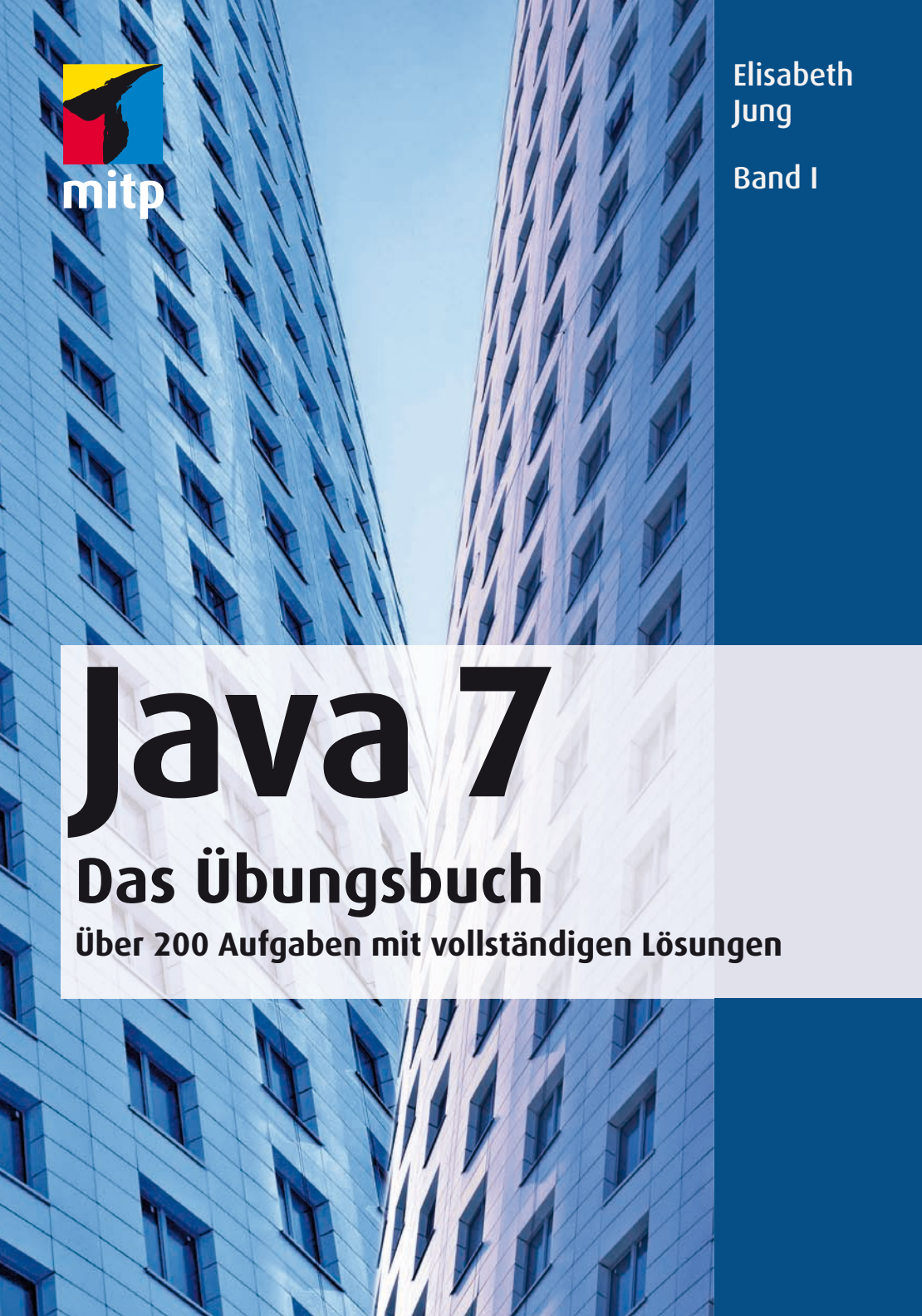

mitp
Elisabeth Jung
Band I
Java 7
Das Übungsbuch
Über 200 Aufgaben mit vollständigen Lösungen

Kapitel 1

Klassendefinition und Objektinstantiierung

1.1 Klassen und Objekte

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

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

Eine Schnittstelle (Interface) ist eine reine Spezifikation, die definiert, wie eine Klasse sich zu verhalten hat. Sie wird mit dem Schlüsselwort `interface` eingeleitet und kann keine Implementationen von Feldern und Methoden enthalten, mit Ausnahme von Konstantendefinitionen.

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

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

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

Die Klassen bilden in Java eine Klassenhierarchie. Jede Klasse hat eine Oberklasse, deren Felder und Methoden sie erbt. Die Oberklasse aller Klassen in Java ist die Klasse `java.lang.Object` (s. Kapitel 2, Abgeleitete Klassen und Vererbung und Kapitel 3, Abstrakte Klassen und Interfaces).

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

Konstruktoren

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

Klassenfelder und Klassenmethoden

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

Klassenmethoden werden ebenfalls über den Klassennamen angesprochen.

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

Instanzfelder und Instanzmethoden

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

Das Schlüsselwort `this` bezeichnet die Referenz auf das Objekt, an dem die Methode bzw. der Konstruktor aufgerufen wird, auch das »aktuelle Objekt« bzw. das »aufrufende Objekt« genannt. Konstruktoren können nur mit dem Schlüssel-

wort new aufgerufen werden, sowie innerhalb eines anderen Konstruktors über das Schlüsselwort this, gefolgt von der Parameterliste.

Java-Programme

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

Ein Java-Programm manipuliert Werte, die durch Typ und Name gekennzeichnet werden. Über die Festlegung von Name und Typ wird eine Variable definiert. Man spricht von Variablen in Zusammenhang mit einem Programm und von Feldern in Zusammenhang mit einer Klassendefinition. Variablen können auf mehrere Arten und Weisen klassifiziert werden. Eine Aufteilung wäre in Variablen von einem primitiven Datentyp bzw. einem Referenztyp und eine zweite in Variablen, die einen Wert oder mehrere Werte (Arrayvariablen) aufnehmen können. Außerdem spricht man von Klassenvariablen, wenn die Felder einer Klasse gemeint sind, Instanzvariablen, wenn die Felder einer Instanz gemeint sind, und Arrayelementen, wenn die einzelnen Elemente eines Array gemeint sind. Von Parametervariablen wird in Methodendeklarationen gesprochen und von lokalen Variablen, wenn diese innerhalb von einem Methodenrumpf oder Anweisungsblock definiert wurden.

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

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

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

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

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

- Ein Java-Applet ist ein Java-Programm, das im Kontext eines Webbrowsers mit bestimmten Sicherheitseinschränkungen abläuft. Es wird mittels einer HTML-Seite gestartet und kann im Browser oder mit Hilfe des Appletviewer ausgeführt werden.
- Ein Servlet ist ein Java-Programm, das im Kontext eines Webservers abläuft.
- Eine Java-Applikation ist ein eigenständiges Programm, das direkt von der JVM gestartet wird.

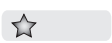
Jede Java-Applikation benötigt eine `main()`-Methode, welche einen Eingangspunkt für die Ausführung des Programms durch die JVM definiert. Diese Methode muss für alle Klassen der JVM zugänglich sein und deshalb mit dem Modifikator `public` definiert werden. Sie muss auch mit dem Modifikator `static`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

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

Gleich in den ersten Beispielen werden für Bildschirmausgaben die Methoden `System.out.print(…)` und `System.out.println(…)` der Klasse `java.io.PrintStream` verwendet. Mit der Methode `System.out.print(…)` wird in der gleichen Bildschirmzeile weiter geschrieben, in welcher eine vorangehende Ausgabe erfolgte. Die Methode `System.out.println()` ohne Parameter schließt eine vorher ausgegebene Bildschirmzeile ab und bewirkt, dass danach in eine neue Zeile geschrieben wird. Ein Aufruf der Methode `System.out.println(…)` mit Parameter ist äquivalent mit dem Aufruf von `System.out.print(…)` gefolgt von einem Aufruf von `System.out.println()` ohne Parameter, d.h. dieser Aufruf führt immer zu einem Zeilenende. Auch auf die Definition von diesen Methoden kommen wir in der Beschreibung der Java-Standard-Klasse `java.lang.System` noch mal zurück.

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

Aufgabe 1.1



Definition einer Klasse

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

Hinweise für die Programmierung:

Ein Erzeugen von Instanzen der Klasse ist nicht erforderlich.

Achten Sie auf den richtigen Abschluss der Ausgabezeile.

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

Aufgabe 1.2 ☆

Objekt (Instanz) einer Klasse erzeugen

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

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

1.2 Die Member einer Klasse: Felder und Methoden

Aufruf von Instanz- und Klassenmethoden

Aufruf von Instanzmethoden:

- aus der gleichen Klasse
  - aus einer Instanzmethode – direkt über den Methodennamen und die Parameterliste.
  - aus einer Klassenmethode – es muss ein Objekt der Klasse instantiiert werden und die Instanzmethode wird an diesem Objekt aufgerufen.
- aus einer anderen Klasse
  - es muss ein Objekt der Klasse instantiiert werden und die Instanzmethode wird an diesem Objekt aufgerufen.

Aufruf von Klassenmethoden:

- aus der gleichen Klasse
  - direkt über den Methodennamen und die Parameterliste.
- aus einer anderen Klasse
  - über den Klassennamen, gefolgt von einem Punkt und dem Methodennamen und der Parameterliste.

Zugriff auf Instanz- und Klassenfelder

Zugriff auf Instanzfelder:

- aus der gleichen Klasse
  - aus einer Instanzmethode – direkt über den Feldnamen.

  - aus einer Klassenmethode – es muss ein Objekt der Klasse instantiiert werden und auf das Feld wird über die Objektreferenz gefolgt von einem Punkt und dem Feldnamen zugegriffen.
- aus einer anderen Klasse
  - es muss ein Objekt der Klasse instantiiert werden und auf das Feld wird über die Objektreferenz zugegriffen.

Zugriff auf Klassenfelder:

- aus der gleichen Klasse
  - direkt über den Feldnamen.
- aus einer anderen Klasse
  - über den Klassennamen, gefolgt von einem Punkt und dem Feldnamen.

Aufgabe 1.3 ☆

Zugriff auf Felder

Eine Klasse `FeldZugriffe` soll den Unterschied zwischen Klassen- und Instanzfeldern demonstrieren.

Sie definiert ein Klassenfeld `zaehlerAlsKlassenfeld` und ein Instanzfeld `zaehlerAlsInstanzfeld`. Die Werte von beiden Feldern werden im parameterlosen Konstruktor der Klasse inkrementiert (um eins erhöht). Definieren Sie eine Klassenmethode `anzeigeKlsMeth()` und eine Instanzmethode, `anzeigeInstMeth()`, welche jeweils beide Feldwerte am Bildschirm ausgeben.

Zum Testen dieser Klassendefinition wird die Klasse `FeldZugriffeTest` erstellt, die in ihrer `main()-Methode` eine Instanz der Klasse `FeldZugriffe` erzeugt, an welcher die Instanzmethode der Klasse aufgerufen wird. Ihre Klassenmethode wird über den Klassennamen aufgerufen.

Java-Dateien: `FeldZugriffe.java, FeldZugriffeTest.java`
Programmaufruf: `java FeldZugriffeTest`

Aufgabe 1.4 ☆☆

Aufruf von Methoden

Definieren Sie eine Klasse `MethodenAufrufe1,` welche die Instanzmethoden `instMethode1()` und `instMethode2()` und die Klassenmethoden `klsMethode1()` und `klsMethode2()` implementiert und eine zweite Klasse `MethodenAufrufe2` mit der Instanzmethode `instMethode()` und der Klassenmethode `klsMethode()`. Die Methode `instMethode1()` gibt am Bildschirm die Zeichenkette: "1. Instanzmethode der Klasse MethodenAufrufe1" aus und ruft die Metho-

den `instMethode2()` und `klsMethode1()` auf. Die Methode `instMethode2()` gibt die Zeichenkette: "2. Instanzmethode der Klasse MethodenAufrufe1« aus und die Methode `klsMethode1()` die Zeichenkette: "1. Klassenmethode der Klasse MethodenAufrufe1«. Die Methode `klsMethode2()`, gibt am Bildschirm die Zeichenkette: "2. Klassenmethode der Klasse MethodenAufrufe1" aus und ruft die Methoden `instMethode2()` und `klsMethode1()` auf.

