

Objektorientiertes Programmieren

Kurseinheit 3: Klassen, Objekte und Felder

Autoren: Bernd J. Krämer, Silvia Schreier, Helga Huppertz

Abbildungen und Animationen: Volker Winkler, Silvia Schreier

Inhaltsverzeichnis

Kurseinheit 1

1	Von der Aufgabenstellung zum Programm	1
1.1	Motivation	1
1.2	Softwareentwicklung	2
1.3	EXKURS: Unified Modeling Language (UML)	4
2	Anforderungsanalyse	9
2.1	Fallstudie	9
2.2	Analyse der Anwendungswelt.....	10
2.3	Anwendungsfälle und Akteure	10
2.4	Beziehung zwischen Akteuren	12
2.5	Beziehung zwischen Anwendungsfällen.....	13
2.6	Anwendungsfallbeschreibungen	16
2.7	Datenlexikon	19
2.8	Pflichtenheft	20
3	Einführung in die Objektorientierung	22
3.1	Objekte und Klassen	22
3.2	Beziehungen.....	24
3.3	Kommunikation	27
3.4	Objektorientierte Analyse und Entwurf.....	28
3.5	UML-Klassendiagramm	29
3.6	UML-Objektdiagramm	32
4	Einführung in die Algorithmik	33
4.1	Algorithmen	33
4.2	Verhaltensbeschreibung und Kontrollstrukturen.....	38
5	Programmiersprachen	42
5.1	EXKURS: Entwicklungsgeschichte von Programmiersprachen.....	42
5.2	Eingabe, Verarbeitung, Ausgabe	44
5.3	Interaktive Programme	45
6	Einführung in die Java-Programmierung	47
6.1	Entwicklung und Eigenschaften der Programmiersprache Java	47
6.2	Erstellen, Übersetzen und Ausführen von Java-Programmen.....	48
7	Zusammenfassung	52
	Lösungshinweise	55
	Index	65

Kurseinheit 2

8 Praktischer Einstieg in die Java-Programmierung	67
8.1 Ein erstes Programm	67
8.2 Klassen für die Praxis	69
8.3 Programmier- und Formatierhinweise	70
9 Primitive Datentypen und Ausdrücke	72
9.1 Ganze Zahlen	72
9.2 Gleitkommazahlen	75
9.3 Operatoren und Ausdrücke	77
9.4 Auswertung von Ausdrücken	79
9.5 Datentyp <code>boolean</code>	82
9.6 Datentyp <code>char</code>	83
10 Variablen und Zuweisungen	85
10.1 Variablen	85
10.2 Zuweisung	88
11 Typanpassung	93
11.1 Implizite Typanpassung	93
11.2 Explizite Typanpassung	98
12 Anweisungen	102
12.1 Blöcke	102
12.2 Kontrollstrukturen	104
13 Bedingungsanweisungen	105
13.1 Die einfache Fallunterscheidung	105
13.2 Die Mehrfach-Fallunterscheidung	109
14 Wiederholungs- und Sprunganweisungen	113
14.1 Bedingte Schleifen: <code>while</code> und <code>do ... while</code>	113
14.2 Die Zähl- oder <code>for</code> -Schleife	116
14.3 Strukturierte Sprunganweisungen: <code>break</code> und <code>continue</code>	119
15 Gültigkeitsbereich und Sichtbarkeit von lokalen Variablen	124
16 Zusammenfassung	127
Lösungshinweise	129
Index	145

Kurseinheit 3

17 Objekttypen	149
17.1 Ein Objekt verwenden	149
17.2 Erzeugung und Lebensdauer von Objekten.....	153
17.3 Wertvariablen und Verweisvariablen.....	154
17.4 Zusammenspiel von Objekten.....	159
18 Klassenelemente	163
18.1 Klassenvereinbarung.....	163
18.2 Attributdeklaration	164
18.3 Methodendeklaration	165
18.4 Konstruktordeklaration	176
19 Klassenvariablen und -methoden	180
19.1 Klassenvariablen	180
19.2 Klassenmethoden	182
20 Felder	185
20.1 Felder einführen und belegen	185
20.2 Mehrdimensionale Felder	189
20.3 Erweiterte for-Schleife.....	193
21 Zusammenfassung	195
Lösungshinweise	197
Index	221

Abbildungsverzeichnis

Abb. 17.1-1	Das Rechnung-Objekt <code>rechnung1</code>	149
Abb. 17.1-2	Die Klasse <code>Rechnung</code> mit zwei Attributen	150
Abb. 17.1-3	Das Objekt <code>rechnung1</code> mit zwei möglichen Attributwerten ..	151
Abb. 17.1-4	Die Klasse <code>Rechnung</code> , erweitert um einige Methoden	151
Abb. 17.2-1	Die Klasse <code>Rechnung</code> , erweitert um einen Konstruktor	153
Abb. 17.3-1	Eine <code>int</code> -Variable mit einem Wert	154
Abb. 17.3-2	Eine <code>Rechnung</code> -Variable mit einer Referenz	155
Abb. 17.3-3	Kopiervorgang zwischen Wertvariablen	155
Abb. 17.3-4	Kopiervorgang zwischen Verweisvariablen	156
Abb. 17.3-5	Zwei unabhängige Verweisvariablen	157
Abb. 17.3-6	Eine <code>null</code> -Referenz	158
Abb. 17.4-1	Die Klassen <code>Rechnung</code> und <code>Kunde</code>	159
Abb. 18.2-1	Die Assoziation zwischen <code>Rechnung</code> und <code>Kunde</code>	165
Abb. 18.3-1	UML-Diagramm mit <code>Rechnungsposten</code> und <code>Artikel</code>	176
Abb. ML 13	Drei Variablen und drei Objekte	198
Abb. ML 14	Drei Variablen und zwei Objekte	198
Abb. ML 15	UML-Diagramm mit <code>Angestellter</code> und <code>Abteilung</code>	208
Abb. ML 16	UML-Diagramm mit <code>Auto</code>	209

Einführung

In der letzten Kurseinheit haben wir Grundlagen kennen gelernt, die von nun an immer wieder zum Einsatz kommen werden, wenn wir Java-Programme schreiben. Wir kennen einfache Anweisungen, Anweisungssequenzen und komplexere Konstrukte wie Fallunterscheidungen und Schleifenanweisungen.

In dieser Kurseinheit werden wir uns **Objekttypen**, den zentralen Elementen der objektorientierten Programmierung, zuwenden. Objekttypen werden durch **Klassenvereinbarungen** eingeführt. So ist es möglich Konzepte und Gegenstände des Anwendungsfalls angemessen in Java umsetzen zu können. **Klassen** bilden die elementaren Bausteine objektorientierter Programmiersprachen; sie sind die Baupläne für **Objekte**. Objekte sind **Exemplare** von Klassen. Wir lernen **Attribute**, **Methoden** und **Konstruktoren** zu deklarieren und zu verwenden. Diese Kurseinheit zielt darauf ab, ein detailliertes Verständnis der Konstruktion von Objekten, ihrer internen Arbeitsweise und der Kooperation zwischen Objekten zu entwickeln.

Ergänzend zu Klassen lernen wir eine weitere Art von Objekttypen kennen, denn in Anwendungsfällen kommt es häufig vor, dass wir mit Ansammlungen gleichartiger und systematisch geordneter Daten umgehen müssen. Typische Beispiele sind Telefonbücher, Adresskarteien, Tabellen und andere Verzeichnisse. Um solche Sammlungen auch in Java abbilden zu können, werden wir **Felder** kennen lernen.

Lernziele

- Wissen, wie man auf Attribute eines Objekts zugreift und
- wie man seine Methoden aufruft.
- Objekte beliebiger Klassen erzeugen können.
- Den Unterschied zwischen Wert- und Verweisvariablen und den Begriff der Objektidentität kennen.
- Wissen, was eine `null`-Referenz ist.
- Klassenvereinbarungen mit all ihren Bestandteilen, insbesondere (Klassen-) Attribute, (Klassen-)Methoden und Konstruktoren, in Java implementieren können.
- Eine Assoziation in Java implementieren können.
- Die Begriffe Signatur, Überladen und Verdecken kennen.
- Den Unterschied zwischen Attributen, lokalen Variablen und Parametern erläutern können.
- Wissen, wozu die Schlüsselwörter `this` und `static` verwendet werden können.
- Rückgabeeinweisungen in Java formulieren können.
- Wissen, was unter den Begriffen Getter- und Setter-Methoden verstanden wird und wozu diese Konzepte dienen.
- Den Unterschied zwischen Aufruf mit Wertübergabe und Aufruf mit Verweisübergabe erläutern können.
- Wissen, wie man Felder in Java nutzt, um einheitliche Sammlungen von Elementen abzulegen und zu verwalten.
- Feldvereinbarungen und Zugriffsoperationen auf Feldern formulieren können.
- Die Funktionsweise der erweiterten `for`-Schleife auf Feldern kennen.

17 Objekttypen

Bisher haben wir Anweisungen verwendet, um Variablen und Werte unterschiedlicher Typen zu manipulieren: ganze Zahlen, Gleitkommazahlen, Zeichen und Wahrheitswerte. Diese unterschiedlichen Typen haben gemeinsam, dass sie sogenannte primitive Datentypen sind.

Neben den primitiven Datentypen gibt es in Java die Objekttypen [JLS: § 4.3], die nicht von der Sprache vordefiniert sind, sondern durch Klassenvereinbarungen eingeführt werden. Dieses Kapitel stellt die Verwendung von Objekttypen vor. Objekttypen

17.1 Ein Objekt verwenden

Bei der objektorientierten Programmierung verkörpern Objekte reale oder abstrakte Gegenstände und Rollen der Anwendungswelt. Typischerweise existieren zur Laufzeit objektorientierter Programme mehrere Objekte, die durch den Austausch von Nachrichten miteinander interagieren. Wir beginnen unsere Betrachtung zunächst an einem einzelnen Objekt der Klasse `Rechnung`, das einer Variablen mit dem Namen `rechnung1` zugewiesen ist (Abb. 17.1-1).

rechnung1:Rechnung

Abb. 17.1-1: Das Rechnung-Objekt `rechnung1`

Der Typ der Variablen `rechnung1` ist `Rechnung`. `Rechnung` ist ein Objekttyp. Anders als primitive Datentypen sind Objekttypen nicht in Java vordefiniert. Ein Objekttyp `Rechnung` existiert genau dann, wenn im Kontext seiner Verwendung eine Klasse `Rechnung` definiert ist.

Bemerkung 17.1-1: Kontext

Damit eine Klasse verwendet werden kann, muss sie am Ort der Verwendung sichtbar sein. Die Sichtbarkeit von Klassen, Methoden und Attributen sowie Mechanismen zur Beeinflussung der Sichtbarkeit sind Gegenstand einer späteren Kurseinheit. Solange keine ausdrücklichen Vorkehrungen zur Sichtbarkeit getroffen werden, umfasst der Kontext gewisse Klassen der Java-Laufzeitbibliothek (API) sowie diejenigen eigenen Klassen, die sich in demselben Verzeichnis wie die aufrufende Klasse befinden.

┘

Rekapitulieren wir kurz, was wir aus der ersten Kurseinheit – zunächst noch unabhängig von Java-Sprachkonstrukten – bereits über Objekte und Klassen wissen:

- Objekte sind Exemplare¹⁵ einer Klasse.
- Objekte haben Attribute, auch Exemplarvariablen genannt.
- Attribute haben einen Typ und einen Attributwert.
- Der Zustand eines Objekts ist durch die Werte seiner Attribute festgelegt.
- Objekte haben Methoden, deren Ausführung durch das Senden einer Nachricht an das Objekt ausgelöst wird.
- Klassen beschreiben die Eigenschaften und das Verhalten aller ihrer Exemplare, denn durch die Klassenbeschreibung ist festgelegt, welche Attribute und Methoden Objekte dieser Klasse besitzen.

Um ein Objekt, beispielsweise das Objekt `rechnung1`, korrekt verwenden zu können, benötigen wir einige Informationen über den Objekttyp. Um Attribute des Objekts `rechnung1` ansprechen zu können, benötigen wir Kenntnis über die Namen und Typen der Attribute der Klasse `Rechnung`. Auch müssen wir die Methoden der Klasse `Rechnung` kennen, um korrekte Nachrichten an das Objekt `rechnung1` senden zu können. Solche Informationen können auf unterschiedliche Art dokumentiert sein. Im Rahmen dieses Kapitels entnehmen wir sie aus UML-Klassendiagrammen, wie wir sie in Kapitel 3 kennen gelernt haben.

Abb. 17.1-2 zeigt ein erstes Diagramm der Klasse `Rechnung` in UML-Notation. Unterhalb des Klassennamens sind die zwei Attribute `betrag` und `rabatt` dargestellt. Hinter jedem Attributnamen ist, durch einen Doppelpunkt getrennt, der Typ des Attributs angegeben. (Weitere Attribute und Methoden der Klasse `Rechnung` werden wir später aufführen, sie sind vorerst durch Punkte angedeutet.)

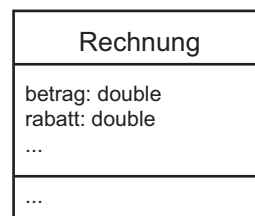


Abb. 17.1-2: Die Klasse `Rechnung` mit zwei Attributen

Wir können dieser UML-Klassenbeschreibung entnehmen, dass jedes Objekt der Klasse `Rechnung` über die `double`-Attribute `betrag` und `rabatt` verfügt. Beachten Sie, dass nur die Typen, nicht jedoch die Werte der Attribute durch die Klas-

15 Objekte werden bisweilen fälschlich auch als Instanzen (engl. *instance*) bezeichnet. Der Begriff „Instanz“ kommt aus dem Lateinischen und bedeutet „die für eine Entscheidung zuständige Stelle bei Behörden oder Gerichten“. Der Begriff „instance“ aus dem Englischen entspricht in unserem Kontext dem deutschen Begriff *Ausprägung* oder dem aus dem Lateinischen stammenden Wort *Exemplar* (für „Einzelstück aus einer Reihe gleichartiger Gegenstände“). Wir übersetzen den englischen Begriff „instance“ daher besser mit dem Wort „Exemplar“.

senbeschreibung festgelegt sind. Die Attributwerte können sich von Exemplar zu Exemplar unterscheiden und bilden den Zustand eines Objekts.

Objektzustand

Das Objekt `rechnung1` könnte beispielsweise zu einem bestimmten Zeitpunkt folgende Attributwerte aufweisen:

<u>rechnung1:Rechnung</u>
betrag = 248.20 rabatt = 0.05

Abb. 17.1-3: Das Objekt `rechnung1` mit zwei möglichen Attributwerten

In Java lassen sich die Attribute des Objekts `rechnung1` wie folgt ansprechen [JLS: § 6.2, § 15.11]:

Attributzugriff

```
rechnung1.betrag
rechnung1.rabatt
```

Attributwerte lassen sich als Ausdrücke in Anweisungen auswerten:

```
System.out.println(rechnung1.betrag);
double nettopreis = rechnung1.betrag * (1 - rechnung1.rabatt);
```

Weiterhin können Attributen neue Werte zugewiesen werden:

```
rechnung1.betrag = 78.70;
rechnung1.rabatt = 0.0;
```

Bei der Auswertung wie bei der Zuweisung von Attributwerten gelten die Regeln der Typverträglichkeit (vgl. Kapitel 11).

Das UML-Klassendiagramm in Abb. 17.1-4 ist erweitert um einige Methoden der Klasse `Rechnung`.

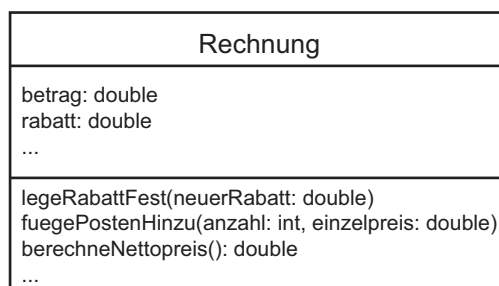


Abb. 17.1-4: Die Klasse `Rechnung`, erweitert um einige Methoden

Wir erkennen, dass jede Methode einen Namen hat, der mit einem Klammerpaar endet. Über den Namen kann die Methode aufgerufen werden. Falls im Inneren des Klammerpaars ein oder mehrere Parameter angegeben sind, so bedeutet dies, dass die Methode bei ihrem Aufruf entsprechende Argumente benötigt. So muss

Methodenaufruf

beim Aufruf der Methode `legeRabattFest()` ein `double`-Wert an das Objekt übergeben werden. Die Anweisung

```
rechnung1.legeRabattFest(0.05);
```

bewirkt, dass die Methode `legeRabattFest()` des Rechnung-Objekts `rechnung1` mit dem Parameter `0.05` ausgeführt wird. [JLS: § 6.2, § 15.12]

Bemerkung 17.1-2: Parameternamen

Parameternamen, wie sie in der UML und auch in anderen Dokumentationsarten für Methoden oft angegeben werden, dienen dem Verständnis der Methode. Für den Methodenaufruf sind sie bedeutungslos. ┘

Die Methode `fuegePostenHinzu()` benötigt als erstes Argument einen `int`-Wert für die Anzahl und als zweites Argument einen `double`-Wert für den Einzelpreis eines neuen Rechnungspostens:

```
rechnung1.fuegePostenHinzu(21, 1.50);
```

Bemerkung 17.1-3: Reihenfolge bei der Parameterübergabe

Parameterreihenfolge *Die Parameter müssen bei einem Methodenaufruf stets in der dokumentierten Reihenfolge übergeben werden.* ┘

Ergebnis einer Methode Methoden können weiterhin ein Ergebnis zurück liefern, das außerhalb des Objekts weiter verwendet werden kann. Falls eine Methode ein Ergebnis zurück liefert, ist dies in einem UML-Klassendiagramm am Rückgabotyp zu erkennen, der hinter dem Klammerpaar und einem Doppelpunkt angegeben ist. Die Methode `berechneNettopreis()` liefert einen `double`-Wert an die aufrufende Stelle zurück, der beispielsweise in einer Zuweisung weiter verwendet werden kann:

Rückgabotyp

```
double nettopreis = rechnung1.berechneNettopreis();
```

Für die Übergabe von Argumenten wie für die Auswertung von Rückgabewerten gelten die Regeln der Typverträglichkeit.

Selbsttestaufgabe 17.1-1:

Welche der folgenden Zeilen sind korrekte (ausführbare) Anweisungen?

```
rechnung1.betrag = 32.20 + 2.40;           // 1
rechnung1.betrag = 50;                     // 2
rechnung1.fuegePostenHinzu(3);             // 3
rechnung1.legeRabattFest(0);               // 4
rechnung1.berechneNettopreis(3 * 3.50);    // 5
```



17.2 Erzeugung und Lebensdauer von Objekten

Im Unterschied zu Werten primitiver Datentypen haben Objekte eine Lebensdauer. Ein Objekt muss explizit erzeugt werden, bevor es einer Variablen zugewiesen werden kann. Nach seiner Erzeugung existiert ein Objekt solange, wie ein Verweis auf das Objekt existiert. Ein solcher Verweis ist beispielsweise eine andere Variable, der dasselbe Objekt zugewiesen ist (vgl. Abschnitt 17.3). Ein Objekt, auf das kein Verweis mehr existiert, wird von Javas automatischer Speicherverwaltung (engl. *garbage collection*) aus dem Programmsystem entfernt.

