java 13!
```

stellen wir fest, dass bereits durch die Positionierung des Inhalts relativ zum abschließenden Begrenzungszeichen führende Leerzeichen den Zeilen hinzugefügt (bzw. darin gelöscht) werden können.

1. Beispiel:

```
String hallo = ""
    Hallo Java 13!
    "";
```

Weil der Inhalt des Textblocks eine einzige Zeile beinhaltet, die dieselbe Einrückung (denselben Einzug) hat wie die Zeile mit dem abschließenden Begrenzungszeichen (vier Leerzeichen), wird diese gänzlich gelöscht und das Ergebnis ist:

```
"Hallo Java 13!\n"
```

2. Beispiel:

Verschieben wir stattdessen das abschließende Begrenzungszeichen nach links, bringt uns ein anderer Weg zum gleichen Ergebnis (weil keine gemeinsamen Leerzeichen vorhanden sind):

```
String hallo = ""  
    Hallo Java 13!!  
"";
```

3. Beispiel:

Wird der Inhalt um vier Leerzeichen nach rechts geschoben, bleibt die gemeinsame Einrückung weiter bei 4 Leerzeichen, die gelöscht werden. Die andere Hälfte der Einrückung wird als »relevant« interpretiert und das Ergebnis ist:

```
"    Hallo Java 13!\n"
```

Wird in einem ergänzenden 4. Beispiel das abschließende Begrenzungszeichen der letzten Zeile aus dem Inhalt hinzugefügt, gibt es keine Möglichkeit, eine der Einrückungen als »relevant« zu markieren, der Compiler wird immer alle Leerzeichen aus allen Zeilen löschen:

```
String hallo = ""  
    Hallo Java 13!"";
```

und das Ergebnis bleibt wie im 1. Beispiel. Dies bedeutet auch, dass ohne ein abschließendes Begrenzungszeichen in einer eigenen Zeile keine Einrückung erreicht werden kann.

Als Wiederholung möchte ich nochmals festhalten, dass generell das Löschen von führenden Leerzeichen durch die Positionierung des abschließenden Begrenzungszeichens bestimmt werden kann. Zusammengefasst können dafür folgende grundlegende Regeln formuliert werden:

- Der Inhalt eines Textblocks wird so weit nach links geschoben, bis die Zeile mit den wenigsten führenden Leerzeichen keine Leerzeichen mehr besitzt.
- Um das Löschen von Leerzeichen zu vermeiden, damit relevante Leerzeichen nicht als nicht relevant interpretiert werden, kann der Inhalt eines Textblocks so weit nach rechts verschoben werden wie die Einrückung in der Zeile, die das abschließende Begrenzungszeichen enthält.

- Durch die Positionierung des abschließenden Begrenzungszeichens auf die Position des ersten Characters einer Zeile aus dem Sourcecode (am Anfang der letzten Zeile des Inhalts) kann das Entfernen von jeglichen sogenannten nicht-relevanten Leerzeichen unterbunden werden.

Es ist aber nicht möglich, das Löschen von Leerzeichen zu unterbinden, die am Ende von Zeilen stehen. In Situationen, in denen die Leerzeichen am Ende wichtig sind, muss manchmal nachgeholfen werden, wie z. B. durch die Aufteilung des Textblocks oder die Benutzung von Substitutionen und oktalen Escape-Sequenzen wie in den folgenden Beispielen:

```
String textBlock1 =
    ""
    In Java können Textblöcke und \040\040
    String-Literale identisch sein.
    """;
String textBlock2 =
    ""
    In Java können Textblöcke und"" + " \n" + ""
    String-Literale identisch sein.
    """;
String textBlock3 =
    ""
    In Java können Textblöcke und $$
    String-Literale identisch sein.
    ""$.replace("$", "");
```

Weil der Java-Compiler nicht die Fähigkeit besitzt, alle Tabulatoren aus den unterschiedlichen Textsystemen zu erkennen, wurde für die Benutzung von Tabulatoren im Text (TAB \t -Characters) die Regel aufgestellt, diese wie einzelne Leerzeichen zu behandeln: Ein Tabulator entspricht in Textblöcken einfach einem Leerzeichen. So wird aus:

```
String tabString = ""
    Hallo
    Java
    13! """;
```

nach dem Kompilieren der String:

```
    Hallo
Java
    13!
```

ermittelt, weil die zwei Tabs aus der zweiten Zeile mit zwei Leerzeichen gleichgestellt werden und jede Zeile zwei Zeichen nach links verschoben wird.

Nachdem die Zeilen des Textblockinhalts entsprechend eingerückt wurden, werden die Escape-Sequenzen aus dem Text interpretiert. Wie bereits erwähnt, werden die gleichen Escape-Sequenzen wie in einem String-Literal unterstützt: `\n`, `\t`, `\'`, `\"`, `\\` und als Programmierer können Sie auf diesen Prozess mittels der neuen Instanzmethode von `String` `String::translateEscapes()` Einfluss nehmen. Letztendlich dient diese Methode der Ausführung von Textblöcken und allen anderen String-Literalen.

Weitere Methoden, die zur Unterstützung von Textblöcken der `String`-Klasse hinzugefügt wurden, sind:

- `String::stripIndent()` – wird zum Eliminieren von nicht-relevanten Whitespaces benutzt, indem sie den gesamten Text nach links rückt, ohne die Formatierung zu verändern. Sie sollte eingesetzt werden, wenn ein Einrücken von Textzeilen in Eingabedaten in gleicher Art und Weise wie in Textblöcken gewünscht ist.
- `String::indent()` – wird zum Hinzufügen von Whitespaces benutzt.
- `String::formatted(Object... args)` – vereinfacht das Ersetzen von Werten in Textblöcken, die durch Platzhalter (Templates) angegeben wurden. Dafür gibt es bereits in der `String`-Klasse die statische Methode `format()`. Der Vorteil der `formatted()`-Methode liegt darin, dass diese als Instanzmethode direkt an einem Textblock, durch Punkt getrennt, aufgerufen werden kann.

Alle neuen Methoden wurden als deprecated (zum Löschen gedacht) gekennzeichnet, um zu unterstreichen, dass diese Teil eines Preview Features sind.

Weil die meisten höheren Programmiersprachen bereits String-Literale enthalten, die sich über mehrere Textzeilen erstrecken und einfach mit Whitespace zu formatieren sind, ist die Einführung der Textblöcke in Java eine bedeutende Neuerung. Darüber hinaus vereinfachen Textblöcke die Nutzung von Code aus anderen Programmiersprachen in String-Definitionen, wie z.B. Java-Script-Code, und ermöglichen, HTML- oder auch SQL-Statements (dienen Datenbankzugriffen) übersichtlicher darzustellen.

Im Umgang mit Textblöcken können dieselben Eigenschaften wie im Falle von String-Literalen festgestellt werden. Wie mit den Beispielen aus der Aufgabe 1.14 gezeigt wird, können beide String-Arten in der Zusammenführung von Strings alternativ eingesetzt werden.

Parallel zum Text Blocks Preview empfehlen wir zur Verdeutlichung von Syntax und Einsatz von Textblöcken einen Blick in den »Programmer's Guide to Text Bocks« auf der Webseite http://cr.openjdk.java.net/~jlaskey/Strings/TextBlocksGuide_v9.html.

Stichwortverzeichnis

...-Operator 48
@FunctionalInterface 174
@Override 104, 176
<>-Operator siehe Diamond-Operator
==-Operator 34, 122

A

AbstractCollection 210
AbstractList 210
AbstractMap 212
AbstractSet 210
accept() 315, 317
Accumulator 321, 323
add() 210, 219, 231, 316
andThen() 316
Annotation 173
annotationType() 173
apply() 306, 315
Argumentenliste 49
Array 36, 37, 208, 209, 304, 324
Array (Klasse) 36
ArrayIndexOutOfBoundsException 280
ArrayList 210, 213, 322, 326
Array-Objekt 214
asList() 319, 322
AtomicInteger 159
AtomicLong 159
Aufzählung siehe Enumeration
Ausnahme siehe Exception
Autoboxing 33
Auto-unboxing 33
average() 319

B

BaseStream 313
Begrenzungszeichen 39
BiConsumer 315
BiFunction 315
BigDecimal 159, 160

BigInteger 159
BinaryOperator 316
BiPredicate 315
Boolean 33
break 338
bridge methods siehe Brückenmethode
Brückenmethode 107, 206, 207
builder() 318, 320
Bulk-Operation 310, 323
Byte 33, 159
Bytecode 21, 123

C

Calendar 51
call by reference 26
call by value 26
catch-Block 32, 275