Die Methoden der Klasse `MethodenAufrufe2` rufen beide die Methoden `instMethode1()` und `klsMethode2()` der Klasse `MethodenAufrufe1` auf.

Zum Testen des Aufrufes von Instanz- und Klassenmethoden aus Methoden der gleichen Klasse oder einer anderen Klasse wird die Klasse `MethodenAufrufeTest` erstellt, die in ihrer `main()`-`Methode` eine Instanz der Klasse `MethodenAufrufe2` erzeugt und deren Methoden aufruft.

Java-Dateien: `MethodenAufrufe1.java`, `MethodenAufrufe2.java`, `MethodenAufrufeTest.java`
Programmaufruf: `java MethodenAufrufeTest`

1.3 Das Überladen von Methoden

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

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

Aufgabe 1.5 ☆

Eine Methode überladen

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

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

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

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

1.4 Die Datenkapselung, ein Prinzip der objektorientierten Programmierung

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

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

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

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

Aufgabe 1.6 ☆

Zugriffsmethoden

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

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

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

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

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

Aufgabe 1.7 ☆☆

Konstruktordefinitionen

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

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

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

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

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

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

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

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

1.6 Die Wert- und Referenzübergabe in Methodenaufrufen

In Java-Methoden werden alle Argumente, ob es Werte von primitiven Typen oder Referenzen sind, als Kopie per Wert übergeben. Der Mechanismus der Wertübergabe wird auch »call by value« bzw. »copy per value« genannt. Wenn ein Argument übergeben wird, wird dessen Wert an eine Speicheradresse in den Stack der Methodenaufrufe (»method call stack«) kopiert. Egal ob dieses Argument eine Variable von einem primitiven oder Referenztyp ist, wird der Inhalt der Kopie als Parameterwert übergeben, nur diese kann innerhalb der Methode abgeändert werden und

nicht der Wert selbst. D. h., eine Parametervariable wird als lokale Variable betrachtet, die zum Zeitpunkt des Methodenaufrufs mit dem entsprechenden Argument initialisiert wird und nach dem Beenden der Methode nicht mehr existiert.

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

Aufgabe 1.8

☆☆☆

Wertübergabe in Methoden (»call by value«)

Die Klasse `MethodenParameter`, definiert drei Klassenmethoden mit den Signaturen `methode1(int x, int[] y)`, `methode2(Punkt x, Punkt[] y)` und `methode3(Punkt x)`, wobei `Punkt` eine Referenz vom Typ der Klasse aus der Aufgabe 1.6 bezeichnet.

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

Hinweise für die Programmierung:

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

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

1.7 Globale und lokale Referenzen

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

Aufgabe 1.9

☆☆

Der Umgang mit Referenzen

Definieren Sie eine Klasse `GlobaleReferenzen`, welche an Stelle der lokalen Variablen aus den Methoden der Klasse `MethodenParameter` globale Programmvariablen definiert, und die Methoden selbst ohne Parametervariablen.

Hinweise für die Programmierung:

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

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

Aufgabe 1.10 ☆☆☆

Wiederholungsaufgabe

Definieren Sie eine Klasse `GanzeZahlen`, die ein Instanzfeld `z` vom Typ `int` besitzt, und einen Konstruktor, der einen `int`-Wert zu dessen Initialisierung übergeben bekommt.

Definieren Sie die Zugriffsmethoden `getZahl()` und `setZahl()` zum Lesen und Setzen des Attributwertes der Klasse und vier weitere Instanzmethoden `negativ()`, `gleich()`, `kleiner()` und `anzeige()` zum Setzen eines negativen Vorzeichens, zur »Gleich-Abfrage«, zur »Kleiner-Abfrage« und zur Anzeige von `GanzeZahlen`- Objekten.

Der »größte gemeinsame Teiler« und das »kleinste gemeinsame Vielfache« von `GanzeZahlen`-Instanzen sollen mit Hilfe von zwei Klassenmethoden `ggTeiler()` und `kgVielfaches()` berechnet werden. Implementieren Sie zwei Klassenmethoden `add()` und `divid()` für das Ermitteln von Summe und Quotient und zwei Instanzmethoden `subtr()` und `multipl()` für das Ermitteln der Differenz und des Produktes von `GanzeZahlen`-Objekten.

Definieren Sie eine weitere Klasse `RationaleZahlen`, die zwei globale Referenzen vom Typ der Klasse `GanzeZahlen` besitzt, welche den Zähler und Nenner einer rationalen Zahl beschreiben. Die Klasse definiert die Zugriffsmethoden `setZaehler()`, `setNenner()`, `getZaehler()` und `getNenner()` und in Analogie zu den Methoden der Klasse `GanzeZahlen` Methoden für die Anzeige und Durchführung von Operationen mit `RationeleZahlen`-Instanzen, aus welchen die gleichnamigen Methoden der Klasse `GanzeZahlen` aufgerufen werden. Für das Kürzen der Werte von Zähler und Nenner soll eine Instanzmethode `kuerzen()` implementiert werden.

Die Klasse `ZahlenTest` definiert zwei globale Referenzen `z1` und `z2` vom Typ der Klasse `GanzeZahlen` als Instanzfelder und zwei Klassenfelder `GANZEZAHLEN` und `RATIONALEZAHLEN` mit konstanten Werten vom Typ `int` für die Beschreibung von Objektzuständen. Eine Konstantendefinition erfolgt in Java mit dem Schlüsselwort `final`. Das dritte Instanzfeld `zahlenTyp` bekommt im Konstruktor der Klasse einen dieser konstanten Werte zugewiesen und dient als Parameter in einer `switch`-Anweisung, der Auswahl von Methodenaufrufen zur Instantiierung von Objekten von Typ `GanzeZahlen` bzw. `RationaleZahlen`. Implementieren Sie für

diese Klasse zwei Methoden `defGanzeZahlen()` und `defRationaleZahlen()`, welche Operationen mit diesen Instanzen durchführen.

Methodenaufrufe können in Java über den »Punkt-Operator« gekettet werden. Rufen Sie für die Ausgabe der Ergebnisse am Bildschirm die Methode `anzeige()` gekettet an die Methodenaufrufe `add()`, `subtr()`, `multpl()` und `divid()` auf.

Hinweise für die Programmierung:

Die Methoden einer Klasse können sowohl auf Feldwerte, als auch auf Objekte der eigenen Klasse operieren. Die Entscheidung, ob Methoden als Instanz- oder Klassenmethoden definiert werden, ist dem Programmierer überlassen. Methoden, welche keine Eigenschaften des Objekts der Klasse benutzen, sollten als Klassenmethoden definiert werden, Methoden, die nur ein Objekt der Klasse benutzen, als Instanzmethoden. Für Methoden, die mehrere Objekte einer Klasse manipulieren, können beide Definitionsarten in Betracht gezogen werden.

Beachten Sie bei einer Übergabe von Referenzen, dass Veränderungen, die innerhalb von Methoden an einem übergebenen Objekt durchgeführt werden, auch außerhalb der Methode sichtbar bleiben und dass Zuweisungen von Objektreferenzen nicht wie Zuweisungen von primitiven Datentypen zum Erzeugen eines neuen Objektes, als Kopie eines schon vorhandenen führen. Wird einer Referenzvariablen eine Objektreferenz zugewiesen, so zeigt die Variable auf dieses Objekt, ohne dass ein neues erstellt wird.

Java-Dateien: `GanzeZahlen.java, RationaleZahlen.java, ZahlenTest.java`
Programmaufruf: `java ZahlenTest`

1.8 Selbstreferenzierende Klassen und Felder (»self-referential classes and fields«)

Selbstreferenzierende Klassen sind Klassen, die mindestens ein Feld als globale Referenz vom Typ der eigenen Klasse definieren. Als selbstreferenzierende Felder werden Felder vom Typ der eigenen Klasse bezeichnet. Diese werden auch Links genannt, weil sie es ermöglichen, Objekte von der gleichen Klasse zu verbinden, indem ihre Referenzen jedem anderen selbstreferenzierenden Feld zugewiesen werden können.

Aufgabe 1.11 ☆☆☆

Der Einsatz von selbstreferenzierenden Feldern

Das Vorwärts- und Rückwärtsblättern in einem Buch soll mit Hilfe von Instanzen einer selbstreferenzierenden Klasse mit dem Namen `Buch` illustriert werden. Dafür definiert die Klasse `Buch` ein Instanzfeld `seite` vom Typ `int`, ein selbstreferenzierendes Instanzfeld `naechsteSeite`, über welches Objekte der Klasse verknüpft werden können, und Zugriffsmethoden für das Setzen und Lesen deren Werte. Im

Konstruktoraufruf der Klasse wird ein `int`-Wert übergeben und für das Instanzfeld `naechsteSeite` eine `null`-Referenz gesetzt.

Die Klasse `Buch` definiert zwei Klassenmethoden `rueckwaertsBlaettern()` und `vorwaertsBlaettern()`, die zwei Argumente vom Typ `int` für die Angabe der Seitennummern, zwischen welchen geblättert werden soll, besitzen.

Definieren Sie eine Klasse `BuchTest`, die die Methoden der Klasse `Buch` mit verschiedenen Angaben von Seitenbereichen aufruft.

Hinweise für die Programmierung:

In den Methoden wird der Referenz auf ein Objekt der Klasse die Referenz auf das gerade vorher konstruierte Objekt der Klasse zugeordnet. Damit verbindet das selbstreferenzierende Feld alle Objekte der Klasse in der Methode `rueckwaertsBlaettern()` vom ersten zum letzten und in der Methode `vorwaertsBlaettern()` vom letzten zum ersten. Beim Verknüpfen der Felder bekommt das als erstes erzeugte Objekt in der Methode `rueckwaertsBlaettern()` und das als letztes erzeugte in der Methode `vorwaertsBlaettern()` eine `null`-Referenz zugeordnet, weil diese keinen Vorgänger bzw. Nachfolger besitzen.

Für die Anzeige der durchgeblätterten Seiten am Bildschirm kann in einer `while`-Schleife mit der Methode `getNaechsteSeite()` nacheinander auf die verknüpften Instanzen der Klasse zugegriffen werden.

Java-Dateien: `Buch.java, BuchTest.java`
Programmaufruf: `java BuchTest`

1.9 Java-Pakete

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

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

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

Die Namen von Klassen und deren Pakete werden vom Compiler in die schon vorher erwähnte Klassendatei, die mit dem Suffix `.class` gespeichert wird, eingetragen. Diese Datei ist eine Unterstützung für den JVM-Klassenlader beim Auffinden der Klasse. Eine zusätzliche Hilfe ist auch die Umgebungsvariable `CLASSPATH`, die eine Liste von Dateiverzeichnissen und Namen von Archivdateien für die Suche zur Verfügung stellen kann. Archivdateien sind Dateien, die selbst andere Dateien

beinhalten und werden in Java mit dem Suffix `.jar` abgeschlossen. Unter Windows wird die CLASSPATH-Variable über das Betriebssystemkommando: `set classpath = c:\pfadname1;pfadname2;archivname1;...` gesetzt und mit `set classpath = .;` gelöscht.

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

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

Alle bis jetzt verwendeten Klassen wurden ohne Modifikator definiert. Eine Klasse ohne `public` ist nicht uneingeschränkt öffentlich, es können nur Klassen aus dem gleichen Paket Instanzen davon erzeugen. Eine Klasse hat nur zwei Zugriffsebenen: standard (ohne Modifikator) und `public`. Eine mit `public` definierte Klasse ist für alle anderen Klassen zugänglich und muss immer in einer Java-Datei mit gleichem Namen gespeichert werden.

Aufgabe 1.12 ☆

Die package-Anweisung

Für ein Dateiverzeichneis `kapitel1`, wird ein Unterverzeichnis `paket1` definiert. Erstellen Sie eine Klasse `PackageTest`, die in ihrer `main()`-Methode, die Zeichenkette "Test der package-Anweisung" am Bildschirm ausgibt und speichern Sie diese, als die Java-Datei `PackageTest.java`, im Verzeichnis `paket1` ab. Wie muss die `package`-Anweisung in dieser Klassendefinition lauten, so dass das Programm im Verzeichnis `kapitel1` übersetzt und ausgeführt werden kann?

Hinweise zu den Programmaufrufen:

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

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

Aufgabe 1.13 ☆

Die import-Anweisung

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

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

Das mit der Klasse `KlassenTest` erstellte Java-Programm soll im Verzeichnis `paket1` übersetzt und ausgeführt werden. Die Verwendung des Klassennamens `Klasse` kann entweder über eine `import`-Anweisung erfolgen oder es muss immer der Präfix `paket2.` angegeben werden.