Objekterzeugung

Zur Erzeugung von Objekten einer bestimmten Klasse wird ein Konstruktor dieser Klasse aufgerufen. Ein Konstruktor ist eine spezielle Methode, die dazu dient, ein Objekt zu initialisieren, d. h. in einem definierten Anfangszustand in der Laufzeitumgebung bereitzustellen.

In Java lautet der Name eines Konstruktors stets gleich wie der Klassenname. In der UML können Konstrukturen zudem durch das Schlüsselwort «create» gekennzeichnet werden. Das Klammerpaar eines Konstruktors kann leer sein oder eine Parameterliste enthalten. Das UML-Klassendiagramm in Abb. 17.2-1 ist um einen parameterlosen Konstruktor der Klasse Rechnung erweitert.

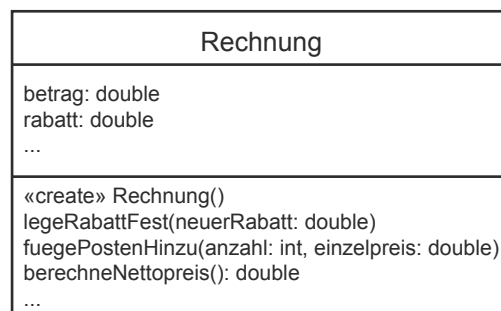


Abb. 17.2-1: Die Klasse Rechnung, erweitert um einen Konstruktor

Zur Erzeugung eines Objekts wird dem Konstruktor das Schlüsselwort `new` vorangestellt [JLS: § 15.9]. Ein Objekt vom Typ `Rechnung` können wir mit dem Ausdruck `new Rechnung()` erzeugen. Die Erzeugung wird in der Regel mit einer Zuweisung verbunden. Durch die Anweisung

Konstruktoraufruf
`new`

```
Rechnung rechnung2 = new Rechnung();
```

wird eine Variable vom Typ `Rechnung` deklariert und dieser ein neu erzeugtes `Rechnung`-Objekt zugewiesen. Die Zuweisung kann auch an eine bereits zuvor deklarierte Variable erfolgen:

```
Rechnung rechnung3;
...
rechnung3 = new Rechnung();
```

Selbsttestaufgabe 17.2-1:

Kopieren Sie die Datei `Rechnung.java` von unserer Kursseite in ein Arbeitsverzeichnis und übersetzen Sie sie. Legen Sie in demselben Verzeichnis ein Programmgerüst mit einer `main()`-Methode an. Verfassen Sie innerhalb der `main()`-Methode eine Anweisungssequenz, die eine Variable vom Typ `Rechnung` mit einem `Rechnung`-Objekt initialisiert, dem `Rechnung`-Objekt einige `Rechnung`-Posten hinzufügt, einen `Rabatt` festlegt und zuletzt den `Nettopreis` am Bildschirm ausgibt. Kompilieren Sie Ihr Programm und führen Sie es aus. ◇

Wie wir gesehen haben, werden Variablen für Objekttypen syntaktisch ebenso vereinbart wie Variablen für primitive Typen. Es bestehen jedoch zwei bedeutsame Unterschiede zwischen Variablen für Objekttypen und Variablen für primitive Typen.

Zunächst kann eine Variable eines Objekttyps nur dann deklariert werden, wenn dieser Objekttyp im Kontext der Deklaration definiert ist. (Diese Einschränkung gilt verständlicherweise auch für die Erzeugung von Objekten.) Demgegenüber sind primitive Datentypen in Java global verfügbar, so dass Variablen – wie auch Literale – primitiver Datentypen an jeder Stelle eines Java-Programms auftreten können.

Um den zweiten, besonders gewichtigen Unterschied zu verstehen, müssen wir uns etwas genauer mit der Beschaffenheit von Variablen auseinandersetzen. Diesem Thema ist der folgende Abschnitt gewidmet.

17.3 Wertvariablen und Verweisvariablen

Eine Variable ist eine benannte Speicherstelle im Arbeitsspeicher eines Rechners. Im Falle von primitiven Datentypen befindet sich an dieser Speicherstelle jeweils der Wert, der einer Variablen zugewiesen ist [JLS: § 4.12.1]. Abb. 17.3-1 zeigt schematisch die `int`-Variable `laenge1` mit dem Wert 100.

<code>int laenge1</code>	100
--------------------------	-----

Abb. 17.3-1: Eine `int`-Variable mit einem Wert

Im Falle von Objekttypen befindet sich an der zugehörigen Speicherstelle nun aber nicht, wie man vielleicht vermuten könnte, das Objekt, das einer Variablen zugewiesen ist, sondern lediglich eine Referenz auf einen anderen Speicherbereich, der das eigentliche Objekt beherbergt [JLS: § 4.12.2]. Abb. 17.3-2 zeigt schematisch die Variable `rechnung1` vom Typ `Rechnung`. Das ihr zugewiesene `Rechnung`-Objekt befindet sich in einem anderen, von der virtuellen Maschine verwalteten Speicherbereich, auf dessen Adresse die Speicherstelle von `rechnung1` verweist.

Wertvariable Zur Unterscheidung dieser beiden Arten von Variablen sprechen wir von Wertva-

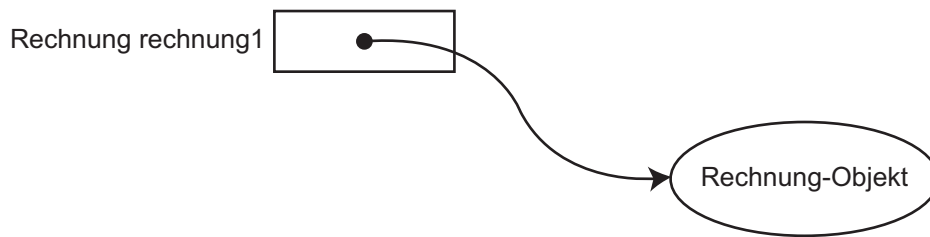


Abb. 17.3-2: Eine Rechnung-Variable mit einer Referenz

riablen und Verweisvariablen. Alle Variablen für primitive Datentypen sind Wertvariablen. Alle Variablen für Objekttypen sind Verweisvariablen.

Der entscheidende Unterschied zwischen Wertvariablen und Verweisvariablen tritt bei Auswertungen, beispielsweise im Zusammenhang mit Zuweisungen der Art

```
var1 = var2;
```

zu Tage. Eine solche Zuweisung bewirkt, dass der Inhalt der Speicherstelle von `var2` in die Speicherstelle von `var1` kopiert wird.

Handelt es sich bei `var1` und `var2` um Variablen eines primitiven Datentyps (und somit um Wertvariablen), so wird der Wert von `var2` in die Speicherstelle von `var1` kopiert (Abb. 17.3-3).

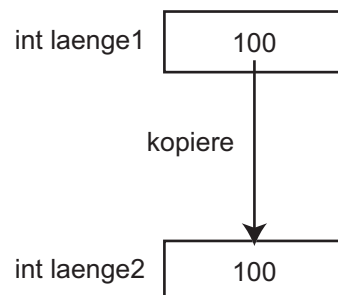


Abb. 17.3-3: Kopiervorgang zwischen Wertvariablen

Nach Ausführung der Anweisungsfolge

```
int laenge1 = 100;
int laenge2 = laenge1;
```

sind sowohl `laenge1` als auch `laenge2` mit dem Wert 100 belegt. (Die Zuweisung in der zweiten Zeile entspricht dem in Abb. 17.3-3 gezeigte Kopiervorgang.) Wie wir wissen, sind diese Werte zwar gleich, aber voneinander vollkommen unabhängig. So bewirkt die nachfolgende Anweisung

```
laenge2 += 300;
```

dass `laenge2` den Wert 400 erhält, während `laenge1` weiterhin mit dem Wert 100 belegt ist.

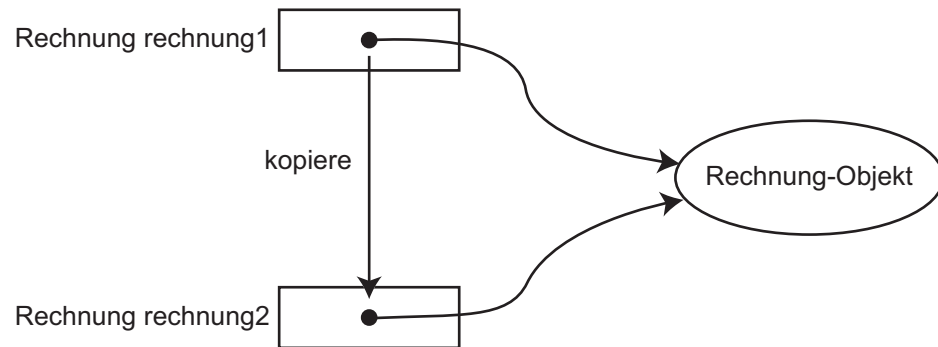


Abb. 17.3-4: Kopiervorgang zwischen Verweisvariablen

Handelt es sich bei `var1` und `var2` jedoch um Variablen eines Objekttyps (und somit um Verweisvariablen), so wird die Referenz auf ein Objekt in die Speicherstelle von `var1` kopiert (Abb. 17.3-4).

Das Resultat des Kopiervorgangs sind somit zwei identische Referenzen, so dass `rechnung1` und `rechnung2` auf ein und dasselbe `Rechnung-Objekt` verweisen.

Während die Wertvariablen `laenge1` und `laenge2` nach dem Kopiervorgang voneinander unabhängig geblieben sind, sind die Verweisvariablen durch den Kopiervorgang miteinander gekoppelt worden. Nach Ausführung der Anweisungsfolge

```
Rechnung rechnung1 = new Rechnung();
rechnung1.betrag = 100.0;
Rechnung rechnung2 = rechnung1;
rechnung2.betrag += 300.0;
```

ist sowohl `rechnung1.betrag` als auch `rechnung2.betrag` mit dem Wert `400.0` belegt, denn es handelt sich bei `rechnung1.betrag` und `rechnung2.betrag` um ein und dasselbe Attribut eines einzigen Objekts. Betrachten wir die Vorgänge Schritt für Schritt:

- `new Rechnung()` veranlasst die Erzeugung eines `Rechnung-Objekts` in einem von der virtuellen Maschine verwalteten Speicherbereich und liefert eine Referenz auf das Objekt. Eine `Rechnung-Variable` `rechnung1` wird deklariert und erhält durch Zuweisung diese Referenz.
- Dem `betrag`-Attribut des von `rechnung1` referenzierten `Rechnung-Objekts` wird der Wert `100.0` zugewiesen.
- Eine weitere `Rechnung-Variable` `rechnung2` wird deklariert und erhält durch Zuweisung eine Kopie der in `rechnung1` gespeicherten Referenz. Die Zuweisung entspricht dem in Abb. 17.3-4 gezeigte Kopiervorgang. Sie hat zur Folge, dass beide Variablen `rechnung1` und `rechnung2` auf dasselbe `Rechnung-Objekt` verweisen.
- Der Wert des `betrag`-Attributs dieses nun von `rechnung1` und `rechnung2` referenzierten `Rechnung-Objekts` wird um `300.0` erhöht.

Die Koppelung zwischen zwei Verweisvariablen, die dasselbe Objekt referenzieren, wird bei einer Zuweisung an eine der beiden Variablen wieder aufgehoben. Nach Ausführung der Anweisung

```
rechnung2 = new Rechnung();
```

verweisen `rechnung1` und `rechnung2` auf zwei separate Rechnung-Objekte mit jeweils eigenen, unabhängigen Zuständen (Abb. 17.3-5).

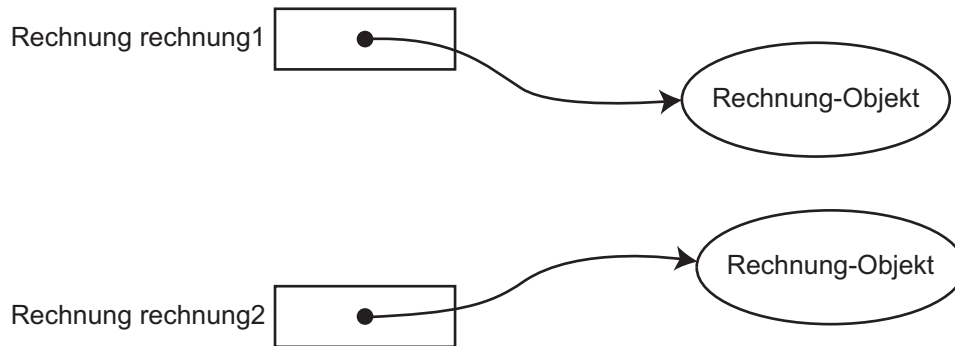


Abb. 17.3-5: Zwei unabhängige Verweisvariablen

Ob zwei Verweisvariablen dasselbe Objekt oder zwei separate Objekte referenzieren, kann mit den Operatoren `==` bzw. `!=` überprüft werden [JLS: § 15.21.3]. Der Ausdruck

```
rechnung1 == rechnung2
```

liefert dann und nur dann `true`, wenn Objekt-Identität vorliegt, d. h. wenn beide Variablen dasselbe Objekt referenzieren. Umgekehrt liefert der Ausdruck

```
rechnung1 != rechnung2
```

genau dann `true`, wenn die beiden Variablen zwei separate Objekte referenzieren. Beachten Sie, dass zwei separate Objekte, deren Zustand (Attributwerte) gleich ist, nicht identisch sind. Nach der Ausführung von

```
Rechnung rechnung3 = new Rechnung();
Rechnung rechnung4 = new Rechnung();
boolean sindIdentisch = (rechnung3 == rechnung4);
```

hat `sindIdentisch` den Wert `false`. Zwar wurden beide Rechnung-Objekte mit gleichen (durch die Klasse `Rechnung` definierten) Anfangszuständen erzeugt und nicht weiter verändert, doch verweisen die beiden Variablen `rechnung3` und `rechnung4` auf zwei separate Objekte; ihre Referenzen sind nicht identisch.

Eine Verweisvariable kann auch den undefinierten Verweis `null` enthalten [JLS: § 3.10.7]. Man spricht auch von einer `null`-Referenz.

Abb. 17.3-6 zeigt die Variable `rechnung5` vom Typ `Rechnung`, die auf kein Objekt verweist.

Rechnung rechnung5

**Abb. 17.3-6:** Eine null-Referenz

Das Schlüsselwort `null` können wir in Verbindung mit einem Gleichheits- oder Ungleichheitsoperator sowie in Zuweisungen verwenden. Der Ausdruck

```
rechnung5 == null
```

liefert `true`, falls `rechnung5` auf kein Objekt verweist. Der Ausdruck

```
rechnung5 != null
```

liefert in demselben Fall entsprechend `false`. Wird ein referenziertes Objekt nicht mehr benötigt, so kann es durch

```
rechnung5 = null
```

von der referenzierenden Variablen gelöst werden. Falls es keine anderen Verweise auf dasselbe Objekt gibt, wird es in der Folge durch die automatische Speicherverwaltung entfernt.

Wie Variablen für primitive Datentypen können auch Variablen für Objekttypen mit dem Schlüsselwort `final` versehen werden:

```
final Rechnung RECHNUNG;
```

Nachdem der finalen Variablen `RECHNUNG` zum ersten Mal ein `Rechnung`-Objekt zugewiesen worden ist, kann keine erneute Zuweisung an diese Variable erfolgen. Beachten Sie jedoch, dass der Zustand des zugewiesenen `Rechnung`-Objekts weiterhin veränderlich ist.

Selbsttestaufgabe 17.3-1:

Stellen Sie, ähnlich wie in Abb. 17.3-4 und Abb. 17.3-5, dar, welche Variablen nach der Ausführung der folgenden Anweisungen auf welche Objekte verweisen. Welche Werte besitzen die `betrag`-Attribute der Objekte?

```
Rechnung re1 = new Rechnung();
re1.betrag = 12.0;
Rechnung re2 = new Rechnung();
re2.betrag = 24.0;
Rechnung re3 = new Rechnung();
re3.betrag = 36.0;
re2.betrag = re3.betrag;
re3 = re2;
re1.betrag = re3.betrag;
```



17.4 Zusammenspiel von Objekten

Wir beenden dieses Kapitel mit einem Anwendungsbeispiel, das in das Zusammenspiel von Objekten einführt. Das UML-Klassendiagramm in Abb. 17.4-1 zeigt noch einmal die Klasse Rechnung, weiterhin eine neue Klasse Kunde und eine Assoziation zwischen diesen beiden Klassen.

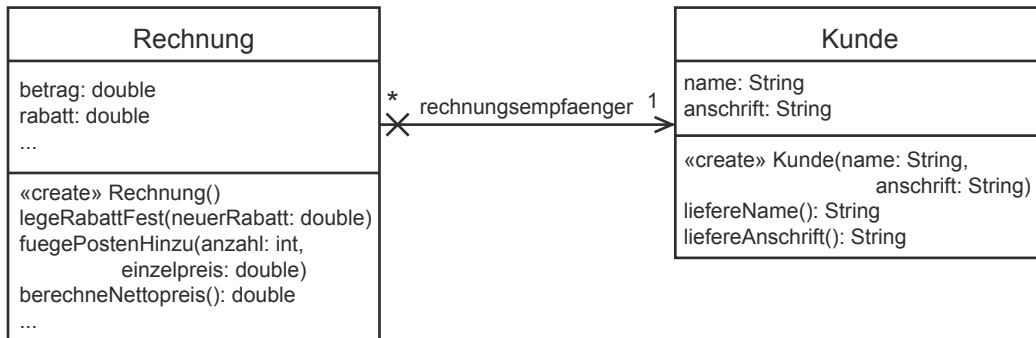


Abb. 17.4-1: Die Klassen Rechnung und Kunde

Betrachten wir zunächst die neue Klasse Kunde. Sie verfügt über zwei Attribute eines Typs `String`. Der Typ `String` wird auch in den Parametern des Konstruktors erwartet und tritt in den Rückgabewerten der beiden Methoden auf.