Character 33
Class 30, 35, 281
ClassCastException 219, 280
Classpath 422
Cleaner 124
Clock 56
clone() 115, 122
Cloneable 115, 122
CloneNotSupportedException 122
Code as data 298
collect() 318, 319, 323, 326
Collection 210, 211, 213, 311, 314
Collector 318, 322, 326
Combiner 322, 323
Comparable 210, 218, 221
Comparator 210, 218, 224, 304
compare() 218, 223, 304
compareTo() 218, 221
concat() 46, 276
ConcurrentMap 326
Concurrent-Reduction 326
Constructor 281

Consumer 312, 317
contains() 210, 219
Copy-Konstruktor 25, 123
count() 319

D

Date 51
Date&Time-API 51
Datenkapselung 24
DateTimeFormatter 54
Default-Methode 309, 314
Diamond-Operator 228, 229, 231, 308
distinct() 324
Double 33, 159, 204
DoubleStream 313
doubleValue() 160
Down-casting
 verkleinernde Konvertierung 108
Duration 52

E

effectively final 166
elements() 216
empty() 327
--enable-preview 339
entrySet() 323
Enum 217
enum siehe Enumeration
Enumeration 212, 216, 313
EnumMap 212
equals() 111, 116, 117, 210, 217, 218, 222
Erasure 205
Error 32, 275
Erweitern siehe Klasse, abgeleitet
eval() 39
Exception 32, 275, 278, 317
exit() 32
exports 420
extends-Klausel 99

F

Field 35
 File 31, 278
 Files.lines() 330
 FileWriter 31
 filter() 312, 328
 final 166
 finalize() 124
 finally-Block 275, 277
 findAny() 312
 findFirst() 312
 flatMap() 327, 328, 329
 Float 33, 159, 204
 floatValue() 33, 160
 forEach() 312, 314, 317, 325
 forEachRemaining() 314
 for-each-Schleife 48, 212
 format() 45, 54
 formatted() 45
 forName() 280
 for-Schleife 48
 Function 305, 307, 316

G

Garbage Collector 123
 generate() 317, 320, 326
 Generizität siehe Generics
 get() 214, 306, 312, 318, 328
 getAsDouble() 319
 getAvailableZoneIds() 53
 getBytes() 47
 getClass() 111, 209
 getConstructor() 283
 getenv() 31
 getInt() 37
 getMessage() 275
 getProperties() 31
 getRuntime() 32
 getUnits() 54
 getValue() 205
 GregorianCalendar 51, 56
 groupingBy() 323, 326
 groupingByConcurrent() 326

H

hash() 116
 hashCode() 112, 116, 117, 210, 218, 222
 HashMap 212
 hashMoreElements() 217
 HashSet 210
 Hashtable 212, 216

hasNext() 217, 313
 Heap 123
 HTML 38
 Hüllenklasse siehe Wrapper-Klasse

I

identity() 316
 if-Anweisung 34
 ifPresent() 313, 328, 331
 IllegalAccessException 280
 IllegalArgumentException 215
 import-Anweisung 28
 indent() 45
 InputStream 31
 instanceof 111
 Instant 51
 InstantiationException 280
 Instanz siehe Objekt
 Instanzfeld 20
 Instanzmethode 20, 105
 dynamisches Binden 105
 Integer 33, 159, 204
 Interface 161
 Static-Member 166
 IntStream 313
 intValue() 33, 160
 invoke() 317
 invokedynamic 298
 isDirectory() 279
 isFile() 279
 isInstance() 111
 isPresent() 312
 Iterable 210, 212, 216, 314
 iterate() 318, 320
 Iterator 210, 216, 313
 iterator() 216, 314

J

Java
 -API 30
 -Applet 22
 -Applikation 22
 -Servlet 22
 Java Collection Framework 209
 java.io 31
 java.lang 31
 java.lang.annotation 173
 java.lang.reflect 34, 36, 281
 java.math 160
 java.time 51

java.util 37, 51, 209
 java.util.stream 310
 javadoc 177
 javax.script.ScriptEngine 39
 joining() 322
 JVM
 Virtuelle Java Maschine 21, 123

K

keys() 216
 Klasse 19
 abgeleitete 99
 abstrakte 159
 anonyme 166, 297, 301
 immutable 49, 303
 lokale 166, 297
 Member 165
 mutable 51, 303
 Static-Member 165
 Klassendatei, Objektdatei 28
 Klassenfeld 19
 Klassenmethode 19
 statisches Binden 103
 Komposition 105
 Konsole 31
 Konstruktor 19, 20
 privater 56
 Konstruktor-Referenz 304
 Kovarianter Rückgabotyp 207

L

Lambda-Ausdruck 297
 Lambda-Expression siehe Lambda-Ausdruck
 Legacy Code 205
 limit() 318, 319, 320, 324
 List 210, 213, 215, 316
 ListIterator 211
 listIterator() 211
 LocalDate 52
 LocalDateTime 52
 LocalTime 52
 Long 33, 159
 LongStream 313