Java-Dateien: `kapitel1\paket1\KlassenTest.java`,
`kapitel1\paket1\paket2\Klasse.java`

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

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

Auf ein Member einer Klasse, das ohne Modifikator definiert wurde, kann von außerhalb eines Paketes nicht zugegriffen werden. Nur mit `public` deklarierte Member sind uneingeschränkt öffentlich. Ein mit `protected` definiertes Member ist außerhalb eines Paketes nur für abgeleitete Klassen einer Klasse sichtbar. Weitere Ergänzungen zu diesen Aussagen können im Kapitel 2, Abgeleitete Klassen und Vererbung, gelesen werden.

Aufgabe 1.14 ☆☆

Pakete und die Sichtbarkeit von Membern einer Klasse

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

Die Klasse `Klasse1` definiert drei Klassenfelder vom Typ `int`: `privatesFeld`, `geschuetztesFeld`, `oeffentlichesFeld` mit den Modifakatoren `private`, `pro-`

tected, public und ein weiteres Klassenfeld feld ohne Modifikator. Sie soll die Zeichenkette »Instanz der Klasse1« am Bildschirm anzeigen und den Paketnamen über die Anweisung: package paket1; angeben. Die Klasse Klasse2 soll die Zeichenkette »Instanz der Klasse2« am Bildschirm anzeigen und die Anweisung: package paket2; fuer die Angabe des Paketnamens definieren.

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

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

Java-Dateien: kapitel1\PackageTest1.java, kapitel1\paket1\Klasse1.java, kapitel1\paket1\paket2\Klasse2.java
Programmaufrufe im Verzeichnis kapitel1: javac PackageTest1.java und java PackageTest1

1.11 Standard-Klassen von Java

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

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

Die Klasse System

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

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

Die Instanz, auf welche die Referenz `out` zeigt, wird auch mit den Methoden `System.out.print(…)` und `System.out.println(…)` der Klasse `PrintStream` genutzt, um Bildschirmausgaben zu realisieren.

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

Die Klasse File

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

Die Klassen Runtime, ProcessBuilder und Process

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

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

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

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

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

Die Klassen Exception und Error

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

Exceptions werden über das Schlüsselwort `throw` (werfen) ausgelöst und können von einem `catch`-Block (fangen) aufgefangen werden, in welchem deren Verarbei-

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

Diese Klassen und ihre Unterklassen befinden sich auch im Paket `java.lang` und werden im Kapitel 9, Exceptions und Errors, ausführlicher behandelt.

Die Klasse Math

Diese Klasse definiert viele mathematische Funktionen und Operationen in Form von Klassenfeldern und Klassenmethoden. Sie ist mit dem Modifikator `final` deklariert, so dass keine Unterklassen von dieser Klasse erzeugt werden können. Der Zugriff auf Felder und Methoden der Klasse erfolgt über ihren Namen, mit dem Klassennamen `Math` als Präfix.

Aufgabe 1.15

Aufruf von Methoden der Klasse Math

Mathematisch gesehen sind zwei Dreiecke kongruent (deckungsgleich), wenn sie in drei Seiten (sss) oder zwei Seiten und dem von diesen Seiten eingeschlossenen Winkel (sws) oder zwei Seiten und dem Gegenwinkel der längeren Seite (ssw) oder einer Seite und den beiden anliegenden Winkeln (wsw) übereinstimmen. So reicht die Angabe einer dieser Gruppen mit drei Größen, um die anderen Seiten und Winkel eines Dreiecks zu bestimmen. Dazu können die Relationen, die in einem beliebigen Dreieck zwischen seinen Seiten a, b, c und Winkeln A, B, C bestehen: $a^2 = b^2+c^2-2*b*c*\cos A$ (Rotationen von Werten sind möglich) und $a/\sin A=b/\sin B= c/\sin C$ genutzt werden. Für das Berechnen von fehlenden Winkelmaßen kann zusätzlich die Relation $A+B+C = 180°$ genutzt werden.

Definieren Sie eine Klasse `KongruenteDreiecke`, die über drei Instanzfelder `x`, `y` und `z` vom Typ `double` Werte für die Spezifikation von Seitenlängen und Winkelmaßen für ein Dreieck aufnehmen kann. Für das Bilden von Instanzen der Klasse wird ein Konstruktor mit drei Argumenten vom Typ `double` definiert, der eine Unterscheidung zwischen Seiten und Winkeln bei deren Übergabe nicht erforderlich macht.

Die Klasse `KongruenteDreiecke` implementiert vier Instanzmethoden: `sss()`, `sws()`, `ssw()` und `wsw()`, zum Erzeugen eines neuen `KongruenteDreiecke`-Objektes aus dem aktuellen, indem im Konstruktor der Klasse die von der Methode errechneten fehlenden Komponenten übergeben werden. Die Methode `sss()` soll also aus den drei Seiten die drei Winkel berechen, die Methode `sws()` aus den zwei Seiten und einem Winkel die dritte Seite und die beiden anderen Winkel, usw.

Diese Methoden rufen die Methoden: `cos()`, `sin()`, `acos()` und `asin()` der Klasse `Math` für die Berechnung der Werte auf.

Definieren Sie zwei weitere Methoden `winkelRadian()` und `winkelGrad()` für die Umsetzung der Winkelmaße von Grad in Radiant und Radiant in Grad. Diese haben eine Objektinstantiierung nicht nötig und sollten deshalb als Klassenmethoden definiert werden. Oder Sie verwenden an deren Stelle die Methoden der Klasse `Math: toRadians() und toDegrees()`. Verwenden Sie beim Rechnen die in der Klasse `Math` definierte Konstante `Math.PI` oder Sie definieren wie im Lösungsvorschlag für die Aufgabe ein eigenes Klassenfeld mit dem gleichen Wert.

Definieren Sie eine Klasse `KongruenteDreieckeTest`, die Instanzen dieser Klasse bildet und ihre Methoden aufruft. Der Zugriff auf die neu errechneten Komponenten von Instanzen der Klasse `KongruenteDreiecke` soll zwecks Bildschirmausgabe über die von der Klasse definierten Zugriffsmethoden erfolgen. Beim Erzeugen von Objekten der Klasse `KongruenteDreiecke` soll die gleiche Referenzvariable sowohl auf das über den Konstruktor der Klasse erzeugte Objekt als auch auf den Rückgabewert der an diesem Objekt aufgerufenen Methode zeigen.

Java-Dateien: `KongruenteDreieckejava, KongruenteDreieckeTest.java`
Programmaufruf: `java KongruenteDreieckeTest`

Aufgabe 1.16 ☆☆

Wiederholungsaufgabe

Erstellen Sie eine Klasse `Strecke`, die zwei globale Referenzen `P` und `Q` vom Typ der Klasse `Punkt` aus der Aufgabe 1.5 definiert. Im Konstruktor der Klasse werden auch Referenzen auf Objekte der Klasse `Punkt` übergeben. Definieren Sie analog zur Klasse `Punkt` die Zugriffsmethoden `setP()`, `setQ()`, `getP()` und `getQ()` für die Instanzfelder der Klasse. Zur Berechnung der Distanz zwischen zwei Punkten definiert die Klasse eine Methode `distanz()` und zur Anzeige der Gleichung der Geraden, die durch beide Punkte geht, eine Methode `geradenGleichung()`. Die Punktkoordinaten sollen mit Hilfe der Zugriffsmethoden der Klasse `Punkt` ermittelt werden.

Definieren Sie eine Klasse `Kreis`, die als Instanzfeld eine globale Referenz `PQ` vom Typ der Klasse `Strecke` besitzt. Implementieren Sie für diese Klasse zwei Methoden `flaeche()` und `umfang()` zum Berechnen des Flächeninhaltes und der Länge des Kreises mit einem Radius gleich der Länge der Strecke PQ und eine weitere Methode `kreisGleichung()`, welche die Gleichung des Kreises mit dem Mittelpunkt in `P` und dem Radius `PQ` ausgibt. Rufen Sie in diesen die Methode `round()` der Klasse `Math` auf, um die errechneten Ergebnisse zu runden. Die Klasse `Kreis` soll ein zusätzliches Klassenfeld mit dem Namen `PI` definieren, das den Wert der Konstanten `Math.PI` aus der Java-Standard-Klasse `Math` zugewiesen bekommt, und die Zugriffsmethoden `setPQ()` und `getPQ()` für das Instanzfeld der Klasse.

Zum Testen der neuen Klassendefinitionen soll eine Klasse `PunktStreckeKreisTest` definiert werden, welche in ihrer `main()-Methode` Instanzen der Klassen `Punkt`, `Strecke` und `Kreis` erzeugt und an diesen die Methoden der Klassen aufruft.

Java-Dateien: `Punkt.java, Strecke.java, Kreis.java, PunktStreckeKreisTest.java`
Programmaufruf: `java PunktStreckeKreisTest`

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

Sowohl primitive Datentypen als auch Referenztypen legen die Eigenschaften ihrer Werte innerhalb von Klassen in Form von Felddefinitionen fest. Für die Manipulation der Werte stehen für primitive Datentypen Operatoren zur Verfügung, während Klassen Methoden benutzen. Operatoren sind von der Programmiersprache her vordefiniert und können in Java vom Programmierer nicht überladen werden. Methoden bringen den Vorteil mit sich, dass diese ein Überladen erlauben, ohne dass eine bestimmte Anzahl davon vorgegeben wird (s. die Aufgaben 1.5 und 1.22).

Weil höhere Datenstrukturen nur Objekte aufnehmen können, wurden Hüllenklassen, auch Wrapper-Klassen genannt, für alle primitiven Datentypen definiert: `Boolean`, `Byte`, `Integer`, `Float`, `Double`, `Long`, `Short`, `Character`. Während die primitiven Datentypen Bestandteil der Java-Programmiersprache sind, gehören die Wrapper-Klassen zur Java-API. Eine Hüllenklasse definiert ein Instanzfeld vom entsprechenden Datentyp und Methoden, mit welchen dieses manipuliert werden kann. Dies sind z.B. Konvertierungsmethoden wie `toBinaryString()`, `toHexString()` oder Methoden für die Rückgabe des primitiven Wertes des Wrapper-Objektes, wie `intValue()`, `floatValue()`, etc. Alle Hüllenklassen definieren einen Konstruktor, der ein Argument vom Typ des primitiven Datentypes besitzt, und einen Konstruktor mit einem Argument vom Typ der Klasse `String`.

Wrapper-Klassen definieren auch eine Reihe von nützlichen Konstanten wie z. B. `MIN_VALUE` oder `MAX_VALUE`, die den kleinsten bzw. größten Wert des entsprechenden arithmetischen Datentyps bezeichnen. Sie definieren jedoch keine Methoden für arithmetische oder logische Operationen zwischen Objekten einer Klasse oder zur Änderung des innerhalb einer Klasse gespeicherten Wertes.

Diese Klassen sind als `final` deklariert, d.h. sie sind nicht erweiterbar und werden alle von der abstrakten Klasse `Number` abgeleitet, deren Methoden sie implementieren (s. Kapitel 2, Abgeleitete Klassen und Vererbung und Kapitel 3, Abstrakte Klassen und Interfaces).

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

Damit wurde ein weiterer wesentlicher Beitrag zur Typsicherheit von Daten gewährleistet, wie auch mit den in Java 5.0 eingeführten generischen Datentypen, welche im Kapitel 8, Generics beschrieben werden.

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

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

Aufgabe 1.17 ☆☆

Die Felder und Methoden von Wrapper-Klassen

Definieren Sie in einer Klasse mit dem Namen `WrapperKlassen` globale Referenzen vom Typ aller Hüllenklassen von Java. Übergeben Sie im Konstruktor dieser Klasse die korrespondierenden primitiven Werte und erzeugen Sie mit Hilfe der Konstruktoren von Hüllenklassen je eine Instanz für jede dieser Klassen. Rufen Sie nach Belieben die verschiedensten Methoden, wie `byteValue()`, `intValue()`, `toOctalString()`, `getNumericValue()`, `digit()`, etc., dieser Klassen auf.

Zum Bilden von Instanzen der Klasse und Aufruf ihrer Methoden soll eine Klasse `WrapperKlassenTest` erstellt werden.