Die Klasse `String` ist Bestandteil der Java-Laufzeitbibliothek (API) und dient zur Verwaltung von Zeichenketten [JLS: § 4.3.3]. In einer späteren Kurseinheit werden wir die Klasse `String` genauer kennen lernen. An dieser Stelle wollen wir lediglich zwei `String`-Objekte für die Zeichenketten "Anna Müller" und "Mühlenweg 2, 12345 Irgendwo" erzeugen, um sie den beiden Attributen eines `Kunde`-Objekts zuweisen zu können. `String`-Objekte können auf verschiedene Weise erzeugt werden, wir zeigen zwei Varianten:

Klasse `String`

```
String annasName = new String("Anna Müller");
String annasAnschrift = "Mühlenweg 2, 12345 Irgendwo";
```

Die Syntax der ersten Zeile ist uns im Wesentlichen vertraut: Durch das Schlüsselwort `new` in Verbindung mit einem Konstruktor der Klasse `String` wird ein `String`-Objekt erzeugt. Dem Konstruktor wird ein `String`-Literal übergeben. Die zweite Zeile zeigt eine Kurzform zur Erzeugung von `String`-Objekten. Hierbei handelt es sich um eine syntaktische Besonderheit der Klasse `String`. [JLS: § 3.10.5]

`String`-Literal

Mit Hilfe dieser beiden `String`-Objekte lässt sich ein `Kunde`-Objekt wie folgt erzeugen:

```
Kunde kunden1 = new Kunde(annasName, annasAnschrift);
```

Um die Kundendaten abzufragen, sind Objekte der Klasse Kunde mit zwei Methoden `liefereName()` und `liefereAnschrift()` ausgestattet. Die Anweisung

```
String nameDesKunden = kunde1.liefereName();
```

liefert die Zeichenkette "Anna Müller" an eine neue String-Variable `nameDesKunden`, die außerhalb des Kunde-Objekts weiter verwendet werden kann.

Die Assoziation zwischen den Klassen Kunde und Rechnung in Abb. 17.4-1 lässt uns erkennen, dass Rechnung-Objekte jeweils genau ein Kunde-Objekt kennen, während Kunde-Objekte bei beliebig vielen Rechnung-Objekten bekannt sein können, aber diese selbst nicht kennen. Die Umsetzung dieses Sachverhalts in Java geschieht dadurch, dass die Klasse Rechnung über ein Attribut `rechnungsempfaenger` vom Typ Kunde verfügt.

Erstmalig treffen wir hier auf den Sachverhalt, dass eine Klasse über ein Attribut einer anderen Klasse verfügt. Solche Konstrukte sind charakteristisch für die objektorientierte Programmierung.

Bemerkung 17.4-1: Attribute in UML-Klassendiagrammen

Attribute primitiver Datentypen werden in der Attributliste einer Klasse aufgeführt. Ein Attribut eines Objekttyps kann entweder in der Attributliste aufgeführt oder durch eine Assoziation kenntlich gemacht werden. Die Darstellung durch eine Assoziation empfiehlt sich insbesondere bei der gleichzeitigen Darstellung des zugehörigen Objekttyps.

┘

Wir erzeugen ein neues Rechnung-Objekt und weisen seinem Attribut `rechnungsempfaenger` vom Typ Kunde das Objekt `kunde1` zu:

```
Rechnung rechnung6 = new Rechnung();
rechnung6.rechnungsempfaenger = kunde1;
```

Erinnern wir uns an dieser Stelle, dass in einer Variablen eines Objekttyps ein Verweis auf ein Objekt (oder die null-Referenz) gespeichert ist. Sollte das Rechnung-Objekt `rechnung6` eine Änderung am Kunde-Objekt seines Attributs `rechnungsempfaenger` vornehmen, so wäre unmittelbar das von `kunde1` referenzierte Objekt verändert. Der umgekehrte Fall ist ebenso möglich, wie wir abschließend kurz demonstrieren.

Wir wollen unserem Objekt `rechnung6` noch einen Rechnungsposten hinzufügen, einen Rabatt festlegen und anschließend einen Ausdruck der Rechnung anfordern:

```
rechnung6.fuegePostenHinz(25, 1.76);
rechnung6.legeRabattFest(0.04);
rechnung6.gebeAus();
```

Die Anweisung in der letzten Zeile veranlasst den folgenden Bildschirmausdruck:

```
An:
Anna Müller
Mühlenweg 2, 12345 Irgendwo
Netto: 42.24
```

Nehmen wir nun an, die Anschrift der Kundin Anna Müller hätte sich geändert. Wir passen das Objekt `kunde1` entsprechend an und lassen anschließend erneut einen Rechnungsausdruck erstellen:

```
kunde1.anschrift = "Gartenstr. 13, 12345 Irgendwo";
rechnung6.gebeAus();
```

Da wir wissen, dass das Objekt `rechnung6` eine Referenz auf dasselbe Objekt `kunde1` enthält, dessen Attribut `anschrift` wir soeben manipuliert haben, verwundert uns der resultierende Bildschirmausdruck nicht:

```
An:
Anna Müller
Gartenstr. 13, 12345 Irgendwo
Netto: 42.24
```

Auf eine Gefahr im Umgang mit Verweisvariablen wollen wir noch aufmerksam machen. Sei `rechnung7` ein Rechnung-Objekt, und wir wollen wissen, an welchen Kunden die Rechnung gerichtet ist. Wir verschaffen uns zunächst eine Referenz auf das Attribut `rechnungsempfaenger`:

```
Kunde werAuchImmer = rechnung7.rechnungsempfaenger;
```

Sodann versuchen wir, mit der Anweisung

```
String name = werAuchImmer.name;
```

den Namen des Kunden in Erfahrung zu bringen. Im ungünstigeren Fall scheitert die Auswertung von `werAuchImmer.name` zur Laufzeit mit der Fehlermeldung

```
NullPointerException
```

Der Fehler tritt auf, wenn das Attribut `rechnung7.rechnungsempfaenger` und damit auch unsere Variable `werAuchImmer` eine `null`-Referenz enthält [JLS: § 15.11.1]. Zwar kann eine `null`-Referenz problemlos an eine Kunde-Variable zugewiesen werden, doch nur ein Kunde-Objekt hätte ein `String`-Attribut `name`. Ein Methodenaufruf an einer `null`-Referenz würde zu demselben Problem führen. [JLS: § 15.12.4.4]

`NullPointerException`

Im Allgemeinen können wir nicht sicher wissen, ob sich hinter einer Verweisvariablen ein Objekt oder eine `null`-Referenz verbirgt. Es ist deshalb im Sinne einer robusten Programmierung ratsam, durch eine Konstruktion wie

```
String name;  
if (werAuchImmer != null) {  
    name = werAuchImmer.name;  
} else {  
    name = "kein Empfänger";  
}
```

diese Unsicherheit abzufangen.

Selbsttestaufgabe 17.4-1:

Erzeugen Sie zwei Kunde-Objekte kunde2 und kunde3 mit beliebigen Namen und Anschriften. Schreiben Sie eine Anweisung, die dem Objekt kunde2 die Anschrift von kunde3 zuweist.



Selbsttestaufgabe 17.4-2:

Können Sie einem Rechnung-Objekt auch zwei Kunde-Objekte zuweisen? Falls ja, schreiben Sie eine passende Anweisungsfolge. Falls nein, begründen Sie.



Selbsttestaufgabe 17.4-3:

Können Sie ein einziges Kunde-Objekt an zwei verschiedene Rechnung-Objekte zuweisen? Falls ja, schreiben Sie eine passende Anweisungsfolge. Falls nein, begründen Sie.



Selbsttestaufgabe 17.4-4:

Erzeugen Sie ein Rechnung-Objekt, fügen Sie einige Rechnungsposten hinzu und legen Sie einen Rabatt fest. Weisen Sie kein Kunde-Objekt zu. Was passiert, wenn nun die Methode gebeAus () ausgeführt wird?



18 Klassenelemente

Im letzten Kapitel haben wir bestehende Klassen genutzt. Im Laufe dieses Kapitels werden wir lernen, wie in Java Klassen deklariert werden und welche Bestandteile in einer solchen Deklaration enthalten sein können.

Klassen sind die wichtigsten Bausteine in Java-Programmen. Eine Klasse ist ein Gebilde, das dazu dient, Objekte mit der gleichen Menge von Attributen und Methoden zu definieren. Der Klassenname bezeichnet typischerweise ein Fachkonzept.

Eine Klasse ist ein Vorbild, das beschreibt, wie ein Objekt auszusehen hat, wenn es erzeugt wird. Das Objekt ist ein Exemplar (engl. *instance*) der Klasse. Wir können uns eine Klasse einerseits als eine Fabrik vorstellen, die Objekte erzeugt, und auf der anderen Seite als Schablone, die die Eigenschaften und das Verhalten der Objekte beschreibt.

18.1 Klassenvereinbarung

Klassen sind der hauptsächliche Strukturierungs- und Abstraktionsmechanismus in Java. Klassendeklarationen verkörpern das gesamte Wissen über Objekte, da alle Objekte Exemplare von Klassen sind.

Jede Klasse muss deklariert werden, damit sie benutzt werden kann. In diesem und in nachfolgenden Abschnitten lernen wir nach und nach die Anatomie von Klassenvereinbarungen im Detail kennen, um es Ihnen so zu ermöglichen eigene Klassen für gegebene Probleme zu erstellen. Die Bestandteile einer solchen Vereinbarung ähneln den Elementen einer Klasse in der UML.

Definition 18.1-1: Klassenvereinbarung

In der Regel hat eine Klassenvereinbarung die folgende Struktur:

Klassenvereinbarung

```
class Klassenname {
    Attributdeklarationen
    Konstruktordeklarationen
    Methodendeklarationen
}
```

Dabei sind die drei Bestandteile im Inneren der Klasse optional. [JLS: § 8.1] ┘

Eine Klassenvereinbarung (engl. *class declaration*) beginnt mit dem Schlüsselwort `class`. Eine Klassenvereinbarung legt einen neuen Objekttyp fest und spezifiziert seine Implementation im Rumpf der Klassenvereinbarung. Die Bestandteile des Rumpfs werden wir in den nächsten Abschnitten kennen lernen.

`class`



Namen von Klassen sollten mit einem Großbuchstaben beginnen.

18.2 Attributdeklaration

Wir erinnern uns, dass die Attribute (engl. *attributes*, *fields*) einer Klasse beschreiben, welche Eigenschaften die Objekte dieser Klasse aufweisen. Die konkreten Werte der Attribute können sich von Objekt zu Objekt unterscheiden. Wie schon in der UML müssen wir auch in Java die Attribute der Klasse deklarieren. Die Klasse `Rechnung` besitzt beispielsweise die Attribute `betrag`, `rabatt` und `mehrwertsteuer`. Wie Variablen haben auch Attribute einen Typ und einen Namen. Der Typ charakterisiert sowohl die Menge von Werten, die das Attribut annehmen kann, als auch die Menge von Operatoren oder Methoden, die auf diese Attribute anwendbar sind. Eine Attributdeklaration hat die gleiche Form wie eine Variablendeklaration (vgl. Kapitel 10), sie besteht folglich aus dem Typen und dem Namen des Attributs [JLS: § 8.3]. Sie befindet sich direkt innerhalb des Klassentrumpfes. Attribute können sowohl Wert- als auch Verweisvariablen sein. In unserem Beispiel sind alle Attribute vom Typ `double`.

Attributdeklaration

```
class Rechnung {
    // Attribute
    double betrag = 0;
    double mehrwertsteuer;
    double rabatt;

    // ...
}
```

 Für Attributnamen gelten die gleichen Richtlinien wie für Variablennamen.

Bemerkung 18.2-1: Initialisierung von Attributen

Eine Attributdeklaration kann genauso wie eine Variablendeklaration eine Initialisierung beinhalten. Enthält eine Attributdeklaration keine Initialisierung und wird das Attribut vor seiner Benutzung nicht anderweitig initialisiert, so besitzt es einen Standardwert. Dieser ist bei allen numerischen Datentypen und Zeichen 0 bzw. 0.0, bei `boolean` der Wert `false` und bei Verweisvariablen `null`. [JLS: § 8.3.2, § 4.12.5]

Initialisierung von
Attributen

Standardbelegung von
Attributen

Bemerkung 18.2-2: Exemplarvariable

Wir nennen Attribute auch Exemplarvariablen, da es sich in Java bei Attributen um Variablen handelt, die genau einmal pro Exemplar existieren. In der Literatur werden Sie auch oft den Begriff Instanzvariable finden.

Exemplarvariable

finale Attribute

Ein Attribut kann, ebenso wie eine Variable, als `final` deklariert werden. Sein Attributwert kann und muss dann genau einmal explizit festgelegt werden, danach ist er unveränderlich. Die Festlegung kann durch eine Initialisierung in der Deklaration oder beim Erzeugen des Objekts (siehe Abschnitt 18.4) erfolgen. Eine Standardbelegung gibt es bei als `final` deklarierten Variablen nicht. [JLS: § 8.3.1.2]

Selbsttestaufgabe 18.2-1:

Deklarieren Sie eine Klasse `Kreis` mit dem Attribut `radius` sowie die Klasse `Rechteck` mit den Attributen `laenge` und `breite`.

**Selbsttestaufgabe 18.2-2:**

Deklarieren Sie die Klasse `Kunde` mit den Attributen `name` und `anschrift`.



Wie wir schon im vorangegangenen Kapitel gelernt haben, können Assoziationen in Java in Form von Exemplarvariablen umgesetzt werden.

Selbsttestaufgabe 18.2-3:

Versuchen Sie, die gerichtete Assoziation zwischen den Klassen `Rechnung` und `Kunde` in Java zu implementieren.

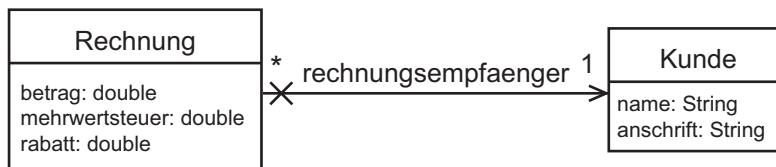


Abb. 18.2-1: Die Assoziation zwischen `Rechnung` und `Kunde`



Im Unterschied zu Attributen, die die Eigenschaften beschreiben, implementieren Methoden das Verhalten, das für alle Objekte dieser Klasse gültig ist. Die Deklaration von Methoden wird im folgenden Abschnitt behandelt.

18.3 Methodendeklaration

Nachdem wir uns zuvor mit der Deklaration von Attributen beschäftigt haben, wenden wir uns jetzt der Deklaration von Methoden (engl. *method*) und der Verhaltensbeschreibung im Methodenrumpf zu.

Eine Methode implementiert das gemeinsame Verhalten für alle Objekte einer Klasse. Methoden können den Zustand eines Objekts ändern, Informationen über das Objekt preisgeben oder neue Informationen erzeugen. Eine Methodendeklaration enthält im Allgemeinen ausführbaren Quelltext, der aus dem Kontext, in dem die Methodendeklaration gültig ist, aufgerufen werden kann.

Jede Methode besitzt einen Namen. Darüber hinaus kann sie einen Ergebnistyp festlegen, falls sie ein Ergebnis liefert, wie zum Beispiel eine Methode, die die Mehrwertsteuer der Rechnung berechnet. Eine Methode, die nur die Informationen zur Rechnung am Bildschirm ausgibt, besitzt keinen Ergebnistyp. Als Ergebnistyp sind

sowohl primitive Datentypen als auch Objekttypen zulässig. Des Weiteren kann eine Methode noch Parameter festlegen, die ihr bei einem Aufruf in Form von passenden Argumenten übergeben werden müssen.

Definition 18.3-1: Methodendeklaration

Methodendeklaration

Eine Methode hat einen Kopf und einen Rumpf. Die grundlegende Methodendeklaration hat folgende Form [JLS: § 8.4]:

```
ErgebnisTyp Methodenname(Parameterliste) {
    Anweisungen
}
```

Der Kopf in der ersten Zeile:

Ergebnistyp

void

- zeigt uns, ob die Methode ein Ergebnis liefert und, wenn ja, von welchem Typ das Ergebnis ist: ErgebnisTyp ist entweder ein in diesem Kontext sichtbarer Typname oder das Schlüsselwort void, das angibt, dass kein Ergebnis bei der Methodenausführung geliefert wird,

- führt einen neuen Namen Methodenname ein und

formale Parameter

- umfasst eine, möglicherweise leere Liste formaler Parameter (Parameterliste).

Eine Liste formaler Parameter hat die folgende Form:

```
t1 n1, ..., tm nm
```

wobei n1, ..., nm die paarweise verschiedenen Namen der formalen Parameter vom Typ t1, ..., tm sind.

Methodenrumpf

Der Methodenrumpf umfasst eine Folge von Anweisungen, wie wir sie in der vorangegangenen Kurseinheit behandelten.

┘

So können wir die Klasse Rechnung zum Beispiel um die Methode legeRabattFest(double neuerRabatt) ergänzen. Da die Methode kein Ergebnis erzeugt, ist sie als void deklariert.

```
class Rechnung {
    // Attribute
    double betrag = 0;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void legeRabattFest(double neuerRabatt) {
        // ...
    }

    // ...
}
```

Selbsttestaufgabe 18.3-1:

Ergänzen Sie die Deklaration der Klasse `Rechnung` um die Methode `legeMehrwertsteuerFest()`, die die neue Mehrwertsteuer als Parameter besitzt, sowie die Methode `fuegePostenHinzu()`, die den Stückpreis sowie die Stückzahl übergeben bekommt. Die Methodenrümpfe können Sie zunächst leer lassen.

**Definition 18.3-2: Signatur einer Methode**

In Java besteht die Signatur einer Methode aus dem Methodennamen sowie den Typen ihrer formalen Parameter in der Reihenfolge der Vereinbarung. [JLS: § 8.4.2]

Signatur einer Methode

Selbsttestaufgabe 18.3-2:

Bestimmen Sie die Signaturen der Methoden der Klasse `Kreis`.

```
class Kreis {
    // Methoden
    double berechneFlaecheninhalt() {
        // ...
    }

    double berechneUmfang() {
        // ...
    }
}
```

**Selbsttestaufgabe 18.3-3:**

Welche Methoden der Klasse `X` besitzen die gleiche Signatur?

```
class X {
    // Methoden
    void a(int x) {
        // ...
    }

    int a(double x) {
        // ...
    }

    int a(int u) {
        // ...
    }

    int b(double x, int y) {
        // ...
    }
}
```

```

void b(int x, double y) {
    // ...
}

void c(int a, int b) {
    // ...
}

void c(int a, int b, int c) {
    // ...
}

void d(int x) {
    // ...
}
}

```



überladener
Methodenname

In einer Klasse darf es niemals mehrere Methoden mit der gleichen Signatur geben, sonst führt dies zu einem Übersetzungsfehler. Da die Signatur sich aber aus dem Methodennamen und den Parametertypen zusammensetzt, ist es möglich, dass mehrere Methoden mit dem gleichen Namen, aber mit verschiedenen Parametertypen existieren. Man spricht in diesem Fall von einem überladenen Methodennamen [JLS: § 8.4.9]. Bei einem Aufruf mit diesem Methodennamen wird an Hand der Typen der Argumente entschieden, welche der Methoden ausgeführt wird.