 lower bound wildcard
 untere Schranke 204

M

main() 22
 Map 211, 214, 215
 Map.Entry 212, 215

- map() 311, 312, 318, 322, 327, 328
 - Math 30
 - max() 319
 - Method 35
 - Methoden
 - überladen 23
 - Methoden-Referenz 304, 306
 - Migration 426
 - min() 319
 - Modul 420
 - Modularisierung 31, 419
 - Moduldeskriptordatei 420
 - Modul-Path 422
 - Modul-Source-Path 422
 - Mutable reduction 318
- N**
- newInstance() 35, 36, 37, 208, 209, 280, 283
 - new-Operator 19, 123, 162, 167, 201
 - next() 217, 313
 - nextElement() 217
 - now() 53
 - Number 33, 159, 160, 204
 - NumberFormatException 279
- O**
- Oberklasse, Superklasse 99
 - Object 30, 105, 106, 110, 113, 161, 202
 - Objekt 19
 - of() 53, 211, 215, 216, 313, 318, 320, 328
 - ofNullable() 328
 - opens 421
 - Optional 312, 318, 319, 327
 - ordinal() 217
 - orElse() 313
 - orElseGet() 331
 - orElseThrow() 331
- P**
- package-Anweisung 28, 102
 - Paket 27, 29, 420
 - parallel() 313, 323
 - parallelPrefix() 324, 327
 - parallelSetAll() 324, 327
 - parallelSort() 324, 327
 - parallelStream() 323
 - parseDouble() 46
 - parseInt() 46, 276, 277
 - parseShort() 46
 - peek() 312, 325
 - Period 52
 - Polymorphismus
 - impliziter 109
 - parametrisierter 109
 - überladener 109
 - Predicate 312
 - printStackTrace() 275
 - PrintStream 22, 31
 - Process 30
 - ProcessBuilder 30, 32
 - provides 421
 - put() 214
 - sorted() 324
 - split() 330
 - Stack 210
 - Standardausgabe 31
 - Standardeingabe 31
 - Stream 310, 313, 316, 317, 322
 - parallel 313
 - sequenziell 313
 - stream() 311, 313, 319
 - String 37, 40, 46
 - StringIndexOutOfBoundsException 284
 - stripIndent() 45
 - substring() 46, 276, 277
 - super() 99, 278
 - Supplier 305, 307
 - Switch
 - Expression 335, 337
 - Statements 335, 337
 - switch-Anweisung 48
 - System 23, 30, 275
 - System.exit() 32
 - System.gc() 124
 - System.out.print() 22
 - System.out.println() 22
 - Systemeigenschaften 31
- T**
- Target-Typ 299
 - TemporalAccessor 54
 - TemporalUnit 54
 - TemporalAdjuster 54
 - test() 315
 - Textblock 39
 - this-Referenz 25, 302
 - throw 32, 276, 279
 - Throwable 275
 - toArray() 214, 312
 - toBinaryString() 33
 - toConcurrentMap() 326
 - toHexString() 33
 - toList() 322
 - toString() 37, 106, 161, 217
 - toURI() 39
 - translateEscapes() 45
 - TreeMap 214
 - TreeSet 210, 224
 - try-Block 32, 275
 - Typ
 - generischer 201, 207
 - inferred 299
 - parametrisierter 201
 - reifbar 206
- Q**
- Queue 210
- R**
- Raw-Typ 205
 - reduce() 318, 319, 320, 323, 326
 - reducing() 323, 326
 - Reduction 326
 - Referenz 19
 - globale 27
 - lokale 27
 - Reflection-API 34
 - remove() 210
 - requires 420
 - Rückgabotyp
 - kovarianter 207
 - Runtime 30, 32
 - RuntimeException 276, 278, 317
- S**
- Schnittstelle siehe Interface
 - Scoping 302
 - sequential() 313
 - Service 424
 - Service-Interface 425, 432
 - ServiceLoader 425
 - Set 210, 211, 215
 - setInt() 37
 - Short 33, 159
 - Sichtbarkeits Ebenen 100
 - size() 210
 - skip() 319, 324
 - sort() 220, 304

Typargument 201
Typinferenz 227, 231, 298,
331
Typlöschung siehe Erasure
Typparameter 201
Typsicherheit 33, 201
Typvariable 201

U

Umgebungsvariable 28
UnaryOperator 316
unmodifiableList() 216
unmodifiableMap() 216
unmodifiableSet() 216
UnsupportedOperation-
Exception 216
Unterklasse, Subklasse 99,
104

Up-casting
vergrößernde Konvertie-
rung 108
Upper bound wildcard
obere Schranke 204
URI 39
uses 421

V

valueOf() 277
values() 217
var 331
varargs-Methoden 48
Variable Capture 302
Variablen-Typinferenz 331
Vector 213
Vererbung 99, 105

W

Webbrowser 22
Wildcardtyp 204, 208
Wrapper-Klasse 32

Y

yield 338

Z

Zieltyp siehe Target-Typ
ZonedDateTime 53
ZoneId 53
ZoneOffset 53
ZoneRules 53
Zugriffsmethode 24, 25