Java-Dateien: `WrapperKlassen.java, WrapperKlassenTest.java`
Programmaufruf: `java WrapperKlassenTest`

Aufgabe 1.18 ☆

Das Auto(un)boxing

Die Klasse mit dem Namen `WrapperKlassenmitAutoBoxing` definiert die gleichen Instanzfelder wie die Klasse `WrapperKlassen` aus der Aufgabe 1.17 und einen Konstruktor mit den gleichen Argumenten. Prüfen Sie, ob einfache Zuweisungen von primitiven Werten für das Bilden der Instanzen von Hüllenklassen ausreichend sind und ob im Methodenaufruf von `System.out.println()` ein Unboxing bei der Angabe dieser Instanzen durchgeführt wird. Definieren Sie eine Methode mit der Signatur: `public boolean konvert(Boolean b)` und zeigen Sie, dass diese mit einem primitiven Wert vom Typ `boolean` aufgerufen werden kann.

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

Wie auch in der Aufgabe 1.17, soll zum Testen eine weitere Klasse, `WrapperKlassenmitAutoBoxingTest`, erstellt werden.

Hinweise für die Programmierung:

Ein Vergleich mit »==« bleibt weiterhin ein Vergleich von Referenzen und dabei wird kein Unboxing auf primitive Datentypen durchgeführt, sodass ein solcher Vergleich zwischen zwei Wrapper-Objekten trotz gleichem numerischem Wert im Allgemeinen das Ergebnis `false` liefert, wobei es aber Ausnahmen von dieser Regel gibt, je nachdem, wie das Autoboxing intern funktioniert, also ob dabei ein neues Objekt erzeugt wurde (z.B. bei nicht ganzzahligen oder bei großen ganzzahligen Zahlenwerten) oder ein altes Objekt wieder verwendet wurde (z.B. bei kleinen ganzzahligen Zahlenwerten, Zeichen oder logischen Größen).

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

1.13 Arrays (Reihungen) und die Klassen Array und Arrays

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

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

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

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

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

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

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

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

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

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

Aufgabe 1.19

☆☆

Der Umgang mit Array-Objekten

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

Primitive Datentypen können als Objekte der Klasse `Class` von Java über Standard-Namen wie `int.class`, `long.class`, etc. angesprochen werden. Benutzen Sie die Methode `newInstance()` der Klasse `Array`, um ein Array-Objekt dynamisch zu erzeugen und deren Methoden `setInt()` und `getInt()`, um die Elemente des so erzeugten Arrays zu manipulieren. Zeigen Sie am Beispiel der Klasse `Vektor` aus der Aufgabe 1.7, dass Elemente von Referenz-Arrays immer einzeln instantiiert werden müssen und rufen Sie für deren Anzeige am Bildschirm die Methode `anzeige()` der Klasse `Vektor` auf. Achten Sie auf die Java-Klassen bzw. Pakete, welche von der Klasse `ArrayTest1` importiert werden müssen.

Java-Dateien: `ArrayTest1.java`
Programmaufruf: `java ArrayTest1`

1.14 Zeichenketten und die Klasse String

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

Aufgabe 1.20 ☆

Der Umgang mit String-Objekten

Die Java-Standard-Klasse `String` definiert mehrere Konstruktoren. Erzeugen Sie in einer eigenen Klasse `StringTest` mehrere `String`-Objekte, indem Sie einer `String`-Referenz ein Literal direkt zuweisen oder im Konstruktor der Klasse `String` ein Literal, ein `char`-Array oder ein anderes `String`-Objekt übergeben. Rufen Sie an diesen Objekten die Methoden `concat()` und `substring()` der Klasse `String` auf, um einen String zu erweitern oder einen Teilstring davon zu bilden. Mit den Methoden `parseInt()`, `parseDouble()` und `parseShort()` von Wrapper-Klassen, sollen Strings in die angegebenen primitive Typen umgewandelt und diese Werte am Bildschirm angezeigt werden.

Die Klasse `StringTest` definiert eine Methode zum Vergleichen von `String`-Instanzen und ein `String`-Array, dessen Elemente über den Aufruf dieser Methode initialisiert werden.

In der `main()`-Methode jeder Java-Klasse wird ein `String`-Array übergeben, das beim Start der Java-Applikation mit den Kommandozeilen-Parametern, die der Benutzer angibt, initialisiert wird. Rufen Sie dieses Java-Programm mit einer Angabe von Argumentwerten auf und zeigen Sie die Werte der Kommandozeilenparameter am Bildschirm an.

Java-Dateien: `StringTest.java`
Programmaufruf: `java StringTest <Kommandozeilen Parameter>`

1.15 Mit der Version 5.0 eingeführte Spracherneuerungen für Arrays und Methoden

Die for-each-Schleife für Arrays

Die `for`-Schleife kann genutzt werden, um eine Reihe von Daten zu durchlaufen, indem sie einen Index definiert und inkrementiert. Mit Java 5.0 wurde mit der

Anweisung: `for (<Typ> <iterationsVariable>: <iterationsObjekt>;` eine von einem Index unabhängige, bequemere `for`-Schleife in die Sprache eingeführt, die auch für Arrays eingesetzt werden kann.

Damit wird jedes Element der über `iterationsObjekt` angegebenen Menge erreicht, und die `iterationsVariable` entspricht dem Typ dieser Elemente. Dass multidimensionale Arrays in Java Arrays von Arrays sind, ist von Bedeutung für die Iterationen von multidimensionalen Arrays, weil jede Iteration eines Array das nächste Array liefert. So müssen in `for-each`-Schleifen im Falle eines n-dimensionalen Arrays die Iterationsvariablen Referenzen vom Typ eines (n-1)-dimensionalen Array sein und diese wiederum Iterationsvariablen besitzen, die Referenzen vom Typ eines (n-2)-dimensionalen Array sind, etc. Mit `for-each-` Schleifen kann man ein Array nur sequentiell vom Anfang zum Ende durchgehen, sodass sich nicht alle Arten von Algorithmen, wie z.B. ein gezielter Zugriff auf ein Arrayelement, dafür anbieten.

Java-Methoden kennen einen einzigen Rückgabewert. Sollen jedoch mehrere Werte zurückgegeben werden, müssen diese in ein einzelnes Objekt zusammengefasst werden, und das kann auch ein Array sein.

Aufgabe 1.21 ☆

Einfache und erweiterte for-Schleifen

Zur Illustration der vorangehenden Bemerkungen soll eine Klasse `ArrayTest2` erstellt werden, die aus der `main()-Methode` ihre zwei anderen Klassenmethoden `anzeige()` und `rueckgabe()` aufruft. Ein Erzeugen von Instanzen der Klasse ist dafür nicht erforderlich. In der Methode `anzeige()` werden drei lokale Referenzen vom Typ eines eindimensionalen, eines zweidimensionalen und eines dreidimensionalen `int`-Array: `array1`, `array2` und `array3` definiert. Jedes Arrayelement soll mit der Summe seiner Indexwerte initialisiert werden. Da diese Wertezuweisungen vom Index der Elemente abhängig sind, muss dazu die alte Form der `for`-Schleife genutzt werden. Die Anzeige der Elemente am Bildschirm soll jedoch mit Hilfe von `for-each`-Schleifen erfolgen.

Die Methode `rueckgabe()` bekommt drei `char`-Werte übergeben, initialisiert damit die Elemente eines Arrays vom gleichen Typ und gibt dieses an die aufrufende Methode zurück, welche dieses in einer weiteren `for-each`-Schleife am Bildschirm anzeigt. Sie soll auch demonstrieren, wie Arrays definiert werden, damit Werte, die von Methoden zurückgegeben werden sollen, als ein Rückgabewert zusammengefasst werden können.

Java-Dateien: `ArrayTest2.java`
Programmaufruf: `java ArrayTest2`

Arrays als variable Argumentenlisten in Methoden

Mit Java 5.0 können Methoden mit einer variablen Anzahl von Argumenten definiert werden, diese werden auch varargs-Methoden genannt. Ein Parameter, der

eine variable Anzahl von Werten aufnehmen kann wird mit dem »`...`–Operator« definiert, dem der Typ des Parameters vorausgeht. Intern wird ein variables Argument als ein Array vom angegebenen Typ mit unbekannter Länge deklariert. So kann dieses mit einer erweiterten `for`-Schleife durchlaufen werden und auf die tatsächliche Anzahl der Argumente mit dem Schlüsselwort `length`, welches die Dimension eines Arrays definiert, zugegriffen werden.

Aufgabe 1.22 ☆

Methoden mit variablen Argumentenlisten

Definieren Sie eine Klasse `MethodenmitVararg`, die eine Methode mit dem Namen `varArg()` mit einer variablen Anzahl von `char`-Parametern besitzt. Überladen Sie diese Methode, indem Sie den Typ der Argumentenliste abändern bzw. über zwei `int`-Werte erweitern. Der Aufruf dieser Methode soll mit einer unterschiedlichen Anzahl von Argumenten aus der `main()-Methode` der Klasse erfolgen.

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

1.16 Das Initialisieren von Klassen- und Instanzfeldern

Klassenfelder werden in Java über Klassen-Feld- bzw. Klassen-Block-Initialisierer initialisiert. Ein Klassen-Feld-Initialisierer besteht aus einer Zuweisung des Wertes, den das Klassenfeld aufnehmen soll. Wird dieser nicht explizit angegeben, so werden alle Bits des Feldes gleich `0` gesetzt, was einer Initialisierung mit den Default-Werten: `0`, `0.0`, `false` und `null` entspricht.

Ein Klassen-Block-Initialisierer besteht aus dem Schlüsselwort `static` gefolgt von einem in {} eingeschlossenen Initialisierungscode.

Zusätzlich zum Instanz-Feld-Initialisierer unterstützt Java auch sogenannte Instanz-Block-Initialisierer, welche aus einem in {} eingeschlossenen Initialisierungscode bestehen.

Die Konstruktoren einer Klasse führen alle Instanzfeldinitialisierungen durch, auch wenn diese außerhalb des Konstruktors im Programm definiert sind. Der Compiler fügt gleich am Anfang der Übersetzung den gesamten Initialisierungscode für Instanzfelder in den Konstruktor der Klasse ein.

An dieser Stelle sei noch mal darauf hingewiesen, dass Felddeklarationen (mit oder ohne explizite Initialisierung) kein Bestandteil von Methoden sind und deshalb innerhalb von Methoden nicht als Anweisung ausgeführt werden können, wie dies der Fall für lokale Variablen ist.

Selbstreferenzierende Felder können nur dann in ihrer Deklaration initialisiert werden, wenn sie als Klassenfelder definiert wurden. Wenn nämlich ein selbstrefernzierendes Instanzfeld in der Deklaration oder in einem Konstruktor auf ein

entsprechendes Objekt gesetzt würde, ergäbe das eine Endlosschleife: Das Objekt würde für dieses Feld ein Objekt derselben Klasse erzeugen, das dann wieder ein solches Objekt erzeugen würde, das dann wieder usw.

Aufgabe 1.23 ☆

Das Initialisieren von Instanzfeldern

Definieren Sie eine Klasse `InitialisierungInstanzfelder`, welche die Initialisierung für ihre Felder `einlong`, `eindouble` und `einString` in Instanz-Feld-Initialisierern vornimmt, für das `char`-Array `einArray` in einem Instanz-Block-Initialisierer und für die Felder `einPunkt` und `einint` im Konstruktor der Klasse. Die Felder `einshort`, `einfloat` und `einboolean` sollen über Default-Zuweisungen initialisiert werden.

Definieren Sie eine Methode `setselfReferentialFeld()`, die die Initialisierung eines selbreferenzierenden Instanzfeldes `selfReferentialFeld` übernimmt, weil diese Art von Feldern nicht über eine direkte Zuweisung mit dem `new`-Operator in ihrer Deklaration initialisiert werden können.

Um die Instanzfelder zu initialisieren, wird in der `main()-Methode` ein Objekt der Klasse erzeugt.

Die gleiche Methode soll alle Feldwerte am Bildschirm anzeigen. Kann die Anzeige der Werte des `char`-Array über den Methodenaufruf `Arrays.toString(einArray)` erfolgen?