Selbsttestaufgabe 18.3-4:

Benennen Sie alle überladenen Methodennamen der Klasse X.



lokale Variablen

Betrachten wir nun die Methode `gebeAus()`, die neben dem Namen und der Anschrift des Rechnungsempfängers noch den Nettopreis inklusive Rabatt, die Mehrwertsteuer und den Bruttopreis der Rechnung ausgeben soll. Sie muss dafür unter anderem auf die Werte der eigenen Attribute zugreifen. Ein solcher Zugriff erfolgt genauso wie ein Zugriff auf eine Variable. Der Zugriff auf Attribute des Kundenobjekts erfolgt wie in Kapitel 17 gezeigt. Zusätzlich benötigen wir in der Methode Hilfsvariablen, um zum Beispiel den Netto- und den Bruttopreis zu berechnen. Diese Variablen nennen wir lokale Variablen, da sie nur in der Methode sichtbar sind [JLS: § 14.4, § 4.12.13]. Die lokalen Variablen sind außerhalb der Methode nicht verfügbar. Innerhalb des Methodenrumpfes stehen Variablendeklarationen und Anweisungen, wie wir sie in der letzten Kurseinheit kennen gelernt haben.

```

class Rechnung {
    // Attribute
    double betrag = 0.0;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;
}

```

```

// Methoden
void gebeAus() {
    // lokale Variablen
    double netto = betrag * (1 - rabatt);
    double brutto = netto * (1 + mehrwertsteuer);

    System.out.println("An:");
    System.out.println(rechnungsempfaenger.name);
    System.out.println(rechnungsempfaenger.anschrift);
    System.out.print("Netto: ");
    System.out.println(netto);
    System.out.print("MwSt: ");
    System.out.println(mehrwertsteuer);
    System.out.print("Brutto: ");
    System.out.println(brutto);
}

// ...
}

```

Bemerkung 18.3-1: Initialisierung von lokalen Variablen

Alle lokalen Variablen müssen, anders als Attribute, vor ihrer ersten Verwendung explizit initialisiert werden.

┘

In der Methode `legeRabattFest(double neuerRabatt)` muss der im Parameter `neuerRabatt` übergebene Wert im Attribut `rabatt` gespeichert werden, so dass dieser nach der Methodenausführung im Zustand des Rechnungsobjekts gespeichert ist. Wir können in Methoden nicht nur den Wert von Attributen auslesen, sondern diesen auch verändern. Da Parameter lokale Variablen sind, erfolgt der Zugriff auf die gleiche Weise.

```

class Rechnung {
    // Attribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void legeRabattFest(double neuerRabatt) {
        rabatt = neuerRabatt;
    }

    // ...
}

```

 Einem Parameter kann wie jeder anderen lokalen Variable auch ein neuer Wert zugewiesen werden, jedoch sollte dies wenn möglich vermieden werden. Um eine

solche Zuweisung zu verhindern, kann ein Parameter als `final` deklariert werden, z.B. `void legeRabattFest(final double neuerRabatt)`. Finale Parameter werden nicht mit Großbuchstaben geschrieben, da es sich nicht um Konstanten im eigentlichen Sinne handelt.

Innerhalb von Methoden kann es nützlich sein, einen Verweis auf ein Objekt zu haben, an dem die Methode gerade aufgerufen wurde. Diese Referenz erhält man mit Hilfe des Schlüsselworts `this` [JLS: § 15.8.3]. Der Typ der `this`-Referenz entspricht immer der umgebenden Klasse.

☞ Um eindeutig zwischen dem Zugriff auf eine lokale Variable und dem Zugriff auf ein Attribut zu unterscheiden, setzen wir vor jeden Attributzugriff das Schlüsselwort `this`. Dadurch wird beim Lesen direkt klar, ob es sich bei der verwendeten Variable um ein Attribut oder eine lokale Variable handelt. Zudem wird der Quelltext weniger fehleranfällig, wenn später Attribute oder lokale Variablen hinzugefügt, entfernt oder umbenannt werden. Die Methode `legeRabattFest(final double neuerRabatt)` würde dann folgendermaßen lauten:

```
class Rechnung {
    // Attribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void legeRabattFest(final double neuerRabatt) {
        this.rabatt = neuerRabatt;
    }

    // ...
}
```

Selbsttestaufgabe 18.3-5:

Implementieren Sie die Rümpfe der beiden Methoden `fuegePostenHinzu()` und `legeMehrwertsteuerFest()`.



Bemerkung 18.3-2: Verdeckung von Attributen

Wenn in einer Methode eine lokale Variable den gleichen Namen wie ein Attribut trägt, so akzeptiert der Übersetzer dies, allerdings wird das Attribut innerhalb dieser Methode von der lokalen Variable verdeckt (engl. shadow) [JLS: § 14.4.3]. Steht bei einer Namensgleichheit von Attribut und lokaler Variable kein `this` vor dem Zugriff, so erfolgt der Zugriff auf die lokale Variable. Mit Hilfe von `this` kann immer auf das Attribut zugegriffen werden.



Selbsttestaufgabe 18.3-6:

Benennen Sie alle Attribute und lokale Variablen der Klasse A. Bei welchen lokalen Variablen handelt es sich um Parameter?

```
class A {

    // Attribute
    double a;
    int b;
    String c;
    int d;

    // Methoden
    void k(int y, double d) {
        int x = 10, z = 2;
        double m = x * z / 3.0;
    }
    void x(int a, int b) {
        double c = 3;
        System.out.println(this.c);
        double f = a * this.b + c * b;
        double g = this.a + d;
    }
}
```

**Selbsttestaufgabe 18.3-7:**

Bei welchen Zugriffen in der Methode x() der Klasse A aus Selbsttestaufgabe 18.3-6 handelt es sich um Zugriffe auf lokale Variablen und bei welchen um Zugriffe auf Attribute?



Wenn eine Methode einen Ergebnistyp besitzt, so muss auch ein Wert an den Aufrufer zurückgegeben werden. Dies geschieht mit Hilfe einer `return`-Anweisung.

Definition 18.3-3: return- oder Rückgabeanweisung

Eine return-Anweisung im Rumpf einer Methode dient zwei Zwecken:

- sie definiert einen Wert, dessen Typ verträglich sein muss mit dem Ergebnistyp der Methode in der sie vorkommt; dieser Wert ersetzt nach Ausführung der Methode den Methodenaufruf;
- sie gibt die Kontrolle an die Aufrufstelle zurück.

return-Anweisung

Rückgabeanweisung

Nach einer return-Anweisung (im Sinne eines Ablaufpfads der Methode) dürfen keine weiteren Anweisungen vorkommen. Sie würden gemäß der Semantik der return-Anweisung nicht ausgeführt, und sie werden vom Java-Übersetzer auch nicht akzeptiert.

Eine return-Anweisung hat die allgemeine Form:

```
return Ausdruck;
```

Wenn die return-Anweisung im Rumpf einer void-Methode benutzt wird, also einer Methode ohne Rückgabewert, darf kein Ausdruck in der return-Anweisung erscheinen. Andernfalls wird ein Übersetzerfehler auftreten. [JLS: § 14.17] ┘

Wollen wir nun eine Methode `liefereRabatt()` implementieren, so muss diese den aktuellen Wert des Attributs `rabatt` zurück liefern.

```
class Rechnung {
    // Attribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    double liefereRabatt() {
        return this.rabatt;
    }

    // ...
}
```

Bemerkung 18.3-3: return-Anweisung in einer Methode ohne Ergebnistyp

Eine return-Anweisung in einer Methode ohne Ergebnistyp liefert keinen Wert an den Aufrufer zurück, sie ermöglicht aber, zum Beispiel im Falle von ungültigen Argumenten, die Methode vorzeitig zu beenden. Am Ende jeder Methode ohne Ergebnistyp steht implizit eine return-Anweisung. ┘

Bemerkung 18.3-4: return-Anweisungen einer Methode mit Ergebnistyp

In einer Methode mit Ergebnistyp muss am Ende jedes möglichen Ablaufpfades eine return-Anweisung mit einem passenden Ausdruck stehen. ┘

Bemerkung 18.3-5: Getter- und Setter-Methoden

Getter-Methode
Setter-Methode

In den meisten Klassen haben wir Methoden, die das Auslesen und Verändern von Attributen ermöglichen, wie zum Beispiel die Methoden `liefereRabatt()` und `legeRabattFest(final double neuerRabatt)`. Im Englischen wählt man für solche Methoden in der Regel Namen wie `getDiscount()` und `setDiscount(final double newDiscount)`. Deswegen werden diese Methoden auch als Getter- und Setter-Methoden bezeichnet.

Getter- und Setter-Methoden dienen der Wahrung des Geheimnisprinzips. Für die sichere Handhabung von Objekten einer nicht genauer bekannten Klasse ist es von Vorteil, wenn der Aufrufer nicht direkt auf die Attribute zugreift, sondern stattdessen Getter- und Setter-Methoden aufruft. Er muss dann nicht wissen, wie die Attribute intern implementiert sind. Das Verbergen von Klasseninterna ist das Hauptziel des Geheimnisprinzips. Selbst wenn in der Klassendefinition Veränderungen an der Implementierung der Attribute vorgenommen werden, wird der Aufrufer der Methode davon nicht beeinflusst. In einem späteren Kapitel werden wir Mechanismen kennen lernen, die den direkten Zugriff auf Attribute ganz unterbinden. Setter-Methoden können weiterhin so gestaltet werden, dass sie eine Überprüfung beinhalten, ob es sich bei dem neuen Wert für ein Attribut um einen zulässigen Wert handelt. ┘

Geheimnisprinzip

Selbsttestaufgabe 18.3-8:

Ändern Sie die Methode `legeRabattFest(final double neuerRabatt)` so, dass der gespeicherte Rabatt nur verändert wird, wenn der neue Rabatt nicht negativ und nicht größer als 50 % ist.



Selbsttestaufgabe 18.3-9:

Ergänzen Sie die Klasse `Rechnung` um Getter-Methoden für alle Attribute außer dem Rechnungsempfänger sowie um die nachfolgend beschriebenen Methoden.

- `berechneNettopreis()`: berechnet den aktuellen Nettopreis, also Betrag abzüglich Rabatt
- `berechneMehrwertsteuer()`: berechnet die Mehrwertsteuer, die für den aktuellen Nettopreis fällig wäre
- `berechneBruttopreis()`: berechnet den Bruttopreis, also Betrag abzüglich Rabatt, zuzüglich Mehrwertsteuer

Ergänzen Sie außerdem die Klasse `Kunde` um Getter- und Setter-Methoden für die beiden Attribute.



Bisher haben wir in Methoden lediglich Attributzugriffe, `return`-Anweisungen und Anweisungen, die wir aus der vorangegangenen Kurseinheit kennen, verwendet. Allerdings treten in einer Klasse oft an mehreren Stellen die gleichen Berechnungen auf. So haben wir sowohl in der Methode `berechneBruttopreis()` als auch in der Methode `gebeAus()` den Bruttopreis berechnet. Um solche unnötigen Verdoppelungen von Anweisungen zu vermeiden, können wir in einer Methode auch andere Methoden aufrufen und deren Ergebnis zu einer weiteren Verarbeitung verwenden [JLS: § 15.12]. Wir können somit die Methode `gebeAus()` folgendermaßen anpassen:

Methoden aufrufen

```

class Rechnung {
    // Attribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void gebeAus() {
        System.out.println("An:");
        System.out.println(this.rechnungsempfaenger.name);
        System.out.println(this.rechnungsempfaenger.anschrift);
        System.out.print("Netto: ");
        System.out.println(berechneNettopreis());
        System.out.print("MwSt: ");
        System.out.println(berechneMehrwertsteuer());
        System.out.print("Brutto: ");
        System.out.println(berechneBruttopreis());
    }

    // ...
}

```

Steht vor einem Methodenaufruf nicht explizit ein Objekt, an dem sie aufgerufen werden soll, so erfolgt der Aufruf genau an dem Objekt, an dem schon die umgebende Methode aufgerufen wurde. Wir können auch ein `this` vor die Methodenaufrufe setzen. So wird deutlich, dass der Brutto- bzw. Nettopreis genau dieser Rechnung berechnet wird. Somit können auch direkte Zugriffe auf Attribute außerhalb der Getter- und Setter-Methoden vermieden werden.

```

class Rechnung {
    // Attribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void gebeAus() {
        System.out.println("An:");
        System.out.println(this.rechnungsempfaenger.liefereName());
        System.out.println(this.rechnungsempfaenger.
            liefereAnschrift());
        System.out.print("Netto: ");
        System.out.println(this.berechneNettopreis());
        System.out.print("MwSt: ");
        System.out.println(this.liefereMehrwertsteuer());
        System.out.print("Brutto: ");
        System.out.println(this.berechneBruttopreis());
    }

    // ...
}

```

Selbsttestaufgabe 18.3-10:

Untersuchen Sie Ihre Implementierung der Klasse Rechnung auf doppelten Quelltext und versuchen Sie diesen, wenn sinnvoll, durch Methodenaufrufe zu ersetzen. ◇

Bisher waren die Typen der Parameter und Ergebnisse primitive Typen. Jedoch ist es auch möglich Referenzen auf Objekte als Parameter zu übergeben oder zurück zu liefern. Dies wird zum Beispiel nötig wenn wir Getter- und Setter-Methoden für das Attribut rechnungsempfaenger implementieren.

```
class Rechnung {
    // Attribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    Kunde liefereRechnungsempfaenger() {
        return this.rechnungsempfaenger;
    }

    void aendereRechnungsempfaenger(Kunde neuerEmpfaenger) {
        this.rechnungsempfaenger = neuerEmpfaenger;
    }

    // ...
}
```

Dieselben Gesetzmäßigkeiten, die wir in Abschnitt 17.3 im Bezug auf Zuweisungen diskutiert haben, gelten auch bei der Übergabe von Parametern an Methoden. Wird als Parameter ein Wert eines primitiven Datentyps übergeben, so hat die weitere Verarbeitung dieses Wertes durch die Methode keinerlei Auswirkungen auf die aufrufende Stelle. In diesem Fall spricht man von einem Aufruf mit Wertübergabe (engl. *call by value*). Wird einer Methode als Parameter ein Objekt übergeben, so handelt es sich, exakt gesprochen, um eine Referenz auf das Objekt. Sofern weitere Referenzen auf dieses Objekt existieren, was in der Regel zumindest an der aufrufenden Stelle der Fall ist, sind Veränderungen des Objekts an allen Referenzen gleichermaßen bemerkbar. Dies wird als Aufruf mit Verweisübergabe (engl. *call by reference*) bezeichnet.

Aufruf mit
Wertübergabe

Aufruf mit
Verweisübergabe

Bemerkung 18.3-6:

Wird dem Parameter eine andere Referenz zugewiesen, so hat dies keine Auswirkung auf den Aufrufer. ┘

Wann immer eine Klasse in einer Methode Objekte verarbeitet (dies gilt ebenso für Parameter wie für eigene Attribute der Klasse), sollten Sie sich bewusst

sein, dass weitere Referenzen auf diese Objekte existieren können, die von den Veränderungen mit betroffen sind. Würde eine Methode der Klasse Rechnung Manipulationen an dem durch ihr Attribut rechnungsempfaenger referenzierten Kunde-Objekt vornehmen, so sind diese Veränderungen auch außerhalb von Rechnung-Objekten an anderen Referenzen auf dasselbe Kunde-Objekt bemerkbar. Auch der umgekehrte Weg ist möglich. Es kann, falls außerhalb eines Rechnung-Objekts eine entsprechende Referenz existiert, das von seinem Attribut rechnungsempfaenger referenzierte Kunde-Objekt manipuliert werden.

Selbsttestaufgabe 18.3-11:

Implementieren Sie die im UML-Klassendiagramm in Abb. 18.3-1 dargestellten Klassen Artikel und Rechnungsposten. Passen Sie anschließend die Methode `fuegePostenHinzu()` der Klasse Rechnung so an, dass Sie als Parameter ein Objekt der Klasse Rechnungsposten erwartet und an Hand der im Rechnungsposten-Objekt gespeicherten Informationen den Betrag anpasst.

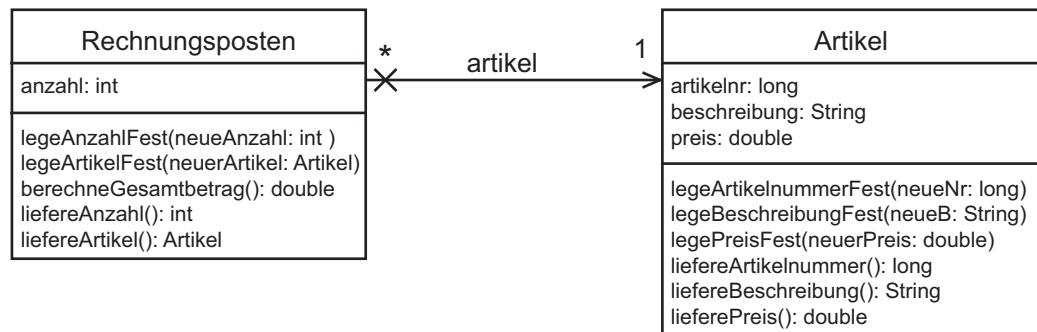


Abb. 18.3-1: UML-Diagramm mit Rechnungsposten und Artikel



18.4 Konstruktordeklaration

Konstruktor (engl. *constructor*) sind spezielle Methoden, um Exemplare einer Klasse zu erzeugen. Konstruktor sind immer nach derjenigen Klasse benannt, deren Exemplare sie erzeugen. In ihnen werden in der Regel die nötigen Initialisierungen, zum Beispiel die Belegung der Attribute mit Anfangswerten, durchgeführt. Es handelt sich bei einem Konstruktor um eine besondere Methode, die immer das eben erzeugte Objekt zurück liefert. Ein Konstruktor wird nie an einem Objekt oder einer Klasse aufgerufen sondern immer in Zusammenhang mit dem Schlüsselwort `new`.

Definition 18.4-1: Konstruktordeklaration

Eine Konstruktordeklaration besteht aus einem Kopf und einem Rumpf. Sie genügt folgender Form: Konstruktordeklaration

```
Konstruktorname(Parameterliste) {
    Anweisungen
}
```

Der Name des Konstruktors muss mit dem Namen der Klasse, der er angehört, identisch sein. Anders als bei einer Methodendeklaration hat eine Konstruktordeklaration keinen expliziten Ergebnistyp und der Rumpf muss keine return-Anweisung enthalten. [JLS: § 8.8] ┘

Innerhalb eines Konstruktors kann auf die Elemente einer Klasse genau wie aus einer Methode, zum Beispiel mit `this` zugegriffen werden, auch ansonsten können im Rumpf die gleichen Anweisungen wie in einer Methode genutzt werden.

Eine Konstruktordefinition mit einer leeren Parameterliste wird als Standardkonstruktor (engl. *default constructor*) bezeichnet. [JLS: § 8.8.9]

Standardkonstruktor

Wird in einer Klasse kein Konstruktor explizit deklariert, so besitzt diese Klasse automatisch einen Standardkonstruktor mit leerem Rumpf. In der Regel sollte jedoch eine Klasse eigene Konstruktoren deklarieren.

Wie Methoden haben auch Konstruktoren eine Signatur.

Definition 18.4-2: Signatur eines Konstruktors

Die Signatur eines Konstruktors besteht aus dem Konstruktornamen sowie den Typen seiner formalen Parameter. [JLS: § 8.8.2] ┘ Signatur eines Konstruktors

Konstruktornamen können ebenso wie Methodennamen überladen werden. [JLS: § 8.8.8]

überladene
Konstruktornamen

Finale Attribute (vgl. Abschnitt 18.2) müssen entweder direkt bei der Deklaration initialisiert werden, oder in allen Konstruktoren der Klasse. Anderenfalls entsteht ein Übersetzungsfehler. So können wir unsere Rechnung beispielsweise mit einer Rechnungsnummer versehen, die im Konstruktor gesetzt wird und sich später nicht mehr verändern darf. Wir übergeben die Rechnungsnummer als Parameter. Eine weitere Zuweisung im gleichen Konstruktor oder in einer Methode würde zu einem Übersetzungsfehler führen.