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

Aufgabe 1.24 ☆

Das Initialisieren von Klassenfeldern

Eine Klasse `InitialisierungKlassenfelder` definiert Klassenfelder mit den gleichen Namen wie die Instanzfelder aus der Aufgabe 1.23. Die Klassenfelder sollen in Analogie zur Klasse `InitialisierungInstanzfelder` mit Hilfe von Klassen-Feld-Initialisierern, Klassen-Block-Initialisierern und über Defaultwerte initialisiert werden. Im Unterschied zu selbstreferenzierenden Instanzfeldern können selbstreferenzierende Klassenfelder in ihrer Deklaration mit Hilfe des `new`-Operators initialisiert werden.

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

1.17 Private Konstruktoren

Private Konstruktoren werden mit dem Modifikator `private` definiert und erlauben nicht, dass Objekte dieser Klasse innerhalb von anderen Klassen erzeugt wer-

den. Die Klasse selbst kann ein oder mehrere Objekte erzeugen und diese allen anderen Klassen über einen Methodenaufruf oder über konstante Klassenfelder zur Verfügung stellen.

Aufgabe 1.25 ☆

Ein Objekt mit Hilfe eines privaten Konstruktoren erzeugen

Die Klasse `PrivaterKonstruktor` definiert einen privaten Konstruktor, der das aktuelle Datum und die gerade gemessene Uhrzeit am Bildschirm anzeigen soll. Diese Daten können über die Methoden der Java-Standard-Klasse `GregorianCalendar` ermittelt werden, zu welcher weitere Details im Kapitel 10 aufgeführt werden.

Definieren Sie dazu ein selbreferenzierendes Klassenfeld mit dem Namen `datumunduhrzeit und` eine Methode `getInstanz()`, welche ein Objekt der Klasse erzeugt und anderen Klassen zur Verfügung stellt.

Definieren Sie eine Klasse `PrivaterKonstruktorTest`, die in ihrer `main()`-Methode die Methode `getInstanz()` aufruft und sich so das erzeugte Objekt vom Typ der Klasse `PrivaterKonstruktor` holt.

Java-Dateien: `PrivaterKonstruktor.java, PrivaterKonstruktorTest.java`
Programmaufruf: `java PrivaterKonstruktorTest`

Aufgabe 1.26 ☆☆

Mehrere konstante Werte (Objekte) mit Hilfe eines privaten Konstruktoren erzeugen

Mit Hilfe einer Klasse `WochenTage` soll ein zweiter sinnvoller Einsatz von Klassen mit privaten Konstruktoren gezeigt werden. Im privaten Konstruktor dieser Klasse wird ein `int`-Wert übergeben, über welchen das öffentliche Instanzfeld `tag` der Klasse mit einem String, der den Namen des entsprechenden Wochentages angibt, initialisiert wird. Die Klasse definiert mit dem Modifikator `static final` sieben globale Referenzen mit dem Namen der Wochentage, die auf Objekte der eigenen Klasse zeigen, und im Konstruktor die Werte 1 bis 7 zugewiesen bekommen. Diese werden als `public` definiert, damit sie auch für andere Klassen uneingeschränkt zugänglich sind.

Eine zweite Klasse `WochenTageTest` definiert einen Konstruktor mit einem Parameter vom Typ der Klasse `WochenTage` und erzeugt in ihrer `main()`-Methode sieben Instanzen der eigenen Klasse, welche als Argument im Konstruktor Objekte der Klasse `WochenTage` übergeben bekommen.

Java-Dateien: `WochenTage.java, WochenTageTest.java`
Programmaufruf: `java WochenTageTest`

1.18 Lösungen

Lösung 1.1

Die Klasse KlassenDefinition

```
class KlassenDefinition {
   public static void main(String args[]) {
// Ausgabe einer Zeichenkette
     System.out.println("Definition einer einfachen "+
       "Java-Klasse");
   }
}
```

Programmausgaben

```
Definition einer einfachen Java-Klasse
```

Lösung 1.2

Die Klasse ObjektInstantiierung

```
class ObjektInstantiierung {
// Konstruktordefinition
   public ObjektInstantiierung() {
     System.out.println("Instanz einer Java-Klasse erzeugen");
   }
   public static void main(String args[]) {
     ObjektInstantiierung instanz = new ObjektInstantiierung();
   }
}
```

Programmausgaben

```
Instanz einer Java-Klasse erzeugen
```

Lösung 1.3

Die Klasse FeldZugriffe

```
class FeldZugriffe {
// Instanzfeld
   int zaehlerAlsInstanzfeld;
// Klassenfeld
   static int zaehlerAlsKlassenfeld;
// Konstruktordefinition
     FeldZugriffe() {
       zaehlerAlsKlassenfeld++;
       zaehlerAlsInstanzfeld++;
   }
```

```
// Instanzmethoden, haben direkten Zugriff auf Instanz- und
// Klassenfelder
  public void anzeigeInstMeth() {
    System.out.print("Zaehler als Instanzfeld: "
     + zaehlerAlsInstanzfeld);
// Zeilenumbruch ausgeben
    System.out.println();
    System.out.print("Zaehler als Klassenfeld: "
     + FeldZugriffe.zaehlerAlsKlassenfeld);
// Innerhalb der Klasse kann auf den Klassennamen verzichtet
// werden
    System.out.print("*"+zaehlerAlsKlassenfeld);
    System.out.println();
  }
// Klassenmethoden haben direkten Zugriff auf Klassenfelder,
// darin können nur die Instanzfelder eines erzeugten Objektes
// der Klasse angesprochen werden
  public static void anzeigeKlsMeth() {
    System.out.print("Zaehler als Klassenfeld: "
     + zaehlerAlsKlassenfeld);
    System.out.print("*"+FeldZugriffe.zaehlerAlsKlassenfeld);
// Zeilenumbruch
    System.out.println();
  // System.out.println(zaehlerAlsInstanzfeld); // Fehler
// Instanz vom Typ der Klasse erzeugen
    FeldZugriffe t = new FeldZugriffe();
    System.out.print("Zaehler als Instanzfeld: "
     + t.zaehlerAlsInstanzfeld);
    System.out.print("*"+t.zaehlerAlsKlassenfeld);
  }
}
Die Klasse FeldZugriffeTest
class FeldZugriffeTest {
  public static void main(String args[]) {
    FeldZugriffe t = new FeldZugriffe();
    t.anzeigeInstMeth();
    FeldZugriffe.anzeigeKlsMeth();
  }
}
```

Programmausgaben

```
Zaehler als Instanzfeld: 1
Zaehler als Klassenfeld: 1*1
Zaehler als Klassenfeld: 1*1
Zaehler als Instanzfeld: 1*2
```

Hinweise zu den Programmausgaben

Es ist zu erkennen, dass das Instanzfeld für jedes erzeugte Objekt neu initialisiert wird, das Klassenfeld jedoch für alle erzeugten Instanzen der Klasse zählt.

Lösung 1.4

Die Klasse MethodenAufrufe1

```
class MethodenAufrufe1 {
// Instanzmethoden
   public void instMethode1() {
      System.out.println("1. Instanzmethode der Klasse "
        + "MethodenAufrufe1");
// Aufruf einer Instanzmethode aus einer Instanzmethode der
// gleichen Klasse
      instMethode2();
// Aufruf einer Klassenmethode aus einer Instanzmethode der
// gleichen Klasse
      MethodenAufrufe1.klsMethode1();
   }
   public void instMethode2() {
      System.out.println("2. Instanzmethode der Klasse "
       + "MethodenAufrufe1");
   }
// Klassenmethoden
   public static void klsMethode1() {
      System.out.println("1. Klassenmethode der Klasse "
        + "MethodenAufrufe1");
   }
   public static void klsMethode2() {
      System.out.println("2. Klassenmethode der Klasse "
        + "MethodenAufrufe1");
// Aufruf einer Instanzmethode aus einer Klassenmethode der
// gleichen Klasse
      MethodenAufrufe1 instanz = new MethodenAufrufe1();
      instanz.instMethode2();
// Aufruf einer Klassenmethode aus einer Klassenmethode der
// gleichen Klasse
      MethodenAufrufe1.klsMethode1();
   }
}
```

Die Klasse MethodenAufrufe2

```
class MethodenAufrufe2 {
 // Instanzmethode
   public void instMethode() {
      System.out.println("Instanzmethode der Klasse "
```

```
      + "MethodenAufrufe2");
// Aufruf einer Instanzmethode aus einer Instanzmethode einer
// anderen Klasse
     MethodenAufrufe1 instanz1 = new MethodenAufrufe1();
     instanz1.instMethode1();
// Aufruf einer Klassenmethode aus einer Instanzmethode einer
// anderen Klasse
     MethodenAufrufe1.klsMethode2();
   }
// Klassenmethode
   public static void klsMethode() {
     System.out.println("Klassenmethode der Klasse "
       + "MethodenAufrufe2");
// Aufruf einer Instanzmethode aus einer Klassenmethode einer
// anderen Klasse
     MethodenAufrufe1 instanz2 = new MethodenAufrufe1();
     instanz2.instMethode1();
// Aufruf einer Klassenmethode aus einer Klassenmethode einer
// anderen Klasse
     MethodenAufrufe1.klsMethode2();
   }
}
```

Die Klasse MethodenAufrufeTest

```
class MethodenAufrufeTest {
   public static void main(String args[]) {
     MethodenAufrufe2.klsMethode();
     MethodenAufrufe2 instanz = new MethodenAufrufe2();
     instanz.instMethode();
   }
}
```

Programmausgaben

```
Klassenmethode der Klasse MethodenAufrufe2
1. Instanzmethode der Klasse MethodenAufrufe1
2. Instanzmethode der Klasse MethodenAufrufe1
1. Klassenmethode der Klasse MethodenAufrufe1
2. Klassenmethode der Klasse MethodenAufrufe1
2. Instanzmethode der Klasse MethodenAufrufe1
1. Klassenmethode der Klasse MethodenAufrufe1
Instanzmethode der Klasse MethodenAufrufe2
1. Instanzmethode der Klasse MethodenAufrufe1
2. Instanzmethode der Klasse MethodenAufrufe1
1. Klassenmethode der Klasse MethodenAufrufe1
2. Klassenmethode der Klasse MethodenAufrufe1
```

```
2. Instanzmethode der Klasse MethodenAufrufe1
1. Klassenmethode der Klasse MethodenAufrufe1
```

Lösung 1.5

Die Klasse QuadratDefinition

```
class QuadratDefinition  {
   int a;
// Konstruktordefinition
   QuadratDefinition(int x) {
      a = x;
   }
// Instanzmethode zum Berechnen des Flächeninhaltes
   public int flaeche() {
      int f = a*a;
      return f;
   }
// Gleichnamige Klassenmethode zum Berechnen des Flächeninhaltes
   public static int flaeche(QuadratDefinition q) {
      int f = q.a*q.a;
      return f;
   }
}
```

Die Klasse QuadratDefinitionTest

```
class QuadratDefinitionTest {
   public static void main(String args[]) {
      System.out.println("Instanz der Klasse erzeugen");
      QuadratDefinition quadrat = new QuadratDefinition(4);
      System.out.println("Aufruf der Instanzmethode");
      int finst = quadrat.flaeche();
      System.out.println("Flaeche: "+finst);
      System.out.println("Aufruf der Klassenmethode");
      int fkls2 = QuadratDefinition.flaeche(quadrat);
      System.out.println("Flaeche: "+fkls2);
   }
}
```

Programmausgaben

```
Instanz der Klasse erzeugen
Aufruf der Instanzmethode
Flaeche: 16
Aufruf der Klassenmethode
Flaeche: 16
```

Lösung 1.6

Die Klasse Punkt

```
class Punkt {
   private double x;
   private double y;
// Konstruktordefinition
   Punkt(double a, double b) {
      x = a;
      y = b;
   }
// Zugriffsmethoden
   public void setX(double X) {
      x = X;
      System.out.println("x-Wert gesetzt");
   }
   public void setY(double Y) {
      y = Y;
      System.out.println("y-Wert gesetzt");
   }
   public double getX() {
      return x;
   }
   public double getY() {
      return y;
   }
// Instanzmethode für eine Punktanzeige
   public void anzeige() {
      System.out.println("("+ x +","+ y +")");
   }
}
```

Die Klasse PunktTest

```
class PunktTest {
   public static void main(String args[]) {
// Erzeugen einer Punkt-Instanz
      Punkt p = new Punkt(1.5, -4.5);
// Anzeige der Punkt-Instanz
      p.anzeige();
// Setzen und Lesen von Punktkoordinaten
      p.setX(-4.0);
      p.setY(4.0);
      System.out.println("x = "+p.getX());
      System.out.println("y = "+p.getY());
   }
}
```

Programmausgaben

```
(1.5, -4.5)
x-Wert gesetzt
y-Wert gesetzt
x = -4.0
y = 4.0
```

Lösung 1.7

Die Klasse Vektor

```
public class Vektor {
   private int x;
   private int y;
   private int z;
// Konstruktordefinitionen
   public Vektor() {
      this(0,0,0);
   }
// Besitzen Methoden- oder Konstruktorenparameter die gleichen
// Namen wie Instanzfelder, müssen die Instanzfelder über this
// angesprochen werden
   public Vektor(int x, int y, int z) {
      this.x = x;
      this.y = y;
      this.z = z;
   }
// Ein mit dem Copy-Konstruktor erzeugtes Objekt hat die
// gleichen Feldwerte wie das übergebene
   public Vektor(Vektor v) {
      x = v.x;
      y = v.y;
      z = v.z;
   }
// Methodendefinitionen
   public Vektor neu(int a, int b, int c) {
      this.x = this.x+a;
      this.y = this.y+b;
      this.z = this.z+c;
      return this;
   }
   public Vektor neu(Vektor v) {
      x = x+v.x;
      y = y+v.y;
      z = z+v.z;
      return this;
   }
```

```
   public Vektor neu1(int a, int b, int c) {
      Vektor vektor = new Vektor(this);
      vektor.x = x+a;
      vektor.y = y+b;
      vektor.z = z+c;
      return vektor;
   }
   public void anzeige() {
      System.out.print("("+ x +","+ y +","+ z +") ");
   }
}
```

Die Klasse VektorTest

```
class VektorTest {
   public static void main(String args[]) {
// Erzeugen von Vektor-Objekten mit den unterschiedlichen
// Konstruktoren
      Vektor v1 = new Vektor();
      v1.anzeige();
      Vektor v2 = new Vektor(1,1,1);
      v2.anzeige();
      Vektor v3 = new Vektor(v2);
      v3.anzeige();
// Erzeugen eines neuen Vektor-Objektes durch Veränderung des
// aufrufenden Objektes
      Vektor vneu = v1.neu(3,3,3);
      v1.anzeige();
      vneu.anzeige();
// oder
      v1 = new Vektor();
      Vektor vcopy = v1.neu(new Vektor(3,3,3));
      v1.anzeige();
      vcopy.anzeige();
// Erzeugen eines neuen Vektor-Objektes durch Beibehalten des
// aufrufenden Objektes
      v1 = new Vektor();
      Vektor vneu1 = v1.neu1(3,3,3);
      v1.anzeige();
      vneu1.anzeige();
   }
}
```

Programmausgaben

```
(0,0,0) (1,1,1) (1,1,1) (3,3,3) (3,3,3) (3,3,3) (3,3,3) (0,0,0) (3,3,3)
```

Lösung 1.8

Die Klasse MethodenParameter

```
class MethodenParameter {
// Methodendefinitionen
   public static void methode1(int x, int[] y) {
      x = 0;
      y[1] = 0;
   }
   public static void methode2(Punkt x, Punkt[] y) {
      x.setX(0.0);
      x.setY(0.0);
      y[1].setX(0.0);
      y[1].setY(0.0);
   }
   public static void methode3(Punkt x) {
      x = new Punkt(-1.0,-1.0);
   }
   public static void main(String args[]) {
// Lokale primitive und Referenz-Variablen
      int i = 1;
      int[] iarray = {1,2,3};
      Punkt p = new Punkt(4.0, 3.0);
      Punkt[] parray = {new Punkt(1.0, 1.0), new Punkt(2.0,2.0)};
      System.out.println("int-Variable  vor dem Aufruf von "
        + "methode1(): "+i);
      System.out.print("int-Array vor dem Aufruf von "
        + "methode1(): ");
      for(int j=0; j<iarray.length; j++) {
         System.out.print(iarray[j]+" ");
      }
      System.out.println();
      methode1(i,iarray);
      System.out.println("int-Variable  nach dem Aufruf von "
        + "methode1(): "+i);
      System.out.print("int-Array nach dem Aufruf von "
        + "methode1(): ");
      for(int j=0; j<iarray.length; j++) {
         System.out.print(iarray[j]+" ");
      }
      System.out.println();
      System.out.print("Die Referenzvariable vom Typ Punkt zeigt"
        + " vor dem Aufruf von methode2() auf das Objekt: ");
// Die Methode anzeige() der Klasse Punkt wird aufgerufen
      p.anzeige();
      System.out.println("Punkt-Array vor dem Aufruf von "
        + "methode2(): ");
```

```
    for(int j=0; j<parray.length; j++) {
       parray[j].anzeige();
    }
    methode2(p, parray);
    System.out.print("Die Referenzvariable vom Typ Punkt zeigt"
     + " nach dem Aufruf von methode2() auf das gleiche Objekt"
       + " mit abgeaenderten Werten: ");
    p.anzeige();
    System.out.println("Punkt-Array nach dem Aufruf von "
     + "methode2(): ");
    for(int j=0; j<parray.length; j++) {
       parray[j].anzeige();
    }
    System.out.print("Die Referenzvariable vom Typ Punkt zeigt"
     +" vor dem Aufruf von methode3() auf das Objekt: ");
    p.anzeige();
    methode3(p);
    System.out.print("Die Referenzvariable vom Typ Punkt zeigt"
     + " nach dem Aufruf von methode3() auf das gleiche "
       + "Objekt: ");
    p.anzeige();
  }
}
```

Programmausgaben

```
int-Variable vor dem Aufruf von methode1(): 1
int-Array vor dem Aufruf von methode1(): 1 2 3
int-Variable  nach dem Aufruf von methode1(): 1
int-Array nach dem Aufruf von methode1(): 1 0 3
Die Referenzvariable vom Typ Punkt zeigt vor dem Aufruf von methode2() auf
das Objekt: (4.0,3.0)
Punkt-Array vor dem Aufruf von methode2():
(1.0,1.0)
(2.0,2.0)
x-Wert gesetzt
y-Wert gesetzt
x-Wert gesetzt
y-Wert gesetzt
Die Referenzvariable vom Typ Punkt zeigt nach dem Aufruf von methode2()
auf das gleiche Objekt mit abgeaenderten Werten: (0.0,0.0)
Punkt-Array nach dem Aufruf von methode2():
(1.0,1.0)
(0.0,0.0)
Die Referenzvariable vom Typ Punkt zeigt vor dem Aufruf von methode3() auf
das Objekt: (0.0,0.0)
Die Referenzvariable vom Typ Punkt zeigt nach dem Aufruf von methode3()
auf das gleiche Objekt: (0.0,0.0)
```

Hinweise zum Lösungsvorschlag

Nach dem Aufruf von `methode1()` bleibt der Wert der übegebenen `int`-Variablen `i gleich 1` und die Referenz `p` vom Typ `Punkt` zeigt nach dem Aufruf von `methode3()` weiter auf den Punkt (0.0,0.0) wie vor dem Aufruf der Methode, auch wenn diese innerhalb der Methoden abgeändert wurden.

Veränderungen, die innerhalb von Methoden an einem Parameterobjekt durchgeführt werden, bleiben jedoch auch außerhalb der Methode sichtbar, wie dies der Aufruf von `methode2()` zeigt.

Referenzen vom Typ Array können innerhalb von Methoden nicht abgeändert werden, weil sie wie alle Referenzen per Wert übergeben werden. D. h., der Parameter `y` bekommt die in `iarray` bzw. `parray` gespeicherte Array-Referenz übergeben und nicht eine Referenz auf diese. Weil aber der Methodenparameter `y` eine Referenz auf ein Array enthält, das durch `iarray` bzw.`parray` referenziert wird, können die Arrayelemente abgeändert werden.

Lösung 1.9

Die Klasse GlobaleReferenzen

```
class GlobaleReferenzen {
// Globale primitive und Referenz-Variablen
   private static int i = 1;
   private int[] iarray = {1,2,3};
   private Punkt p = new Punkt(4.0, 3.0);
   private static Punkt[] parray = {
     new Punkt(1.0, 1.0), new Punkt(2.0,2.0)};
// Methodendefinitionen
   public void methode1() {
      i = 0;
      iarray[1] = 0;
   }
// Das Objekt, auf welches die Referenz zeigt, wird abgeändert
   public void methode2() {
      p.setX(0.0);
      p.setY(0.0);
      parray[1].setX(0.0);
      parray[1].setY(0.0);
   }
// Die Referenz wird abgeändert
   public void methode3() {
      p = new Punkt(-1.0,-1.0);
   }
   public static void main(String args[]) {
// Objekt der Klasse erzeugen
      GlobaleReferenzen instanz = new GlobaleReferenzen();
```

```
      System.out.println("int-Variable vor dem Aufruf von "
        + "methode1(): "+i);
      System.out.print("int-Array vor dem Aufruf von "
        + "methode1(): ");
      for(int j=0; j<instanz.iarray.length; j++) {
         System.out.print(instanz.iarray[j]+" ");
      }
      System.out.println();
      instanz.methode1();
      System.out.println("int-Variable nach dem Aufruf von "
        + "methode1(): "+i);
      System.out.print("int-Array nach dem Aufruf von "
        + "methode1(): ");
      for(int j=0; j<instanz.iarray.length; j++) {
         System.out.print(instanz.iarray[j]+" ");
      }
      System.out.println();
      System.out.print("Die Referenzvariable vom Typ Punkt zeigt"
        + " vor dem Aufruf von methode2() auf das Objekt: ");
// Die Methode anzeige() der Klasse Punkt wird aufgerufen
      instanz.p.anzeige();
      System.out.println("Punkt-Array vor dem Aufruf von "
        + "methode2(): ");
      for(int j=0; j<parray.length; j++) {
        parray[j].anzeige();
      }
      instanz.methode2();
      System.out.print("Die Referenzvariable vom Typ Punkt zeigt"
        +" nach dem Aufruf von methode2() auf das gleiche Objekt"
          + " mit abgeaenderten Werten: ");
      instanz.p.anzeige();
      System.out.println("Punkt-Array nach dem Aufruf von "
        + "methode2(): ");
      for(int j=0; j<parray.length; j++) {
        parray[j].anzeige();
      }
      System.out.print("Die Referenzvariable vom Typ Punkt zeigt"
        + " vor dem Aufruf von methode3() auf das Objekt: ");
      instanz.p.anzeige();
      instanz.methode3();
      System.out.print("Die Referenzvariable vom Typ Punkt zeigt"
        + " nach dem Aufruf von methode3() auf ein neues "
          + "Objekt: ");
      instanz.p.anzeige();
   }
}
```

Programmausgaben

```
int-Variable vor dem Aufruf von methode1(): 1
int-Array vor dem Aufruf von methode1(): 1 2 3
int-Variable  nach dem Aufruf von methode1(): 0
int-Array nach dem Aufruf von methode1(): 1 0 3
Die Referenzvariable vom Typ Punkt zeigt vor dem Aufruf von methode2() auf
das Objekt: (4.0,3.0)
Punkt-Array vor dem Aufruf von methode2():
(1.0,1.0)
(2.0,2.0)
x-Wert gesetzt
y-Wert gesetzt
x-Wert gesetzt
y-Wert gesetzt
Die Referenzvariable vom Typ Punkt zeigt nach dem Aufruf von methode2()
auf das gleiche Objekt mit abgeaenderten Werten: (0.0,0.0)
Punkt-Array nach dem Aufruf von methode2():
(1.0,1.0)
(0.0,0.0)
Die Referenzvariable vom Typ Punkt zeigt vor dem Aufruf von methode3() auf
das Objekt: (0.0,0.0)
Die Referenzvariable vom Typ Punkt zeigt nach dem Aufruf von methode3()
auf ein neues Objekt:
(-1.0,-1.0)
```

Hinweise zu den Programmausgaben

Auch wenn Felddeklarationen kein Bestandteil von Methoden sind, können Wertezuweisungen für Felder auch innerhalb von Methoden erfolgen. So zeigt die in `methode3()` abgeänderte globale Referenzvariable vom Typ `Punkt` nach dem Aufruf der Methode auf ein anderes Objekt als vor dem Aufruf der Methode, d.h. ihr Wert hat sich geändert. Darum sollten Felder, deren Werte von einer eigenen Methode der Klasse abgeändert werden können, immer als `private` deklariert werden, um nur über Zugriffsmethoden erreichbar zu sein.