Initialisierung finaler
Attribute

```
class Rechnung {
    // Attribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;
    final int rechnungsnummer;
```

```

// Konstruktoren
Rechnung(final int rechnungsnummer) {
    this.rechnungsnummer = rechnungsnummer;
}

// ...
}

```

Oftmals ist es sinnvoll, mehr als einen Konstruktor anzubieten. Um aber doppelten Quelltext in den einzelnen Konstruktoren zu vermeiden, kann ein Konstruktor auch genau einen anderen aufrufen. Gehen wir also davon aus, dass unsere Rechnung in der Regel einen Mehrwertsteuersatz von 19 % besitzt. Wir bieten aber noch zusätzlich einen zweiten Konstruktor an, der den Mehrwertsteuersatz als Parameter erwartet. Ein Aufruf an einen anderen Konstruktor erfolgt mit Hilfe von `this(Parameterliste)` [JLS: § 8.8.7.1]. Ein solcher Aufruf muss, wenn vorhanden, immer die erste Anweisung in einem Konstruktor sein.

this-
Konstruktoraufruf

```

class Rechnung {
    // Attribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;
    final int rechnungsnummer;

    // Konstruktoren
    Rechnung(final int rechnungsnummer) {
        this(rechnungsnummer, 0.19);
    }

    Rechnung(final int rechnungsnummer, final double mwst) {
        this.rechnungsnummer = rechnungsnummer;
        this.mehrwertsteuer = mwst;
    }

    // ...
}

```

Bemerkung 18.4-1: Initialisierung der Exemplarvariablen

Die Exemplarvariablen werden vor der Ausführung des Konstruktors initialisiert. ┘

Selbsttestaufgabe 18.4-1:

Ergänzen Sie die Klassen Kunde, Rechnungsposten und Artikel um geeignete Konstruktoren.



Selbsttestaufgabe 18.4-2:

Entwickeln Sie zu der folgenden Beschreibung zunächst ein passendes UML-Klassendiagramm und implementieren Sie die Klassen anschließend in Java:

Ein Angestellter besitzt einen Namen und ein aktuelles Gehalt. Zudem gehört jeder Angestellter genau zu einer Abteilung. Eine Abteilung besitzt eine Nummer und einen Namen. Name, Gehalt und Abteilung des Angestellten können erfragt und das Gehalt um einen gegebenen Prozentsatz erhöht werden.

**Selbsttestaufgabe 18.4-3:**

Entwickeln Sie zu der folgenden Beschreibung zunächst ein passendes UML-Klassendiagramm und implementieren Sie die Klassen anschließend in Java:

Ein Auto hat einen Tank mit einer individuellen, maximalen Füllmenge. Ein Auto kann betankt werden. Zudem besitzt es einen durchschnittlichen Verbrauch (in Litern pro 100 Kilometer). Wenn das Auto eine Strecke gegebener Länge zurücklegt, wird entsprechend Benzin verbraucht.

Implementieren Sie anschließend eine `main()`-Methode in der Klasse `AutoFahrt`, in der ein neues Auto erzeugt wird und die verschiedenen Methoden ausprobiert werden.



19 Klassenvariablen und -methoden

Bisher haben wir nur Elemente der Klassendeklaration betrachtet, die es für jedes Objekt einer Klasse gibt oder die Objekte erzeugen. Jedoch können auch Klassen selbst bestimmte Eigenschaften besitzen oder von konkreten Objekten unabhängiges Verhalten anbieten. Klassenvariablen und Klassenmethoden sind Attribute und Methoden, die einer Klasse und nicht den Exemplaren einer Klasse angehören. Im Gegensatz zu Exemplarvariablen gibt es jede Klassenvariable genau einmal, unabhängig von der Anzahl der existierenden Exemplare.

19.1 Klassenvariablen

Bisher haben wir die Rechnungsnummer im Konstruktor übergeben. Es wäre einfacher wenn diese Nummer automatisch erzeugt und keine Nummer doppelt vergeben werden würde. Die dafür notwendigen Informationen können in der dazugehörigen Klasse gespeichert werden. So können die Rechnungsnummern automatisch aufsteigend vergeben werden. Klassenvariablen können von den Objekten einer Klasse gemeinsam genutzt werden. Es handelt sich bei Ihnen nicht um Eigenschaften eines konkreten Objekts, sondern um Eigenschaften der gesamten Klasse.

Definition 19.1-1: Klassenvariable

Klassenvariable
static

Eine Klassenvariable wird durch das Schlüsselwort `static`, das der Attributvereinbarung vorangestellt wird, nach folgendem Muster deklariert [JLS: § 8.3.1.1]:

```
static Typ Name;
```

┘

Bemerkung 19.1-1:

Klassenattribut

Klassenvariablen werden auch als Klassenattribute bezeichnet. Um keine Verwirrung entstehen zu lassen, können Exemplarvariablen dann auch als Exemplarattribute bezeichnet werden.

┘

Soll von außerhalb einer Klasse auf eine Klassenvariable zugegriffen werden, so muss der Klassenname mit einem Punkt getrennt vorangestellt werden.



Erfolgt ein Zugriff auf eine Klassenvariable von innerhalb der Klasse, stellt man den Klassennamen zur Verdeutlichung auch voran.



In der Klassendeklaration werden in der Regel erst die Klassenvariablen und anschließend die Exemplarvariablen deklariert.

Wir ergänzen die Klassenvariable `naechsteRechnungsnummer` in der Klasse `Rechnung` und initialisieren sie mit 10000. Im Konstruktor kann automatisch auf diese zugegriffen und das Exemplarattribut `rechnungsnummer` entsprechend initialisiert werden. Allerdings darf dabei nicht vergessen werden den in `naechsteRechnungsnummer` gespeicherten Wert anzupassen, so dass eine Nummer nicht mehrfach vergeben wird.

```
class Rechnung {
    // Klassenattribute
    static int naechsteRechnungsnummer = 10000;

    // Exemplarattribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;
    final int rechnungsnummer;

    // Konstruktoren
    Rechnung() {
        this(0.19);
    }

    Rechnung(double mwst) {
        this.mehrwertsteuer = mwst;
        this.rechnungsnummer
            = Rechnung.naechsteRechnungsnummer;
        Rechnung.naechsteRechnungsnummer++;
    }

    // ...
}
```

Häufig werden in Klassen auch Konstanten benötigt, die für alle Methoden zur Verfügung stehen sollen oder auch für andere Klassen. Solche Konstanten werden in der Regel als finale Klassenattribute implementiert. So könnten wir den Standardmehrwertsteuersatz als Konstante der Klasse `Rechnung` deklarieren. Auf diese Konstante kann dann von außerhalb mit `Rechnung.STANDARD_MEHRWERTSTEUERSATZ` zugegriffen werden.

Konstanten

```
class Rechnung {
    // Klassenattribute
    static final double STANDARD_MEHRWERTSTEUERSATZ = 0.19;
    static int naechsteRechnungsnummer = 10000;

    // Exemplarattribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
```

```

Kunde rechnungsempfaenger;
final int rechnungsnummer;


// Konstruktoren
Rechnung() {
    this(Rechnung.STANDARD_MEHRWERTSTEUERSATZ);
}

Rechnung(double mwst) {
    this.mehrwertsteuer = mwst;
    this.rechnungsnummer
        = Rechnung.naechsteRechnungsnummer;
    Rechnung.naechsteRechnungsnummer++;
}

// ...
}

```

Finale Klassenattribute müssen direkt bei der Deklaration initialisiert werden.

 Für die Bezeichner von finalen Klassenattributen gelten die Konventionen finaler lokaler Variablen.

Selbsttestaufgabe 19.1-1:

Ergänzen Sie die Klasse `Kreis` um eine Konstante `PI` mit dem Wert 3.14159265.



19.2 Klassenmethoden

Bei Exemplarattributen haben wir gelernt, dass auf diese nach Möglichkeit nur über ihre Getter- und Setter-Methoden zugegriffen werden sollte. Das gleiche gilt auch für Klassenattribute. Bei Ihnen erfolgt der Zugriff dann über Klassenmethoden. Eine solche Klassenmethode gehört zur Klasse und nicht zu einem konkreten Objekt. Sie kann somit auch nur auf Eigenschaften der Klasse, also Klassenattribute, und auf andere Klassenmethoden, nicht jedoch auf Exemplarmethoden zugreifen. In einer Klassenmethode ist somit auch keine `this`-Referenz verfügbar.

Definition 19.2-1: Klassenmethode

Klassenmethode

Klassenmethoden werden wie Methoden vereinbart, mit dem Unterschied, dass ihnen das Schlüsselwort `static` vorangestellt ist. [JLS: § 8.4.3.2]



Der Zugriff auf Klassenmethoden erfolgt sowohl von außerhalb, als auch von innerhalb wie bei Klassenvariablen.

Statt im Konstruktor die nächste Rechnungsnummer zu berechnen, können wir dafür eine eigene Methode implementieren.

```
class Rechnung {
    // Klassenattribute
    static final double STANDARD_MEHRWERTSTEUERSATZ = 0.19;
    static int naechsteRechnungsnummer = 10000;

    // Exemplarattribute
    double betrag;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;
    final int rechnungsnummer;

    // Konstruktoren
    Rechnung(double mwst) {
        this.mehrwertsteuer = mwst;
        this.rechnungsnummer
            = Rechnung.berechneNaechsteRechnungsnummer();
    }

    // Klassenmethoden
    static int berechneNaechsteRechnungsnummer() {
        return Rechnung.naechsteRechnungsnummer++;
    }

    // ...
}
```

Selbsttestaufgabe 19.2-1:

Warum entstehen bei der Übersetzung der nachfolgenden Klasse Fehler?

```
class A {
    static int x;
    int y;
    int z;

    static int f(int a, int y) {
        return a + 2 * y;
    }

    static int g(int y) {
        return 2 * this.y + A.x + z;
    }
}
```

```

        int h(int z) {
            return A.x + y + z;
        }
    }

```



Mit unserem Wissen über Exemplar- und Klassenmethoden sowie über Exemplar- und Klassenvariablen können wir die Anweisung:

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

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

die wir so häufig benutzen, in ihre Namensbestandteile zerlegen.

`System` bezeichnet eine Klasse, die von der Laufzeitumgebung (JDK) bereit gestellt wird; `out` ist eine Klassenvariable von `System`. `System.out` bezeichnet die Klassenvariable `out` der Klasse `System`. Die Klassenvariable `out` ist ihrerseits ein Objekt (einer Klasse namens `PrintStream`), das eine Methode `println()` hat. Somit steht `System.out.println()` für den Aufruf der Exemplarmethode `println` an der Klassenvariable `out` der Klasse `System`.

Selbsttestaufgabe 19.2-2:

Exemplarmethoden können mit Hilfe der `this`-Referenz auf den Attributen des jeweiligen Objekts operieren, sie können aber auch auf die Klassenvariablen zugreifen.

Erklären Sie, ob und, wenn ja, wie eine Klassenmethode auf Exemplarattribute zugreifen kann.



20 Felder

Viele Anwendungsprogramme erfordern die Speicherung und Pflege einer einheitlichen Sammlung von Datenelementen. Dies ist zum Beispiel der Fall, wenn wir alle Artikel einer Rechnung speichern wollen und nicht nur den Gesamtbetrag. Für jeden Artikel eine einzelne Variable anzulegen ist keine Möglichkeit, da jede Rechnung eine verschiedene Anzahl an Artikel speichern kann.

20.1 Felder einführen und belegen

Der einfachste und gebräuchlichste Typ für Sammlungen von Daten sind Feldtypen. Ein Feld (engl. *array*) gruppiert mehrere Elemente des gleichen Typs zu einer einzelnen Einheit vorgegebener Größe. Die Größe eines Feldes bestimmt die Anzahl der Elemente, die es maximal aufnehmen kann. Ein Feld ist als indizierte Folge organisiert, so dass auf jedes Element mit einem eindeutigen Index zugegriffen werden kann. Der Typ der Elemente kann dabei ein primitiver Datentyp oder ein Objekttyp sein.

Feldtypen
Feld

Definition 20.1-1: Vereinbarung von Feldern

Eine Variable für ein Feld wird wie folgt vereinbart:

```
ElementTyp[] Feldname;
```

Vereinbarung von
Feldern

Der Feldtyp ist durch den Datentyp der Elemente (ElementTyp), die in einem Feld enthalten sind, bestimmt.

┘

Beispiel 20.1-1: Deklaration einer Feldvariable

Die Deklaration

```
int[] artikelnummern;
```

bezeichnet ein Feld mit Elementen vom Typ int, das artikelnummern genannt wird.

┘

Bemerkung 20.1-1:

Feldtypen sind Objekttypen. Jedes Feld ist somit auch ein Objekt. [JLS: § 4.3.1]

┘

Im Gegensatz zu Variablen von primitiven Typen wird einer Feldvariable bei ihrer Deklaration – wie bei Variablen von Objekttypen auch – noch kein Feld und auch kein Speicherplatz zugewiesen [JLS: § 10.2]. Eine Feldvariable ist ein Verweis auf ein Feldobjekt, für das erst dann Speicherplatz angelegt wird, wenn das Feld erzeugt wird.

Definition 20.1-2: Erzeugung eines Feldes

Erzeugung eines Feldes

Einer Feldvariablen wird ähnlich wie bei Objekten mittels des Schlüsselworts `new` ein Feldobjekt fester Größe zugewiesen:

```
Feldname = new ElementType[IntAusdruck];
```

Mit dem Ausdruck auf der rechten Seite der Zuweisung wird ein konkretes Feld mit Elementen des Typs `ElementType` der Größe `IntAusdruck` angelegt. Die Feldgröße wird durch einen Ganzzahlausdruck bestimmt. Der Ausdruck muss zu einem nichtnegativen Wert evaluiert werden. Die Referenz auf das erzeugte Feld wird der Feldvariablen `Feldname`, die vom gleichen Typ sein muss, zugewiesen. Die Feldelemente sind von 0 bis `IntAusdruck - 1` indiziert. [JLS: § 10.3, § 15.10]

┘

Definition 20.1-3: Vereinbarung und Erzeugung eines Feldes

Es ist auch möglich, die Variablenvereinbarung und Felderzeugung zu kombinieren [JLS: § 10.2]:

```
ElementType[] Feldname = new ElementType[IntAusdruck];
```

Erzeugung durch
Aufzählung

Statt das Schlüsselwort `new` zu verwenden, kann ein Feld auch durch eine Aufzählung gebildet und zugleich initialisiert werden [JLS: § 10.6]:

```
ElementType[] Feldname = {v1, v2, ..., vn};
```

Die Ausdrücke `v1, v2, ..., vn` müssen Werte des Typs `ElementType` liefern.

┘

Einmal erstellt, ist die Größe eines Feldes festgelegt und kann nicht mehr verändert werden. Einer Feldvariablen dagegen können innerhalb ihrer Lebensdauer verschiedene Felder zugewiesen werden.

Beispiel 20.1-2: Erzeugung eines Feldes

Der Ausdruck:

```
int[] artikelnummern = new int[10];
```

erzeugt ein Feld mit 10 Elementen vom Typ `int` und weist es der Feldvariablen `artikelnummern` zu.

┘

Beispiel 20.1-3: Erzeugung und Initialisierung eines Feldes

Das Feld `deutscheVokale` gibt ein Beispiel für die Aufzählungsdefinition eines Feldes:

```
char[] deutscheVokale
    = {'a', 'ä', 'o', 'ö', 'u', 'ü', 'e', 'i'};
```

┘

Bemerkung 20.1-2:

Eine solche Aufzählung kann in dieser Form nur bei der Deklaration verwendet werden. Soll sie auch an anderer Stelle genutzt werden, so müssen der Aufzählung der Feldelemente das Schlüsselwort `new` und der Feldtyp vorangestellt werden.

```
char[] deutscheVokale;
// ...
deutscheVokale
    = new char[]{'a', 'ä', 'o', 'ö', 'u', 'ü', 'e', 'i'};
```

Definition 20.1-4: Länge eines Feldes

*Jedes Feld besitzt eine finale Exemplarvariable `length`. Die Größe eines Feldes Länge eines Feldes
Feldname kann somit im Programm durch den ganzzahligen Ausdruck:*

```
Feldname.length
```

ermittelt werden. [JLS: § 10.7]

Beispiel 20.1-4: Länge eines eindimensionalen Feldes

So liefert beispielsweise der Ausdruck

```
deutscheVokale.length;
```

den Wert 8, wenn die Variable `deutscheVokale` wie oben dargestellt initialisiert wurde.

Mit der Erzeugung eines Feldes wurde entsprechender Speicherplatz reserviert. Nun können wir den Elementen des Feldes Werte zuweisen.

Bemerkung 20.1-3: Standardbelegung der Feldelemente

Falls ein Feld aus Elementen primitiver Datentypen aufgebaut ist, wird jedes Feldelement mit dem Standardwert dieses Typs belegt. Für ein Ganzzahlfeld ist der Standardwert 0, für Gleitkommazahlen der Wert 0.0, für Zeichenelemente das Zeichen mit dem Ganzzahlcode 0, und für `boolean` ist es der Wert `false`. Handelt es sich bei dem Typ der Feldelemente um einen Objekttyp, so sind alle Elemente mit `null` initialisiert. [JLS: § 15.10.1, § 4.12.5]

Standardbelegung der
Feldelemente

Definition 20.1-5: Feldzugriff

Die Werte der Elemente eines Feldes können durch den vordefinierten Indexausdruck:

Feldzugriff

```
Feldname[IntAusdruck]
```

abgefragt oder verändert werden. Der Indexausdruck muss ein ganzzahliger Wert sein. [JLS: § 10.4]

Bemerkung 20.1-4: Feldindex

Feldindex *Feldindizes beginnen immer mit 0. Daher dürfen Indexausdrücke in Feldzugriffen weder negative ganze Zahlen sein, noch dürfen sie den größten Indexwert `Feldname.length - 1` überschreiten.*

Falls der `int`-Ausdruck, der den Index bestimmt, außerhalb der Feldgrenzen 0 bis `Feldname.length - 1` liegt, wird dieser Zugriff einen Laufzeitfehler erzeugen. [JLS: § 10.4]

┘

Selbsttestaufgabe 20.1-1:

Welcher Laufzeitfehler tritt auf, wenn Sie auf `deutscheVokale[-1]` und auf `deutscheVokale[100]` zugreifen wollen?

◇

Beispiel 20.1-5: Feldzugriff

Die obige Vereinbarung der Feldvariable `deutscheVokale` vorausgesetzt, ist der Ausdruck

```
deutscheVokale[3]
```

ein gültiger Zugriff und liefert das Ergebnis ö.

┘

Bemerkung 20.1-5: Zugriff auf nicht initialisierte Feldvariable

Bei einem Zugriff auf die Länge oder ein Element einer nicht initialisierten Feldvariablen wird zur Laufzeit ein Fehler auftreten.