Lösung 1.10

Die Klasse GanzeZahlen

```
class GanzeZahlen {
   private int z;
// Konstruktordefinition
   public GanzeZahlen(int z) {
      this.z = z;
   }
   public int getZahl() {
      return this.z;
   }
```

```
  public void setZahl(int a) {
    this.z = a;
  }
  public GanzeZahlen negativ() {
    this.z = -this.z;
    return this;
  }
  public boolean gleich(GanzeZahlen a) {
    if(this.z == a.z)
      return true;
    return false;
  }
  public boolean kleiner(GanzeZahlen a) {
    if(this.z < a.z)
      return true;
    return false;
  }
  public static GanzeZahlen ggTeiler(GanzeZahlen a,
                                     GanzeZahlen b) {
// Mit den nachfolgenen Zuweisungen würde das Vorzeichen der
// Objekte, deren Referenz beim Methodenaufruf übergeben wird,
// abgeändert
   //  GanzeZahlen r = a;
   //  GanzeZahlen s = b;
// Um die ursprünglichen Objekte nicht zu verändern, werden
// Kopien von diesen erzeugt
    GanzeZahlen r = new GanzeZahlen(a.z);
    GanzeZahlen s = new GanzeZahlen(b.z);
    if(r.getZahl()<0)
      r.negativ();
    if(s.getZahl()<0)
      s.negativ();
    if(r.gleich(new GanzeZahlen(0)))
      return s;
    else if(s.gleich(new GanzeZahlen(0)))
      return r;
    else {
      while(!r.gleich(s)) {
        if(r.kleiner(s))
          s = s.subtr(r);
        else
          r = r.subtr(s);
      }
    }
    return r;
  }
  public static GanzeZahlen kgVielfaches(GanzeZahlen a,
                                         GanzeZahlen b) {
```

```
    GanzeZahlen r = new GanzeZahlen(a.z);
    GanzeZahlen s = new GanzeZahlen(b.z);
    if(r.getZahl() < 0)
      r.negativ();
    if(s.getZahl() < 0)
      s.negativ();
    GanzeZahlen t = ggTeiler(r,s);
    GanzeZahlen v = r.multipl(s);
    GanzeZahlen u = divid(v,t);
    return u;
  }
  public void anzeige() {
    System.out.println("Ganze Zahl: "+this.z);
  }
  public static GanzeZahlen add(GanzeZahlen a, GanzeZahlen b) {
    return new GanzeZahlen(a.z+b.z);
  }
  public GanzeZahlen subtr(GanzeZahlen a) {
    return new GanzeZahlen(this.z - a.z);
  }
  public static GanzeZahlen divid(GanzeZahlen a, GanzeZahlen b){
    return new GanzeZahlen(a.z/b.z);
  }
  public GanzeZahlen multipl(GanzeZahlen a) {
    return new GanzeZahlen(z*a.z);
  }
}
```

Die Klasse RationaleZahlen

```
class RationaleZahlen {
// Globale Referenzen vom Typ der Klasse GanzeZahlen
  private GanzeZahlen x, y;
// Konstruktordefinition
  public RationaleZahlen(GanzeZahlen zaehler,
                         GanzeZahlen nenner) {
    this.x = zaehler;
    this.y = nenner;
  }
  public GanzeZahlen getZaehler() {
    return x;
  }
  public GanzeZahlen getNenner() {
    return y;
  }
  public void setZaehler(GanzeZahlen a) {
    x = a;
  }
```

```
   public void setNenner(GanzeZahlen b) {
      y = b;
   }
   public void anzeige() {
      System.out.println("Rationale Zahl: "+ x.getZahl() +
        "/" + y.getZahl());
   }
// Da beim Kürzen der Erhalt des ursprünglichen Objektes nicht
// erforderlich ist, werden die Änderungen im aktuellen Objekt
// durchgeführt und dieses wird auch zurückgegeben
   public RationaleZahlen kuerzen() {
// Lokale Referenz vom Typ der Klasse GanzeZahlen
     GanzeZahlen t = GanzeZahlen.ggTeiler(x, y);
     x = GanzeZahlen.divid(x,t);
     y = GanzeZahlen.divid(y,t);
     return this;
   }
   public static RationaleZahlen add(RationaleZahlen a,
                                     RationaleZahlen b) {
     GanzeZahlen gNenner = GanzeZahlen.kgVielfaches(a.y, b.y);
     return  new RationaleZahlen (GanzeZahlen.add(
       GanzeZahlen.divid(gNenner,a.y).multipl(a.x),
         GanzeZahlen.divid(gNenner,b.y).multipl(b.x)),gNenner);
   }
   public RationaleZahlen subtr(RationaleZahlen a) {
     GanzeZahlen gNenner = GanzeZahlen.kgVielfaches(this.y, a.y);
     return new RationaleZahlen(((GanzeZahlen.divid(
       gNenner,this.y)). multipl(this.x)). subtr((
         GanzeZahlen.divid(gNenner,a.y)).multipl(a.x)), gNenner);
   }
   public static RationaleZahlen divid(RationaleZahlen a,
                                       RationaleZahlen b) {
     return new RationaleZahlen((a.x).multipl(b.y),
       (a.y).multipl(b.x));
   }
   public RationaleZahlen multipl(RationaleZahlen a) {
      return new RationaleZahlen(x.multipl(a.x), y.multipl(a.y));
   }
}
```

Die Klasse ZahlenTest

```
class ZahlenTest {
// Konstantendefinitionen zur Beschreibung von Objektzuständen
   static final int GANZEZAHLEN = 1;
   static final int RATIONALEZAHLEN = 2;
// Der Typ der Zahlen wird zur Vereinfachung von deren Auswahl
// als int-Wert definiert
   int zahlenTyp;
```

```
// Globale Referenzen vom Typ der Klasse GanzeZahlen
   GanzeZahlen z1;
   GanzeZahlen z2;
// Konstruktordefinition
   ZahlenTest(int zahlenTyp,int z1, int z2) {
      this.z1 = new GanzeZahlen(z1);
      this.z2 = new GanzeZahlen(z2);
      this.zahlenTyp = zahlenTyp;
      switch(zahlenTyp) {
         case GANZEZAHLEN:
            defGanzeZahlen();
            break;
         case RATIONALEZAHLEN:
            defRationaleZahlen();
             break;
      }
   }
   public void defGanzeZahlen() {
      z1.anzeige();
      z2.anzeige();
      GanzeZahlen t = GanzeZahlen.ggTeiler(z1, z2);
      System.out.print("Groesste gemeinsame Teiler: ");
      t.anzeige();
      GanzeZahlen v = GanzeZahlen.kgVielfaches(z1, z2);
      System.out.print("Kleinste gemeinsame Vielfache: ");
      v.anzeige();
      System.out.print("Ergebnis der Addition: ");
      GanzeZahlen.add(z1, z2).anzeige();
      System.out.print("Ergebnis der Subtraktion: ");
// Ketten von Methoden
      z1.subtr(z2).anzeige();
      System.out.print("Ergebnis der Division: ");
      GanzeZahlen.divid(z1, z2).anzeige();
      System.out.print("Ergebnis der Multiplikation: ");
      z1.multipl(z2).anzeige();
   }
   public void defRationaleZahlen() {
      RationaleZahlen r1 = new RationaleZahlen(
        new GanzeZahlen(-1), new GanzeZahlen(-5));
      r1.anzeige();
      RationaleZahlen r2 = new RationaleZahlen(z1,z2);
      r2.anzeige();
      System.out.print("Kuerzung von rationalen Zahlen: ");
      r1.kuerzen().anzeige();
      r2.kuerzen().anzeige();
      System.out.print("Ergebnis der Addition: ");
      RationaleZahlen.add(r1, r2).anzeige();
      System.out.print("Ergebnis der Subtraktion: ");
```

```
// Ketten von Methoden
     r1.subtr(r2).anzeige();
     System.out.print("Ergebnis der Division: ");
     RationaleZahlen.divid(r1, r2).anzeige();
     System.out.print("Ergebnis der Multiplikation: ");
     r1.multipl(r2).anzeige();
   }
// Objekte der Klasse erzeugen
   public static void main(String args[]) {
     ZahlenTest test1 = new ZahlenTest(1, 6, -9);
     ZahlenTest test2 = new ZahlenTest(2, 6, -9);
   }
}
```

Programmausgaben

```
Ganze Zahl: 6
Ganze Zahl: -9
Groesste gemeinsame Teiler: Ganze Zahl: 3
Kleinste gemeinsame Vielfache: Ganze Zahl: 18
Ergebnis der Addition: Ganze Zahl: -3
Ergebnis der Subtraktion: Ganze Zahl: 15
Ergebnis der Division: Ganze Zahl: 0
Ergebnis der Multiplikation: Ganze Zahl: -54
Rationale Zahl: -1/-5
Rationale Zahl: 6/-9
Kuerzung von rationalen Zahlen: Rationale Zahl: -1/-5
Rationale Zahl: 2/-3
Ergebnis der Addition: Rationale Zahl: -7/15
Ergebnis der Subtraktion: Rationale Zahl: 13/15
Ergebnis der Division: Rationale Zahl: 3/-10
Ergebnis der Multiplikation: Rationale Zahl: -2/15
```

Lösung 1.11

Die Klasse Buch

```
class Buch {
   private int seite;
// Eine globale Referenz vom Typ der eigenen Klasse definiert
// ein selbstreferenzierendes Feld
   private Buch naechsteSeite;
// Konstruktordefinition
   Buch(int seite) {
     setSeite(seite);
     setNaechsteSeite(null);
   }
```

```
// Zugriffsmethoden
   public void setSeite(int seite) {
      this.seite = seite;
   }
   public int getSeite() {
      return seite;
   }
   public void setNaechsteSeite(Buch naechsteSeite) {
      this.naechsteSeite = naechsteSeite;
   }
   public Buch getNaechsteSeite() {
      return naechsteSeite;
   }
// Links über selbreferenzierende Felder erzeugen
   public static void rueckwaertsBlaettern(int von, int bis) {
// Lokale Referenz vom Typ der eigenen Klasse
      Buch vorigeSeite = null;
      for(int i=von; i<=bis; i++) {
// Eine neue Instanz der Klasse, mit einem Verweis auf die vorher
// konstruierte, wird erzeugt
         Buch aktuelleSeite = new Buch(i);
         aktuelleSeite.setNaechsteSeite(vorigeSeite);
// Bei einer Zuweisung von Referenzvariablen, wird kein neues
// Objekt erzeugt, die Referenzvariable vorigeSeite zeigt auf das
// gleiche Objekt wie die Referenzvariable aktuelleSeite
         vorigeSeite = aktuelleSeite;
      }
      Buch aktuelleSeite = vorigeSeite;
      while(aktuelleSeite != null){
      System.out.print("Seite "+ aktuelleSeite.getSeite() + " ");
         aktuelleSeite = aktuelleSeite.getNaechsteSeite();
      }
      System.out.println();
   }
   public static void vorwaertsBlaettern(int von, int bis) {
// Lokale Referenz vom Typ der eigenen Klasse
      Buch vorigeSeite = new Buch(bis);
      for(int i=bis-1; i>=von; i--) {
// Eine neue Instanz der Klasse, mit einem Verweis auf die vorher
// konstruierte, wird erzeugt
         Buch aktuelleSeite = new Buch(i);
         aktuelleSeite.setNaechsteSeite(vorigeSeite);
// Zuweisung von Referenzvariablen
         vorigeSeite = aktuelleSeite;
      }
      Buch aktuelleSeite = vorigeSeite;
      while(aktuelleSeite != null) {
```

```
      System.out.print("Seite "+ aktuelleSeite.getSeite() + " ");
        aktuelleSeite = aktuelleSeite.getNaechsteSeite();
      }
      System.out.println();
   }
}
```

Die Klasse BuchTest

```
class BuchTest {
   public static void main(String args[]) {
      Buch.vorwaertsBlaettern(1,4);
      Buch.rueckwaertsBlaettern(1,4);
      Buch.vorwaertsBlaettern(10,13);
      Buch.rueckwaertsBlaettern(10,13);
      Buch.vorwaertsBlaettern(100,102);
      Buch.rueckwaertsBlaettern(100,102);
   }
}
```

Programmausgaben

```
Seite 1 Seite 2 Seite 3 Seite 4
Seite 4 Seite 3 Seite 2 Seite 1
Seite 10 Seite 11 Seite 12 Seite 13
Seite 13 Seite 12 Seite 11 Seite 10
Seite 100 Seite 101 Seite 102
Seite 102 Seite 101 Seite 100
```