┘

Bemerkung 20.1-6: Belegung eines Feldes

Die Elemente eines Feldes können durch eine `for`-Schleife einheitlich mit Anfangswerten belegt werden:

```
for (int i = 0; i < Feldname.length; i++) {
    Feldname[i] = Wert;
}
```

wobei `Wert` verträglich mit dem Typ der Feldelemente sein muss.

┘

Bemerkung 20.1-7: finale Feldvariablen

Beachten Sie, dass eine als `final` deklarierte Feldvariable lediglich keine neue Zuweisung an diese Variable zulässt. Eine Veränderung der Feldinhalte ist jedoch trotzdem möglich.

┘

Selbsttestaufgabe 20.1-2:

Gegeben sei die Anweisung:

```
int[] myArray = {12, 34, 7, 8};
```

Welche der folgenden Aussagen sind richtig?

- Das erste Element in `myArray` ist 12
- `myArray` kann vier `int`-Werte beinhalten
- Der obige Code ist ungültig
- `myArray[4] == 8;`

**Selbsttestaufgabe 20.1-3:**

Erstellen Sie ein Feld mit 8 Elementen des Typs `int` und dem Namen `eightInts`, und weisen Sie jedem Element den Wert seines Indexwerts zu.

**Selbsttestaufgabe 20.1-4:**

Da Artikel auch verschiedene Mehrwertsteuersätze aufweisen können, sollte die Klasse `Artikel` um ein entsprechendes Attribut und zugehörige Getter- und Setter-Methoden ergänzt werden. Ergänzen Sie die Klasse `Rechnung` um ein Feld, in dem alle Rechnungsposten gespeichert werden können. Die Methode `fuegePostenHinzu()`, soll nun nicht mehr den Betrag anpassen, sondern den Rechnungsposten in dem Feld speichern. Sie können davon ausgehen, dass eine Rechnung nie mehr als 100 Posten besitzt. Eine solche Information sollte in der Regel in Form von Konstanten dokumentiert werden. Die Attribute `betrag` und `mehrwertsteuer` werden dann nicht mehr benötigt, da die Werte jederzeit aus den einzelnen Posten berechnet werden können. Die zugehörigen Getter- und Setter-Methoden entfallen dann. Passen Sie deshalb, wenn nötig, die Konstruktoren sowie die Methoden `berechneNettopreis()`, `berechneMehrwertsteuer()` und `berechneBruttoPreis()` an. Die Methode `gebeAus()` sollte nun auch alle Rechnungsposten mit ausgeben.



20.2 Mehrdimensionale Felder

Da es Felder von beliebigen Objekttypen geben kann und Feldtypen Objekttypen sind, können wir auch Felder von Feldern bilden. Ein Feld von Feldern nennen wir ein mehrdimensionales Feld. Ein Feld, dessen Elemente nicht Felder sind, nennen wir ein eindimensionales Feld. Ein zweidimensionales Feld ist ein Feld, das als Elemente eindimensionale Felder besitzt. Es können Felder mit beliebig vielen Dimensionen deklariert werden.

Mehrdimensionales Feld

Mehrdimensionale Felder werden ähnlich wie eindimensionale Felder vereinbart und mit Werten belegt. Der Unterschied besteht darin, dass statt einer mehrere Dimensionen angegeben werden müssen.

Beispiel 20.2-1: Ein zweidimensionales Feld

Ein Feld, dessen Elemente vom Typ `int[]` sind, hat den Feldtyp `int[][]`. Mit der Anweisung

```
int[][] a = new int[3][];
```

erzeugen wir ein Feld für drei Elemente vom Typ `int[]`. Diese Elemente können mit den bekannten Indexausdrücken `a[0]`, `a[1]` und `a[2]` angesprochen werden (vgl. Definition 20.1-5). Da es sich bei den Elementen um Felder, also um Objekte handelt, sind sie mit `null` initialisiert (vgl. Bemerkung 20.1-3). Mit der Anweisung

```
a[0] = new int[5];
```

erzeugen wir ein Feld vom Typ `int[]` mit fünf Elementen vom Typ `int` und weisen die Referenz auf dieses Feld dem ersten Element des Feldes `a` zu. Die Elemente des Feldes `a[0]` sprechen wir mit den Indexausdrücken `a[0][0]`, `a[0][1]`, ..., `a[0][4]` an. Sie sind mit dem Wert 0 initialisiert.

Die Längen der Felder `a[0]`, `a[1]` und `a[2]` sind voneinander unabhängig. So können wir den beiden übrigen Elementen von `a` beispielsweise folgende zwei Felder zuweisen:

```
a[1] = new int[2];
a[2] = new int[20];
```

┘

Tabelle Oft werden in Anwendungen Tabellen benötigt. Die Daten in Tabellen sind in Reihen und Spalten organisiert. Wir sind es aus der Mathematik gewohnt, das Element in der i -ten Reihe und j -ten Spalte einer Tabelle T durch den Ausdruck $T_{i,j}$ zu bezeichnen. Tabellen können in Java durch zweidimensionale Felder nachgebildet werden, deren eindimensionale innere Felder eine einheitliche Länge erhalten. Die Reihen werden in der Vereinbarung und in Zugriffen auf Tabellenelemente vor den Spalten genannt.

Sollen alle Elemente eines zweidimensionalen Feldes dieselbe Länge erhalten, so können das äußere und alle inneren Felder gleichzeitig erzeugt werden, indem auch die zweite eckige Klammer mit einer Längenangabe versehen wird.

Beispiel 20.2-2: Schachbrett

Ein Schachbrett ist ein typisches Beispiel für eine Tabelle oder ein zweidimensionales Feld aus 8 mal 8 Elementen. Es wird wie folgt deklariert und erzeugt:

```
final int SIZE = 8;
int[][] chessboard = new int[SIZE][SIZE];
```

Das Element in der dritten Reihe und vierten Spalte unseres Schachbretts wird mit dem Ausdruck

```
chessboard[2][3]
```

bezeichnet.

Analog können Felder beliebiger Dimension, die in jeder Dimension einheitliche Längen aufweisen, erzeugt werden.

Um ein zwei- oder mehrdimensionales Feld vollständig mit eigenen Werten zu belegen, werden für gewöhnlich so viele verschachtelte `for`-Schleifen, wie das Feld Dimensionen hat, verwendet.

Beispiel 20.2-3: Elemente des Schachbretts

Das nachstehende Programm füllt alle Elemente des Feldes `chessboard` mit der Summe ihrer Zeilen- und Spaltenindizes.

```
final int SIZE = 8;
int[][] chessboard = new int[SIZE][SIZE];
for (int row = 0; row < chessboard.length; row++) {
    for (int col = 0; col < chessboard[row].length; col++) {
        chessboard[row][col] = row + col;
    }
}
```

Bemerkung 20.2-1:

Im Vergleichsausdruck der `for`-Schleifen verwenden wir die jeweiligen Längen des äußeren und der einzelnen inneren Felder als Vergleichswerte. Solche Schleifen können bei jeder Größe und Form eines Feldes eingesetzt werden.

Mehrdimensionale Felder können wie eindimensionale Felder durch Aufzählung initialisiert werden (vgl. Definition 20.1-3).

Beispiel 20.2-4: Initialisierung durch Aufzählung

Dasselbe Schachbrett kann durch die folgende Aufzählung gebildet werden:

```
int[][] chessboard = {{0, 1, 2, 3, 4, 5, 6, 7},
                      {1, 2, 3, 4, 5, 6, 7, 8},
                      {2, 3, 4, 5, 6, 7, 8, 9},
                      {3, 4, 5, 6, 7, 8, 9, 10},
                      {4, 5, 6, 7, 8, 9, 10, 11},
                      {5, 6, 7, 8, 9, 10, 11, 12},
                      {6, 7, 8, 9, 10, 11, 12, 13},
                      {7, 8, 9, 10, 11, 12, 13, 14}};
```

Mit Hilfe der Aufzählung können mehrdimensionale Felder jeder Form und Zusammensetzung gebildet werden.

Selbsttestaufgabe 20.2-1:

Betrachten Sie das folgende durch Aufzählung gebildete dreieckige Feld:

```
int[][] dreieck = {{0},
                  {1, 2},
                  {2, 3, 4},
                  {3, 4, 5, 6},
                  {4, 5, 6, 7, 8},
                  {5, 6, 7, 8, 9, 10},
                  {6, 7, 8, 9, 10, 11, 12},
                  {7, 8, 9, 10, 11, 12, 13, 14}};
```

Erstellen Sie Java-Anweisungen, die das gleiche dreieckige Feld erzeugen und mit den gleichen Werten belegen, ohne dabei eine direkte Aufzählung zu benutzen.

Lösungshinweis: *Bedenken Sie, dass die inneren Felder keine einheitlichen Längen erhalten sollen und somit individuell erzeugt werden müssen.* ◇

Selbsttestaufgabe 20.2-2:

Entwickeln Sie Anweisungen, durch welche ein zweidimensionales, quadratisches Feld der Größe 6 erstellt und mit der Einheitsmatrix aufgefüllt wird. Die Einheitsmatrix ist eine quadratische Matrix vom Typ `double`, die den Wert 1 in allen Positionen der Hauptdiagonale und den Wert 0 in allen anderen Positionen trägt. ◇

Selbsttestaufgabe 20.2-3:

Betrachten Sie die folgende Deklaration eines dreidimensionalen Feldes.

```
int[][][] my3dArray;
my3dArray = new int[3][5][4];
```

1. *Wie viele Schleifen würde man benötigen, um das Feld mit Werten zu füllen?*
2. *Wie viele Elemente enthält das Feld?*
3. *Entwickeln Sie Anweisungen, die die Summe aller Feldelemente berechnen.* ◇

Selbsttestaufgabe 20.2-4:

Entwickeln Sie eine Klasse `Matrix`, die unter anderem Methoden anbietet, um zwei Matrizen zu addieren und um eine Matrix zu transponieren. ◇

Selbsttestaufgabe 20.2-5:

Entwickeln Sie eine Methode zur Bestimmung des Maximums eines eindimensionalen `int`-Feldes sowie eine Methode zur Bestimmung des Minimums eines dreidimensionalen `int`-Feldes. Sie können bei dem dreidimensionalen Feld annehmen, dass der Eintrag mit dem Index `[0][0][0]` existiert. Existieren in dem eindimensionalen Feld keine Einträge, so soll 0 zurückgeliefert werden. ◇

20.3 Erweiterte for-Schleife

Bisher haben wir für den Zugriff auf Feldelemente in der Regel eine einfache `for`-Schleife verwendet. Alternativ kann die erweiterte `for`-Schleife verwendet werden.

Sei `meinFeld` ein Feld mit Elementen vom Typ `int`, so konnten wir so bisher auf alle Elemente zugreifen:

```
int[] meinFeld = ...;
for (int i = 0; i < meinFeld.length; i++) {
    int x = meinFeld[i];
    // benutze x
}
```

Mit Hilfe der erweiterten `for`-Schleife können wir die Anweisung folgendermaßen verkürzen:

```
int[] meinFeld = ...;
for (int x : meinFeld) {
    // benutze x
}
```

Die Variable `x` nimmt im `i`-ten Schleifendurchlauf jeweils den Wert `meinFeld[i]` an, wobei `i` die Werte 0 bis `meinFeld.length - 1` annimmt.

Definition 20.3-1: Erweiterte for-Schleife

Die erweiterte `for`-Schleife besitzt die folgende Syntax:

```
for (ElementType VarName : Feldname) {
    // ...
}
```

wobei der Typ `ElementType` mit dem Typ der Elemente des Feldes `Feldname` verträglich sein muss. [JLS: § 14.14.2] ┘

Diese Form der `for`-Schleife kann auch bei mehrdimensionalen Feldern verwendet werden.

```
int[][] meinFeld = new int[4][3];
for (int[] x : meinFeld) {
    for (int y : x) {
        // y ist das aktuelle Feldelement
    }
}
```

Bemerkung 20.3-1:

Im Gegensatz zu der einfachen `for`-Schleife gibt es bei dieser Form keine Variable, die den aktuellen Index im Feld repräsentiert. Somit ist es ohne weitere Hilfsvariable auch nicht möglich zwei Felder parallel zu durchlaufen.

Zudem ermöglicht die erweiterte `for`-Schleife nur einen lesenden Zugriff auf die Feldelemente und somit keine Veränderung der Feldinhalte.

┘

Selbsttestaufgabe 20.3-1:

Versuchen Sie, die Maximumssuche aus Selbsttestaufgabe 20.2-5 mit Hilfe der erweiterten `for`-Schleife zu lösen.

◇

21 Zusammenfassung

Zusätzlich zu primitiven Datentypen gibt es in Java noch **Objekttypen**. Wir haben gelernt durch **Klassendeklarationen** neue Objekttypen einzuführen.

Mit Hilfe der Punktnotation kann auf die Attribute und Methoden von Exemplaren oder Klassen zugegriffen werden. Bei Methodenaufrufen müssen zu den **Parametern** passende **Argumente** übergeben werden. Die **Ergebnisse von Methodenaufrufen** können in Ausdrücken weiterverwendet werden.

Neue Objekte werden durch die Verbindung des Schlüsselworts `new` und eines **Konstruktoraufruf** erzeugt.

Der **Zustand eines Objekt** wird durch die **Attributwerte** gebildet.

Variablen eines primitiven Typs werden **Wertvariablen** genannt, Variablen eines Objekttyps werden **Verweisvariablen** genannt. Wenn eine Verweisvariable auf kein Objekt verweist, so hat sie den Wert `null`. Für den Vergleich zweier Referenzen stehen die Operatoren `==` und `!=` zur Verfügung.

In einer Klassenvereinbarung werden **Attribute**, **Methoden** und **Konstruktoren** deklariert. Attribute und Methoden können durch das Schlüsselwort `static` als **Klassenattributen** bzw. **-methoden** gekennzeichnet werden. Diese existieren nur für die Klasse und sind unabhängig von Exemplaren der Klasse. Sie können auch nicht auf **Exemplarvariablen** und **-methoden** zugreifen. Assoziationen werden in der Regel in Java durch Attribute implementiert.

Methoden und Konstruktoren müssen eine eindeutige **Signatur** besitzen, jedoch können Methoden- und Konstruktornamen **überladen** sein.

Variablen, die innerhalb von Methoden deklariert werden, werden **lokale Variablen** genannt und sind nur innerhalb der Methode gültig. Parameter sind auch lokale Variablen. Lokale Variablen können Attribute **verdecken**. Bei lokalen Variablen gibt es im Gegensatz zu nicht finalen Attributen keine Standardbelegung. **Finale Attribute** müssen entweder bei der Deklaration oder in allen Konstruktoren initialisiert werden. In Methoden darf niemals schreibend auf sie zugegriffen werden.

Innerhalb von Konstruktoren und Exemplarmethoden, kann immer mit Hilfe des Schlüsselworts `this` immer auf die Referenz des aktuellen Objekts zugegriffen werden. In Konstruktoren kann mit Hilfe von `this` auch ein anderer Konstruktor der Klasse aufgerufen werden um doppelten Quelltext zu vermeiden. Dieser Aufruf muss, wenn vorhanden, immer die erste Anweisung im Konstruktor sein.

Methoden können andere Methoden aufrufen. Mit Hilfe einer `return`- oder **Rückgabeanweisung** kann eine Methode beendet und das Ergebnis an den Aufrufer zurückgegeben werden. Liefert eine Methode kein Ergebnis, so ist sie als `void` deklariert.

Getter- und Setter-Methoden dienen dazu um direkte Zugriffe auf Attribute zu verhindern und somit das **Geheimnisprinzip** zu wahren.

Handelt es sich bei einem Parameter um eine Wertvariable so spricht man von **Aufruf mit Wertübergabe** und bei Verweisvariablen von **Aufruf mit Verweisübergabe**.

Die Klasse `String` dient zur Darstellung von Zeichenketten.

Ein **Feld** ist eine lineare Sammlung von Datenelementen oder Feld Objekten des gleichen Typs. **Feldtypen** sind Objekttypen.

Felder können mit Hilfe von `new` oder mit Hilfe von **Aufzählungen** erzeugt werden. Die Größe eines Feldes ist nicht veränderlich. Die Größe eines Feldes kann mit Hilfe des Attributs `length` ermittelt werden. Der Zugriff auf das `i`-te Element des Feldes `meinFeld` erfolgt mit `meinFeld[i]`. Dabei sind alle Werte von 0 bis `meinFeld.length - 1` gültige Indizes.

Mit Hilfe der erweiterten `for`-Schleife können Felder durchlaufen werden. Jedoch kann dabei nur lesend zugegriffen werden.

Lösungen zu Selbsttestaufgaben der Kurseinheit

Lösung zu Selbsttestaufgabe 17.1-1:

Die Anweisung in Zeile 1 ist korrekt. Dem Attribut `betrag` vom Typ `double` wird das `double`-Ergebnis einer Addition zugewiesen.

Die Anweisung in Zeile 2 ist korrekt. Dem Attribut `betrag` wird ein `int`-Wert zugewiesen, was nach der Regeln der Typverträglichkeit unproblematisch ist.

Die Anweisung in Zeile 3 ist nicht korrekt. Die Methode `fuegePostenHinzu()` verlangt zwei Parameter, zunächst einen `int`-Wert für die Anzahl, dann einen `double`-Wert für den Einzelpreis.

Die Anweisung in Zeile 4 ist korrekt. Der Methode `legeRabattFest()`, die einen `double`-Wert als Parameter verlangt, wird ein `int`-Wert übergeben, was nach der Regeln der Typverträglichkeit unproblematisch ist.

Die Anweisung in Zeile 5 ist nicht korrekt. Die Methode `berechneNettopreis()` muss ohne Parameter aufgerufen werden. Zudem wäre auch die syntaktisch korrekte Anweisung

```
rechnung1.berechneNettopreis();
```

unnütz, solange das Ergebnis der Methodenausführung nicht weiter verwendet, z. B. einer Variablen zugewiesen wird.

Lösung zu Selbsttestaufgabe 17.2-1:

Ihr Programmgerüst könnte beispielsweise wie folgt aussehen:

```
class RechnungTest {

    public static void main (String[] args) {

        Rechnung meineRechnung = new Rechnung();
        meineRechnung.fuegePostenHinzu(4, 1.20);
        meineRechnung.fuegePostenHinzu(13, 2.10);
        meineRechnung.fuegePostenHinzu(1, 7.80);
        meineRechnung.legeRabattFest(0.1);
        System.out.println(meineRechnung.berechneNettopreis());
    }
}
```

Lösung zu Selbsttestaufgabe 17.3-1:

Die Situation nach Ausführung der ersten sechs Anweisungen ist in Abb. ML 13 dargestellt.

Nach der Ausführung aller Anweisungen existieren nur noch zwei Rechnung-Objekte, deren beide `betrag`-Attribute den Wert 36.0 haben. Ein Rechnung-Objekt ist von `re1` referenziert, das andere von `re2` und `re3` (Abb. ML 14).

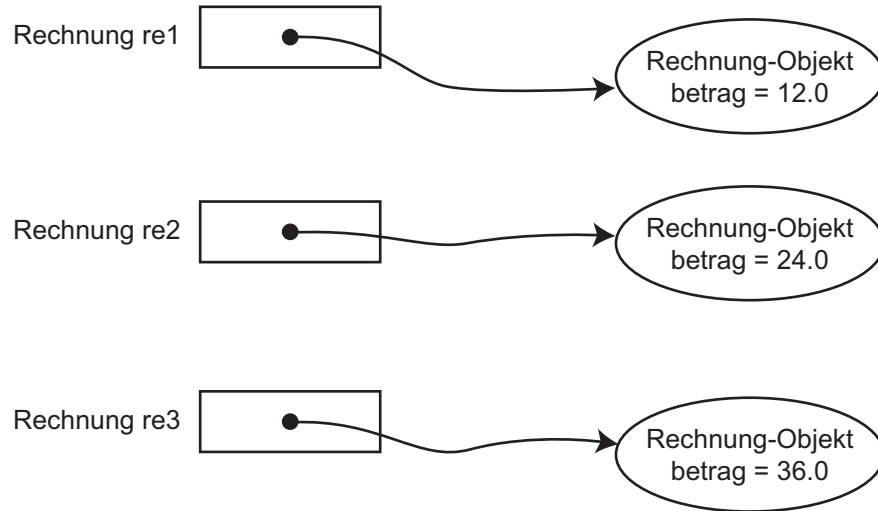


Abb. ML 13: Drei Variablen und drei Objekte

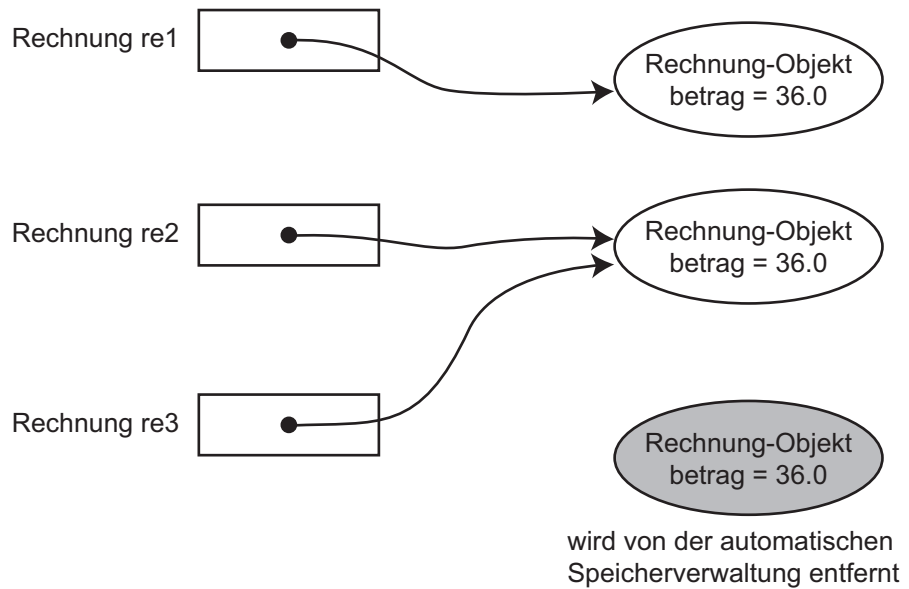


Abb. ML 14: Drei Variablen und zwei Objekte

Lösung zu Selbsttestaufgabe 17.4-1:

```
Kunde kunde2 = new Kunde("Anton Meier",
    "Hauptstr. 100, 12345 Irgendwo");
Kunde kunde3 = new Kunde("Peter Müller",
    "Rosenweg 34, 12347 Irgendwo");
kunde2.anschrift = kunde3.anschrift;
```

Alternativ kann in der dritten Zeile auch die Methode `liefereAnschrift()` aufgerufen werden:

```
kunde2.anschrift = kunde3.liefereAnschrift();
```

Lösung zu Selbsttestaufgabe 17.4-2:

Ein Rechnung-Objekt hat nur ein Attribut vom Typ Kunde. Ihm können folglich nicht gleichzeitig zwei Kunde-Objekte zugewiesen sein.

Lösung zu Selbsttestaufgabe 17.4-3:

Auf ein Kunde-Objekt kann es beliebig viele Referenzen geben. Es ist ohne weiteres möglich, zwei Referenzen auf dasselbe Kunde-Objekt an verschiedene Rechnung-Objekte zu übergeben:

```
Kunde kunde = new Kunde("Anton Meier",
    "Hauptstr. 100, 12345 Irgendwo");
Rechnung rechnungA = new Rechnung();
rechnungA.rechnungsempfaenger = kunde;
Rechnung rechnungB = new Rechnung();
rechnungB.rechnungsempfaenger = kunde;
```

Lösung zu Selbsttestaufgabe 17.4-4:

```
Rechnung meineRechnung = new Rechnung();
meineRechnung.fuegePostenHinzu(4, 1.20);
meineRechnung.fuegePostenHinzu(13, 2.10);
meineRechnung.fuegePostenHinzu(1, 7.80);
meineRechnung.legeRabattFest(0.1);
meineRechnung.gebeAus();
```

Was bei der Ausführung von `gebeAus()` passiert, können Sie nur ermitteln, indem Sie Ihre Anweisungsfolge ausführen. Es hängt von der Implementierung der Klasse `Rechnung` ab, ob beispielsweise eine Ausgabezeile wie "kein Empfänger angegeben" erscheint, oder ob eine `NullPointerException` entsteht.

Lösung zu Selbsttestaufgabe 18.2-1:

```
class Kreis {
    double radius;
}

class Rechteck {
    double laenge;
    double breite;
}
```

Lösung zu Selbsttestaufgabe 18.2-2:

```
class Kunde {
    String name;
    String anschrift;
}
```

Lösung zu Selbsttestaufgabe 18.2-3:

Die Klasse Kunde kann aus der Lösung zu Selbsttestaufgabe 18.2-2 übernommen werden.

```
class Rechnung {

    // Attribute
    double betrag = 0;
    double mehrwertsteuer;
    double rabatt;

    // Assoziation
    Kunde rechnungsempfaenger;
}
```

Lösung zu Selbsttestaufgabe 18.3-1:

```
class Rechnung {
    // Attribute
    double betrag = 0;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void legeRabattFest(double neuerRabatt) {
        // ...
    }
}
```

```

void legeMehrwertsteuerFest(double neueMwSt) {
    // ...
}

void fuegePostenHinzu(int anzahl, double einzelpreis) {
    // ...
}
}

```

Lösung zu Selbsttestaufgabe 18.3-2:

Die Signaturen der Klasse `Kreis` lauten:

- `berechneFlaecheninhalt()`
- `berechneUmfang()`

Lösung zu Selbsttestaufgabe 18.3-3:

Die Methoden `void a(int x)` und `int a(int u)` besitzen die gleiche Signatur, da sie sich nur im Rückgabetypp, der nicht zur Signatur gehört, und in dem Namen des Parameters unterscheiden. Alle anderen Methoden haben eine eindeutige Signatur, da sie sich in Typen, Reihenfolge oder Anzahl der Parameter unterscheiden.

Lösung zu Selbsttestaufgabe 18.3-4:

Die Methodennamen `a`, `b` und `c` sind überladen.

Lösung zu Selbsttestaufgabe 18.3-5:

```

class Rechnung {
    // Attribute
    double betrag = 0;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void legeRabattFest(final double neuerRabatt) {
        this.rabatt = neuerRabatt;
    }

    void legeMehrwertsteuerFest(final double neueMwSt) {
        this.mehrwertsteuer = neueMwSt;
    }
}

```

```
        void fuegePostenHinzu(final int anzahl,
                               final double einzelpreis) {
            this.betrag += anzahl * einzelpreis;
        }

        // ...
    }
```

Lösung zu Selbsttestaufgabe 18.3-6:

Es gibt vier Attribute:

- double a
- int b
- String c
- int d

Es gibt die folgenden lokalen Variablen (inklusive der Parameter):

- int y (Parameter)
- double d (Parameter)
- int x
- int z
- double m
- int a (Parameter)
- int b (Parameter)
- double c
- double f
- double g

Lösung zu Selbsttestaufgabe 18.3-7:

In der Anweisung `System.out.println(this.c)` wird auf das Attribut `c` vom Typ `String` zugegriffen. Im Ausdruck `a * this.b + c * b` wird von links nach rechts auf den Parameter `int a`, das Attribut `int b`, die lokale Variable `double c` und den Parameter `int b` zugegriffen. Im Ausdruck `this.a + d` wird auf das Attribut `double a` und das Attribut `int d` zugegriffen. Der Zugriff auf das Attribut `d` sollte jedoch zur Verdeutlichung mit einem `this` versehen werden.

Lösung zu Selbsttestaufgabe 18.3-8:

```

class Rechnung {
    // Attribute
    double betrag = 0;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void legeRabattFest(final double neuerRabatt) {
        // neuer Rabatt ist nur von 0 % bis 50 % gueltig
        if (neuerRabatt >= 0 && neuerRabatt <= 0.5) {
            this.rabatt = neuerRabatt;
        }
    }

    // ...
}

```

Lösung zu Selbsttestaufgabe 18.3-9:

```

class Rechnung {
    // Attribute
    double betrag = 0;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void legeRabattFest(final double neuerRabatt) {
        // neuerRabatt ist nur von 0 % bis 50 % gültig
        if (neuerRabatt >= 0 && neuerRabatt <= 0.5) {
            this.rabatt = neuerRabatt;
        }
    }

    void legeMehrwertsteuerFest(final double neueMwSt) {
        this.mehrwertsteuer = neueMwSt;
    }

    void fuegePostenHinzu(final int anzahl,
        final double einzelpreis) {
        this.betrag += anzahl * einzelpreis;
    }

    double liefereBetrag() {
        return this.betrag;
    }
}

```

```
double liefereMehrwertsteuer() {
    return this.mehrwertsteuer;
}

double liefereRabatt() {
    return this.rabatt;
}

double berechneNettopreis() {
    return this.betrag * (1 - this.rabatt);
}

double berechneMehrwertsteuer() {
    return this.betrag * (1 - this.rabatt)
        * this.mehrwertsteuer;
}

double berechneBruttoPreis() {
    return this.betrag * (1 - this.rabatt) + this.betrag
        * (1 - this.rabatt) * this.mehrwertsteuer;
}

// ...
}

class Kunde {
    // Attribute
    String name;
    String anschrift;

    // Methoden
    void legeNameFest(final String neuerName) {
        this.name = neuerName;
    }

    void legeAnschriftFest(final String neueAnschrift) {
        this.anschrift = neueAnschrift;
    }

    String liefereName() {
        return name;
    }

    String liefereAnschrift() {
        return anschrift;
    }

    // ...
}
```


Lösung zu Selbsttestaufgabe 18.3-10:

Hauptsächlich tauchen in den Methoden `berechneMehrwertsteuer()` und `berechneBruttoPreis()` Quelltextverdoppelungen auf. Zugleich können auch die direkten Zugriffe mit Hilfe der Getter- und Setter-Methoden verringert werden.

```
class Rechnung {
    // Attribute
    double betrag = 0;
    double mehrwertsteuer;
    double rabatt;
    Kunde rechnungsempfaenger;

    // Methoden
    void legeRabattFest(final double neuerRabatt) {
        this.rabatt = neuerRabatt;
    }

    void legeMehrwertsteuerFest(final double neueMwSt) {
        this.mehrwertsteuer = neueMwSt;
    }

    void fuegePostenHinzu(final int anzahl,
        final double einzelpreis) {
        this.betrag += anzahl * einzelpreis;
    }

    void legeRechnungsempfaengerFest(final Kunde empfaenger) {
        this.rechnungsempfaenger = empfaenger;
    }

    double liefereBetrag() {
        return this.betrag;
    }

    double liefereMehrwertsteuer() {
        return this.mehrwertsteuer;
    }

    double liefereRabatt() {
        return this.rabatt;
    }

    Kunde liefereRechnungsempfaenger() {
        return this.rechnungsempfaenger;
    }

    double berechneNettopreis() {
        return this.liefereBetrag() * (1 - this.liefereRabatt());
    }
}
```

```
double berechneMehrwertsteuer() {
    return this.berechneNettopreis()
        * this.liefereMehrwertsteuer();
}

double berechneBruttoPreis() {
    return this.berechneNettopreis()
        + this.berechneMehrwertsteuer();
}

void gebeAus() {
    System.out.println("An:");
    System.out.println(this.liefereRechnungsempfaenger().
        liefererName());
    System.out.println(this.liefereRechnungsempfaenger().
        liefererAnschrift());
    System.out.print("Netto: ");
    System.out.println(this.berechneNettopreis());
    System.out.print("MwSt: ");
    System.out.println(this.berechneMehrwertsteuer());
    System.out.print("Brutto: ");
    System.out.println(this.berechneBruttoPreis());
}
}
```

Lösung zu Selbsttestaufgabe 18.3-11:

```
class Artikel {
    long artikelnr;
    String beschreibung;
    double preis;

    void legeArtikelnummerFest(final long neueNr) {
        this.artikelnr = neueNr;
    }

    void legeBeschreibungFest(final String neueB) {
        this.beschreibung = neueB;
    }

    void legePreisFest(final double neuerPreis) {
        this.preis = neuerPreis;
    }

    long liefereArtikelnummer() {
        return this.artikelnr;
    }

    String liefereBeschreibung() {
        return this.beschreibung;
    }
}
```

```
double lieferePreis() {
    return this.preis;
}

class Rechnungsposten {
    int anzahl;
    Artikel artikel;

    void legeAnzahlFest(final int neueAnzahl) {
        this.anzahl = neueAnzahl;
    }

    void legeArtikelFest(final Artikel neuerArtikel) {
        this.artikel = neuerArtikel;
    }

    double berechneGesamtbetrag() {
        return this.liefereAnzahl()
            * this.liefereArtikel().lieferePreis();
    }

    int liefereAnzahl() {
        return this.anzahl;
    }

    Artikel liefereArtikel() {
        return this.artikel;
    }
}

class Rechnung {

    double betrag = 0;
    // ...

    void fuegePostenHinzu(final Rechnungsposten posten) {
        this.betrag += posten.berechneGesamtbetrag();
    }

    // ...

}
```

Lösung zu Selbsttestaufgabe 18.4-1:

Für die Klasse Kunde:

```
Kunde(final String name, final String anschrift) {
    this.legeNameFest(name);
    this.legeAnschriftFest(anschrift);
}
```

Für die Klasse Rechnungsposten:

```
Rechnungsposten(final int anzahl, final Artikel artikel) {
    this.legeAnzahlFest(anzahl);
    this.legeArtikelFest(artikel);
}
```

Für die Klasse Artikel:

```
Artikel(final long artikelnr, final String beschreibung,
        final double preis) {
    this.legeArtikelnummerFest(artikelnr);
    this.legeBeschreibungFest(beschreibung);
    this.legePreisFest(preis);
}

Artikel(final long artikelnr, final double preis) {
    this(artikelnr, "", preis);
}
```

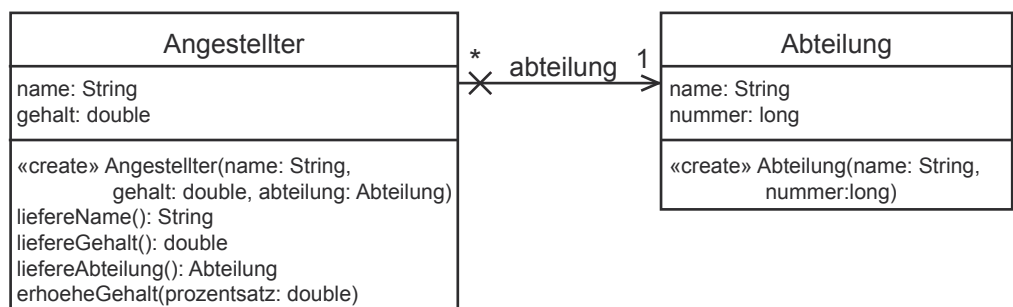
Lösung zu Selbsttestaufgabe 18.4-2:

Abb. ML 15: UML-Diagramm mit Angestellter und Abteilung

```
class Angestellter {
    String name;
    double gehalt;
    Abteilung abteilung;

    Angestellter(final String name, final double gehalt,
        final Abteilung abteilung) {
        this.name = name;
    }
}
```

```

        this.gehalt = gehalt;
        this.abteilung = abteilung;
    }

    String liefereName() {
        return this.name;
    }

    double liefereGehalt() {
        return this.gehalt;
    }

    Abteilung liefereAbteilung() {
        return this.abteilung;
    }

    void erhoeheGehalt(final double prozentsatz) {
        this.gehalt *= (1 + prozentsatz);
    }
}

class Abteilung {
    String name;
    long nummer;

    Abteilung(final String name, final long nummer) {
        this.name = name;
        this.nummer = nummer;
    }

    // hier könnten noch entsprechende Getter- und Setter-
    // Methoden ergänzt werden, diese sind jedoch nicht gefordert
}

```

Lösung zu Selbsttestaufgabe 18.4-3:

Auto
tankvolumen: double aktuellerFuellstand: double verbrauch: double
«create» Auto(tankvolumen: double, verbrauch: double) liefereTankvolumen(): double liefereAktuellenFuellstand(): double liefereVerbrauch(): double tanken(menge: double) fahren(strecke: double)

Abb. ML 16: UML-Diagramm mit Auto

```
class Auto {
    double tankvolumen;
    double aktuellerFuellstand = 0;
    double verbrauch;

    Auto(final double tankvolumen, final double verbrauch) {
        this.tankvolumen = tankvolumen;
        this.verbrauch = verbrauch;
    }

    double liefereAktuellenFuellstand() {
        return this.aktuellerFuellstand;
    }

    double liefereVerbrauch() {
        return this.verbrauch;
    }

    double liefereTankvolumen() {
        return this.tankvolumen;
    }

    void tanken(final double menge) {
        if (this.liefereAktuellenFuellstand()
            + menge > this.liefereTankvolumen()) {
            System.out.print("Es konnte nicht die gesamte ");
            System.out.println("Menge getankt werden.");
            this.aktuellerFuellstand = liefereTankvolumen();
        } else {
            this.aktuellerFuellstand += menge;
        }
    }

    void fahren(final double strecke) {
        double benoetigtesBenzin = strecke
            * this.liefereVerbrauch() / 100;
        if (benoetigtesBenzin
            > this.liefereAktuellenFuellstand()) {
            System.out.print("Es ist nicht genug Benzin für ");
            System.out.println("diese Strecke vorhanden.");
        } else {
            this.aktuellerFuellstand -= benoetigtesBenzin;
            System.out.print("Es wurden ");
            System.out.print(strecke);
            System.out.println(" km zurueckgelegt.");
        }
    }
}
```

```

class Autofahrt {
    public static void main(String[] args) {
        Auto a = new Auto(40, 5);
        a.tanken(60);
        a.fahren(800);
        a.tanken(10);
        a.fahren(1000);
        a.fahren(10);
    }
}

```

Lösung zu Selbsttestaufgabe 19.1-1:

```

class Kreis {
    static final double PI = 3.14159265;
    double radius;

    Kreis(double radius) {
        this.radius = radius;
    }

    double liefereRadius() {
        return this.radius;
    }

    double berechneUmfang() {
        return this.liefereRadius() * 2 * Kreis.PI;
    }

    double berechneFlaecheninhalt() {
        return this.liefereRadius() * this.liefereRadius()
            * Kreis.PI;
    }
}

```

Lösung zu Selbsttestaufgabe 19.2-1:

Die Methode `g(int y)` ist fehlerhaft, da zum einen in einer Klassenmethode das Schlüsselwort `this` nicht verwendet werden darf und auch ohne `this` nicht auf die Exemplarvariable `z` zugegriffen werden darf.

Lösung zu Selbsttestaufgabe 19.2-2:

Um an die Attribute eines Objekts heran zu kommen, muss man einen Verweis auf ein Objekt haben. Auch eine Klassenmethode, die eine Referenz auf ein Objekt besitzt, kann auf dessen Attribute zugreifen. Eine Referenz kann durch einen Konstruktor- oder Methodenaufruf innerhalb der Klassenmethode oder aber durch die Übergabe eines Objekts als Argument entstehen.

Lösung zu Selbsttestaufgabe 20.1-1:

Es tritt in beiden Fällen eine `ArrayIndexOutOfBoundsException` auf.

Lösung zu Selbsttestaufgabe 20.1-2:

Die ersten beiden Aussagen sind richtig, die dritte ist falsch, und die letzte erzeugte den Fehler `ArrayIndexOutOfBoundsException` 4, weil der größte Feldindex den Wert 3 hat.

Lösung zu Selbsttestaufgabe 20.1-3:

Es gibt drei Lösungsmöglichkeiten:

Bei der ersten Alternative belegen wir jedes Feldelement per Einzelzuweisung:

```
int[] eightInts;  
eightInts = new int[8];  
eightInts[0] = 0;  
eightInts[1] = 1;  
eightInts[2] = 2;  
eightInts[3] = 3;  
eightInts[4] = 4;  
eightInts[5] = 5;  
eightInts[6] = 6;  
eightInts[7] = 7;
```

Bei der zweiten Alternative verwenden wir die Aufzählung:

```
int[] eightInts = {0, 1, 2, 3, 4, 5, 6, 7};
```

Die eleganteste Lösung benutzt eine `for`-Schleife :

```
int[] eightInts = new int[8];  
for (int j = 0; j < eightInts.length; j++) {  
    eightInts[j] = j;  
}
```

Lösung zu Selbsttestaufgabe 20.1-4:

```
class Rechnung {  
    final static int MAX_POSTEN = 100;  
    static int naechsteRechnungsnummer = 10000;  
  
    double rabatt;  
    Kunde rechnungsempfaenger;  
    Rechnungsposten[] posten  
        = new Rechnungsposten[Rechnung.MAX_POSTEN];  
    int postenAnzahl = 0;
```



```
final int rechnungsnummer;

Rechnung(Kunde empfaenger) {
    this.rechnungsempfaenger = empfaenger;
    this.rechnungsnummer
        = Rechnung.berechneNaechsteRechnungsnummer();
}

static int berechneNaechsteRechnungsnummer() {
    return Rechnung.naechsteRechnungsnummer++;
}

void fuegePostenHinzu(final Rechnungsposten posten) {
    if (postenAnzahl >= Rechnung.MAX_POSTEN) {
        System.out.print("Es wurde schon die maximal ");
        System.out.print("zulaessige Anzahl an Posten ");
        System.out.println("hinzugefuegt.");
        return;
    }
    this.posten[postenAnzahl] = posten;
    this.postenAnzahl++;
}

void legeRabattFest(final double neuerRabatt) {
    this.rabatt = neuerRabatt;
}

void legeRechnungsempfaengerFest(Kunde empfaenger) {
    this.rechnungsempfaenger = empfaenger;
}

double liefereRabatt() {
    return this.rabatt;
}

Kunde liefereRechnungsempfaenger() {
    return this.rechnungsempfaenger;
}

double berechneNettopreis() {
    double summe = 0;
    for (int i = 0; i < this.postenAnzahl; i++) {
        Rechnungsposten rp = this.posten[i];
        summe += rp.berechneGesamtbetrag();
    }
    return summe * (1 - this.liefereRabatt());
}

double berechneMehrwertsteuer() {
    double summe = 0;
    for (int i = 0; i < this.postenAnzahl; i++) {
        Rechnungsposten rp = this.posten[i];
```

```
        summe += rp.berechneGesamtbetrag()
            * rp.liefereArtikel().liefereMehrwertsteuer();
    }
    return summe * (1 - this.liefereRabatt());
}

double berechneBruttoPreis() {
    return this.berechneNettopreis()
        + this.berechneMehrwertsteuer();
}

void gebeAus() {
    System.out.println("An:");
    System.out.println(this.liefereRechnungsempfaenger().
        liefereName());
    System.out.println(this.liefereRechnungsempfaenger().
        liefereAnschrift());
    System.out.println("Artikel:");
    for (int i = 0; i < postenAnzahl; i++) {
        Rechnungsposten rp = this.posten[i];
        System.out.print(rp.liefereAnzahl());
        System.out.print(" x Nr. ");
        System.out.print(rp.liefereArtikel().
            liefereArtikelnummer());
        System.out.print(" ");
        System.out.println(rp.liefereArtikel().
            liefereBeschreibung());
    }

    System.out.print("Netto: ");
    System.out.println(this.berechneNettopreis());
    System.out.print("MwSt: ");
    System.out.println(this.berechneMehrwertsteuer());
    System.out.print("Brutto: ");
    System.out.println(this.berechneBruttoPreis());
}
}

class Rechnungsposten {
    int anzahl;
    Artikel artikel;

    Rechnungsposten(final int anzahl, final Artikel artikel) {
        this.legeAnzahlFest(anzahl);
        this.legeArtikelFest(artikel);
    }

    void legeAnzahlFest(final int neueAnzahl) {
        this.anzahl = neueAnzahl;
    }

    void legeArtikelFest(final Artikel neuerArtikel) {
```

```

        this.artikel = neuerArtikel;
    }

    double berechneGesamtbetrag() {
        return this.liefereAnzahl()
            * this.liefereArtikel().lieferePreis();
    }

    int liefereAnzahl() {
        return this.anzahl;
    }

    Artikel liefereArtikel() {
        return this.artikel;
    }
}

class Artikel {
    long artikelnr;
    String beschreibung;
    double preis;
    double mehrwertsteuer;

    Artikel(final long artikelnr, final String beschreibung,
            final double preis, double mehrwertsteuer) {
        this.legeArtikelnummerFest(artikelnr);
        this.legeBeschreibungFest(beschreibung);
        this.legePreisFest(preis);
        this.legeMehrwertsteuerFest(mehrwertsteuer);
    }

    void legeArtikelnummerFest(final long neueNr) {
        this.artikelnr = neueNr;
    }

    void legeBeschreibungFest(final String neueB) {
        this.beschreibung = neueB;
    }

    void legeMehrwertsteuerFest(double neueMehrwertsteuer) {
        this.mehrwertsteuer = neueMehrwertsteuer;
    }

    void legePreisFest(final double neuerPreis) {
        this.preis = neuerPreis;
    }

    long liefereArtikelnummer() {
        return this.artikelnr;
    }
}

```

```
String liefereBeschreibung() {
    return this.beschreibung;
}

double lieferePreis() {
    return this.preis;
}

double liefereMehrwertsteuer() {
    return this.mehrwertsteuer;
}
}
```

Lösung zu Selbsttestaufgabe 20.2-1:

```
int[][] dreieck = new int[8][];
for (int i = 0; i < dreieck.length; i++) {
    dreieck[i] = new int[i + 1];
    for (int j = 0; j < i + 1; j++) {
        dreieck[i][j] = i + j;
    }
}
```

Lösung zu Selbsttestaufgabe 20.2-2:

```
double[][] einheitsmatrix = new double[6][6];
for (int i = 0; i < einheitsmatrix.length; i++) {
    einheitsmatrix[i][i] = 1.0;
}
```

Lösung zu Selbsttestaufgabe 20.2-3:

1. Es werden drei ineinander verschachtelte for-Schleifen benötigt.
2. Das Feld enthält $3 \cdot 5 \cdot 4 = 60$ Elemente.
3.

```
int[][][] my3dArray;
my3dArray = new int[3][5][4];
int summe = 0;
for (int i = 0; i < my3dArray.length; i++) {
    for (int j = 0; j < my3dArray[i].length; j++) {
        for (int k = 0; k < my3dArray[i][j].length; k++) {
            summe += my3dArray[i][j][k];
        }
    }
}
```

Lösung zu Selbsttestaufgabe 20.2-4:

```

class Matrix {

    double[][] eintraege;

    Matrix(int x, int y) {
        if (x <= 0 || y <= 0) {
            System.out.println("Ungueltige Matrixgroesse.");
            return;
        }
        this.eintraege = new double[x][y];
    }

    Matrix(final double[][] eintraege) {
        if (eintraege.length <= 0 || eintraege[0].length <= 0) {
            System.out.println("Keine gueltige Matrix.");
            return;
        }
        // hier sollten die Werte kopiert werden
        // damit bei Veraenderungen des uebergebenen Feldes
        // nicht die eigenen Eintraege veraendert werden
        this.eintraege
            = new double[eintraege.length][eintraege[0].length];
        for (int i = 0; i < this.eintraege.length; i++) {
            double[] e = eintraege[i];
            if (e.length != this.eintraege[0].length) {
                System.out.print("Eine Matrix muss ");
                System.out.println("rechteckig sein.");
                return;
            }
            for (int j = 0; j < this.eintraege[i].length; j++) {
                this.eintraege[i][j] = e[j];
            }
        }
    }

    Matrix transponiere() {
        Matrix transponiert = new Matrix(
            this.eintraege[0].length, this.eintraege.length);
        for (int i = 0; i < this.eintraege.length; i++) {
            for (int j = 0; j < this.eintraege[i].length; j++) {
                transponiert.eintraege[j][i]
                    = this.eintraege[i][j];
            }
        }
        return transponiert;
    }

    Matrix addiereMit(Matrix m) {
        if (m.eintraege.length != this.eintraege.length
            || m.eintraege[0].length
                != this.eintraege[0].length) {

```

```

        System.out.print("Die Matrizen haben nicht die ");
        System.out.println("gleiche Groesse.");
        return null;
    }
    Matrix summe = new Matrix(
        this.eintraege.length, this.eintraege[0].length);
    for (int i = 0; i < this.eintraege.length; i++) {
        for (int j = 0; j < this.eintraege[i].length; j++) {
            summe.eintraege[i][j]
                = this.eintraege[i][j] + m.eintraege[i][j];
        }
    }
    return summe;
}
}

```

Lösung zu Selbsttestaufgabe 20.2-5:

```

int maximum(int[] feld) {
    if (feld.length <= 0) {
        return 0;
    }
    int max = feld[0];
    for (int i = 0; i < feld.length; i++) {
        if (feld[i] > max) {
            max = feld[i];
        }
    }
    return max;
}

int minimum(int[][][] feld) {
    // diese Methode funktioniert nur,
    // wenn der erste Feldeintrag existiert
    int min = feld[0][0][0];
    for (int i = 0; i < feld.length; i++) {
        for (int j = 0; j < feld[i].length; j++) {
            for (int k = 0; k < feld[i][j].length; k++) {
                if (feld[i][j][k] < min) {
                    min = feld[i][j][k];
                }
            }
        }
    }
    return min;
}

```

Lösung zu Selbsttestaufgabe 20.3-1:

```
int maximum(int[] feld) {  
    if (feld.length <= 0) {  
        return 0;  
    }  
    int max = feld[0];  
    for (int x : feld) {  
        if (x > max) {  
            max = x;  
        }  
    }  
    return max;  
}
```


Index

A

abweisende Schleife **116**
 Akteur **11**
 Aktion **39**
 Aktivität **38**
 Aktivitätsdiagramm **38**
 Aktivitätsendknoten **39**
 Algorithmus **34**
 von Euklid **35**
 Anforderungsanalyse **9**
 Anpassbarkeit **37**
 Anweisung **102**
 einfache **102**
 Anweisungssequenz **102**
 Anwendungsfall **10**
 Beschreibungsschema **16**
 Anwendungsfallbeschreibung **16**
 Anwendungsfalldiagramm **12**
 API **47**
 arithmetischer Ausdruck **78**
 Auswertungsreihenfolge **80**
 arithmetischer Operator **77**
 Assoziation
 im UML-Klassendiagramm **30**
 zwischen Akteur und
 Anwendungsfall **11**
 zwischen Klassen **24**

Attribut **23**

 -zugriff **151**
 Deklaration **164**
 finales **164**
 im UML-Klassendiagramm **29**
 Initialisierung **164**

Standardbelegung **164**

Attributwert **23**

Aufruf mit Verweisübergabe **175**

Aufruf mit Wertübergabe **175**

Ausdruck

 arithmetischer **78**

Auswertungsreihenfolge **80**

B

Block **102**

boolean **82**

boolean-Literal **82**

boolescher Operator **82**

break-Anweisung **120**

byte **72**

Bytecode **48**

C

cast-Operator **98**

char **83**

char-Literal **84**

class **163**

class-Datei **48**

continue-Anweisung **122**

D

Datenlexikon **19**

Datentyp **72**

 boolean **82**

 byte **72**

 char **83**

 double **75**

 einfacher **72**

 elementarer **72**

float **75**

int **72**

long **72**

primitiver **72**

short **72**

Dekrementoperator **91**

Determiniertheit **34**

Determinismus **34**

Dezimalliteral **73**

do-Schleife **115**

double **75**

E

Effektivität **34**

einfache Fallunterscheidung **105**

Enthält-Beziehung **13**

Entscheidungsknoten **39**

Erweitert-Beziehung **14**

erzwungene Typumwandlung **98**

EVA-Prinzip **44**

Exemplarvariable **164**

explizite Typanpassung **98**

«extend» **14**

F

Fallunterscheidung

 einfache **105**

 Mehrfach- **109**

Feld **185**

 Erzeugung **186**

 Erzeugung durch Aufzählung **186**

 Länge **187**

 mehrdimensionales **189**

 Standardbelegung **187**

 Tabelle **190**

 Vereinbarung **185**

Feldindex **188**

Feldtypen **185**

Feldzugriff **187**

final **90**

finale Variable **90**

finale Attribut **164**

Fintheit **34**

float **75**

for-Schleife **116**

formale Parameter **166**

G

Gültigkeitsbereich **124**

Ganze Zahlen **72**

 Wertebereiche **72**

Geheimnisprinzip **43, 173**

Generalisierungsbeziehung

 in der UML **31**

 zwischen Akteuren **12**

 zwischen Anwendungsfällen **14**

 zwischen Klassen **25**

Getter-Methode **172**

Gleitkommazahlen **75**

H

Hexadezimalliteral **73**

I

if-Anweisung **105**

implizite Typanpassung **93**

«include» **14**

Initialisierung **89**

Initialknoten **39**

Inkrementoperator **91**

int **72**

int-Literal **73**

J

java-Dateien **48**

K

Klasse **23, 163**

Klassenattribut **180**

Klassendiagramm **29**

Klassenmethode **182**

Klassenvariable **180**

Klassenvereinbarung **163**

Kompatibilitätsregeln **100**

Komplexität **37**

Konstante **181**

Konstruktor **176**

Aufruf **153**

Signatur **177**

Konstruktordeklaration **177**

Konstruktorname

überladener **177**

Kontrollknoten **39**

Kontrollstrukturen **38, 104**

Korrektheit **36**

L

Laufzeit-Fehler **50**

Literal **73**

für Gleitkommazahlen **75**

long **72**

long-Literal **74**

M

main()-Methode **50**

Maschinensprache **44**

Mehrfach-Fallunterscheidung **109**

Mehrfachvererbung **26**

Methode **24**

Aufruf **151**

Deklaration **165**

im UML-Klassendiagramm **30**

Signatur **167**

Methodenaufruf **151**

Methodenname

überladener **168**

Methodenrumpf **166**

Modell **2**

Modellierungssprache **2**

Multiplizitäten **31**

N

Nachricht **27**

Nebeneffekt **92**

new **153**

nicht abweisende Schleife **116**

null **157**

O

Oberklasse **25**

Objekt **22**

Identität **157**

Objektdiagramm **32**

Objekttypen **149**

Objektzustand **151**

Oktalliteral **73**

Operator **77**

arithmetischer **77**

boolescher **82**

cast- **98**

Dekrement- **91**

Inkrement- **91**

Rangfolge **79**

Vergleichs- **81**

P

Parameter

formaler **166**

Pflichtenheft **20**

Präzedenzregeln **79**

Programmiersprachen **42**

R

Rückgabeanweisung **171**

Rückwärtskante **40**

Referenz **154**

return-Anweisung **171**

Robustheit **37**

Rolle **11**

S

Schlüsselwörter **86**

Schleife **40**

abweisende **116**

do- **115**

for- **116**

nicht abweisende **116**

while- **113**

Schleifenanweisung **113**

Schleifenrumpf **114**

Seiteneffekt **92**

Sequenz **39, 102**

Setter-Methode **172**

short **72**

Sichtbarkeitsbereich **124**

Softwareentwicklung **2**

Standardfall **110**

Standardkonstruktor **177**

static **180**

Steuerzeichen **84**

Strukturdiagramm **6**

strukturierte Sprunganweisung **119**

Substantivanalyse **29**

System.out.println() **184**

Szenario **16**

T

Tabelle **190**

Terminierung **34**

this **170**

-Konstruktoraufruf **178**

Typanpassung **93**

explizite **98**

implizite **93**

Typeinengung **98**

Typerweiterung **93**

Typumwandlung **93**

erzwungene **98**

Typverträglichkeit **93**

U

überladener Konstrukturname **177**

überladener Methodenname **168**

Überlauf **96**

Übersetzer-Warnungen **50**

Übersetzungsfehler **49**

Überwachungsbedingung **39**

UML **4**

UML-Aktivitätsdiagramm **38**

UML-Klassendiagramm **29**

UML-Objektdiagramm **32**

UML-Schlüsselwort

«create» **153**

«extend» **14**

«include» **14**

Unicode **83**

Unified Modeling Language **4**

Unterklasse **25**

V

Variable **85**

 finale **90**

 lokale **168**

Variablendeklaration **86**

Variablenname **86**

Variablenvereinbarung **86**

Verdecken

 von Attributen **170**

Vereinigungsknoten **40**

Vererbung **25**

Vergleichsoperator **81**

Verhaltensdiagramm **6**

Verknüpfung **24**

Verweisvariable **155**

Verzweigung **39**

Virtuelle Maschine **47**

VM **47**

void **166**

von Neumann-Architektur **44**

W

Wertvariable **154**

while-Schleife **113**

Wiederholung **40**

Wiederholungsanweisung **113**

Wiederverwendbarkeit **37**

Z

Zeichen **84**

Zustand **24, 151**

Zuweisung **88**

 Auswertung **95**

Zuweisungsoperator **88**