Lösung 1.12

Die Klasse PackageTest

```
package paket1;
class PackageTest {
   public static void main(String args[]) {
      System.out.println("Test der package-Anweisung");
   }
}
```

Programmausgaben

```
Test der package-Anweisung
```

Lösung 1.13

Die Klasse Klasse

```
package paket2;
public class Klasse {
```

```
   public Klasse() {
      System.out.println(
        "Definition einer Klasse im Verzeichnis paket2");
   }
}
```

Die Klasse KlassenTest

```
// import paket2.Klasse;
class KlassenTest {
   public static void main(String args[]) {
      paket2.Klasse kls = new paket2.Klasse();
// Wird die import-Anweisung genutzt, kann auf den Klassennamen
// direkt zugegriffen werden
   //   Klasse kls = new Klasse();
   }
}
```

Programmausgaben

```
Definition einer Klasse im Verzeichnis paket2
```

Lösung 1.14

Die Klasse Klasse1

```
package paket1;
// Die Klasse muss als public definiert werden, weil sie aus
// einem externen Paket angesprochen wird
public class Klasse1 {
   private static int privatesFeld = 1;
   protected static int geschuetztesFeld = 2;
   public static int oeffentlichesFeld = 3;
   static int feld = 4;
// Konstruktordefinition
   public Klasse1() {
      System.out.println("Instanz der Klasse1");
   }
}
```

Die Klasse Klasse2

```
package paket1.paket2;
public class Klasse2 {
// Konstruktordefinition
   public Klasse2() {
      System.out.println("Instanz der Klasse2");
   }
}
```

Die Klasse PackageTest1

```
import paket1.Klasse1;
//import paket2.Klasse2; // Fehler
import paket1.paket2.Klasse2;
class PackageTest1 {
   public static void main(String args[]) {
      Klasse1 kls1 = new Klasse1();
      Klasse2 kls2 = new Klasse2();
// Compilerfehler:
// Das Feld aus paket1.Klasse1 verfügt über privaten Zugriff
   //   System.out.println(Klasse1.privatesFeld);
// Das Feld aus paket1.Klasse1 verfügt über geschuetzten Zugriff
   //   System.out.println(Klasse1.geschuetztesFeld);
// Das Feld feld aus paket1.Klasse1 besitzt keinen Modifikator
// und darauf kann nicht von außerhalb des Paketes zugegriffen
// werden
   //   System.out.println(Klasse1.feld);
// Zugriff erlaubt, weil uneingeschränkt öffentlich durch die
// Definition mit public
      System.out.println(Klasse1.oeffentlichesFeld);
   }
}
```

Programmausgaben

```
Instanz der Klasse1
Instanz der Klasse2
3
```

Lösung 1.15

Die Klasse KongruenteDreiecke

```
public class KongruenteDreiecke {
   private double x;
   private double y;
   private double z;
   private static final double PI = 3.14159;
 // Konstruktordefinition
   public KongruenteDreiecke(double x, double y, double z) {
      this.x = x;
      this.y = y;
      this.z = z;
   }
// Zugriffsmethoden
   public double getX() {
      return x;
   }
```

```
public double getY() {
   return y;
}
public double getZ() {
   return z;
}
public double setX(double a) {
   return x = a;
}
public double setY(double b) {
   return y = b;
}
public double setZ(double c) {
   return z  = c;
}
public static double winkelRadian (double winkelGrad) {
   return winkelGrad*PI/180;
}
public static double winkelGrad (double winkelRadian) {
   return winkelRadian*180/PI;
}
public KongruenteDreiecke sss() {
   double w1 = Math.round(winkelGrad(Math.acos((x*x-y*y-z*z)/
     (-2*y*z))));
   double w2 = Math.round(winkelGrad(Math.acos((y*y-x*x-z*z)/
     (-2*x*z))));
   double w3 = Math.round(winkelGrad(Math.acos((z*z-x*x-y*y)/
     (-2*x*y))));
   return new KongruenteDreiecke(w1,w2,w3);
}
public KongruenteDreiecke sws() {
   double s = Math.round(Math.sqrt(x*x+z*z-2*x*z*Math.cos(
     winkelRadian(y))));
   double w1 = Math.round(winkelGrad(Math.acos((x*x-z*z-s*s)/
     (-2*z*s))));
   double w2 = Math.round(winkelGrad(Math.acos((z*z-x*x-s*s)/
     (-2*x*s))));
   return new KongruenteDreiecke(w1,s,w2);
}
public KongruenteDreiecke wsw() {
   double w = 180-x-z;
   double s1 = Math.round(y*Math.sin(winkelRadian(x))/
     Math.sin(winkelRadian(w)));
   double s2 = Math.round(y*Math.sin(winkelRadian(z))/
     Math.sin(winkelRadian(w)));
   return new KongruenteDreiecke(s1,w,s2);
}
```

```
   public KongruenteDreiecke ssw() {
      double w1 = Math.round(winkelGrad(Math.asin(
        x*Math.sin(winkelRadian(z))/y)));
      double w2 = 180-z-w1;
      double s = Math.round(y*Math.sin(winkelRadian(w2))/
        Math.sin(winkelRadian(z)));
      return new KongruenteDreiecke(w1,w2,s);
   }
}
```

Die Klasse KongruenteDreieckeTest

```
public class KongruenteDreieckeTest {
   public static void main(String args[]) {
      KongruenteDreiecke d1 = new KongruenteDreiecke(4,4,4);
      d1 = d1.sss();
      System.out.println("w*w*w = "+ d1.getX()+"*"+d1.getY()+
        "*"+d1.getZ());
      KongruenteDreiecke d2 = new KongruenteDreiecke(4,60,4);
      d2 = d2.sws();
      System.out.println("w*s*w = "+ d2.getX()+"*"+d2.getY()+
        "*"+d2.getZ());
      KongruenteDreiecke d3 = new KongruenteDreiecke(60,4,60);
      d3 = d3.wsw();
      System.out.println("s*w*s = "+ d3.getX()+"*"+d3.getY()+
        "*"+d3.getZ());
      KongruenteDreiecke d4 = new KongruenteDreiecke(4,4,60);
      d4 = d4.ssw();
      System.out.println("w*w*s = "+ d4.getX()+"*"+d4.getY()+
        "*"+d4.getZ());
   }
}
```

Programmausgaben

```
w*w*w = 60.0*60.0*60.0
w*s*w = 60.0*4.0*60.0
s*w*s = 4.0*60.0*4.0
w*w*s = 60.0*60.0*4.0
```

Lösung 1.16

Die Klasse Strecke

```
public class Strecke {
   private Punkt P;
   private Punkt Q;
// Konstruktordefinition
```

```
   public Strecke(Punkt P, Punkt Q) {
      this.P = P;
      this.Q = Q;
   }
// Zugriffsmethoden
   public void setzeP(Punkt A) {
      P = A;
   }
   public void setzeQ(Punkt B) {
      Q = B;
   }
   public Punkt getP() {
      return P;
   }
   public Punkt getQ() {
      return Q;
   }
   public double distanz() {
      double xP = P.getX();
      double yP = P.getY();
      double xQ = Q.getX();
      double yQ = Q.getY();
      return Math.round(Math.sqrt(Math.pow(xQ-xP,2)+
        Math.pow(yQ-yP,2)));
   }
   public void geradenGleichung() {
      double xP = P.getX();
      double yP = P.getY();
      double xQ = Q.getX();
      double yQ = Q.getY();
      System.out.println("(y - "+yP+") = (("+yQ+" - "+yP+
        ")/("+xQ+" - "+xP+"))*(x - "+xP+")");
   }
}
```

Die Klasse Kreis

```
public class Kreis {
   private static final double PI = Math.PI;
   private Strecke PQ;
// Konstruktordefinition
   public Kreis(Strecke PQ) {
      this.PQ = PQ;
   }
   public double flaeche() {
      return Math.round (PI*PQ.distanz()*PQ.distanz());
   }
   public double umfang() {
```

```
    return Math.round(2*PI*PQ.distanz());
  }
  public void kreisGleichung() {
    Punkt P = PQ.getP();
    double xP = P.getX();
    double yP = P.getY();
    System.out.println("(x - " +xP+")**2 + "+ "(y - " +yP+
      ")**2 "+"- "+PQ.distanz()*PQ.distanz()+" = 0");
  }
}
```

Die Klasse PunktStreckeKtreisTest

```
public class PunktStreckeKreisTest {
  public static void main(String args[]) {
// Erzeugen und Anzeigen von zwei Punkt-Objekten
    Punkt P = new Punkt(4.0, 3.0);
    Punkt Q = new Punkt(1.0, 2.0);
    System.out.print("Punkt P: ");
    P.anzeige();
    System.out.print("PunktQ : ");
    Q.anzeige();
// Berechnung der Länge der Strecke PQ und Anzeige der
// Geradengleichung durch diese Punkte
    Strecke PQ = new Strecke(P,Q);
    System.out.println("Laenge der Strecke PQ = " +
      PQ.distanz());
    System.out.print("Geradengleichung: ");
    PQ.geradenGleichung();
// Berechnung von Flaeche und Umfang des Kreises und Anzeige der
// Kreisgleichung
    Kreis k = new Kreis(PQ);
    System.out.println("Flaeche des Kreises mit dem " +
      "Mittelpunkt P" + " und PQ als Radius = " + k.flaeche());
    System.out.println("Umfang des Kreises mit dem " +
      "Mittelpunkt P" + " und PQ als Radius = " + k.umfang());
    System.out.print("Kreisgleichung: ");
    k.kreisGleichung();
  }
}
```

Programmausgaben

```
PunktP: (4.0,3.0)
PunktQ: (1.0,2.0)
Laenge der Strecke PQ  =  3.0
Geradengleichung: (y - 3.0) = ((2.0 - 3.0)/(1.0 - 4.0))*(x - 4.0)
Fläche des Kreises mit dem Mittlpunkt P und PQ als Radius = 28.0
```

```
Umfang des Kreises mit mit dem Mittlpunkt P und PQ als Radius = 19.0
Kreisgleichung: (x - 4.0)**2 + (y - 3.0)**2 - 9.0 = 0
```

Lösung 1.17

Die Klasse WrapperKlassen

```
public class WrapperKlassen {
// Globale Referenzen vom Typ der Hüllenklassen
   private Byte einByte;
   private Character einCharacter;
   private Integer einInteger;
   private Short einShort;
   private Long einLong;
   private Float einFloat;
   private Double einDouble;
   private Boolean einBoolean;
// Konstruktordefinition
   WrapperKlassen(byte by, int i, char c, short s, long l,
                  float f, double d, boolean bo) {
// Objekte der Hüllenklassen erzeugen
      einByte = new Byte(by);
      einCharacter = new Character(c);
      einInteger = new Integer(i);
      einShort = new Short(s);
      einLong = new Long(l);
      einFloat = new Float(f);
      einDouble = new Double(d);
      einBoolean = new Boolean(bo);
   }
// Umwandlungen in Zahlensysteme mit einer anderen Basis,
// am Beispiel von int-Werten
   public void konvert(int i) {
      System.out.println("Der Binaerwert von " + i + " = "+
             Integer.toBinaryString(i));
      System.out.println("Der Octalwert von " + i + " = "+
             Integer.toOctalString(i));
      System.out.println("Der Hexadecimalwert von " + i + " = "+
             Integer.toHexString(i));
   }
// Beispiele von Typumwandlungen mit Methoden von Wrapper-Klassen
   public int charToUnicodeint(char c) {
// Der Unicode-Wert des Zeichens wird der Variablen i
// zugewiesen
      int i  =  Character.getNumericValue(c);
      return i;
   }
   public char intTochar(int i) {
```

```
      char c = Character.forDigit(i,10);
      return c;
   }
   public int charToint(char c) {
      int i  =  Character.digit(c,10);
      return i;
   }
// Liefern des Basistyp-Wertes eines Wrapper-Objektes
   public void objectTonumber() {
      System.out.print(" "+einByte.byteValue());
      System.out.print(" "+einInteger.intValue());
      System.out.print(" "+einLong.longValue());
      System.out.print(" "+einShort.shortValue());
      System.out.print(" "+einFloat.floatValue();
      System.out.print(" "+einDouble.doubleValue());
      System.out.print(" "+einCharacter.charValue());
      System.out.print(" "+einBoolean.booleanValue());
      System.out.println();
   }
// Wrapper-Klassen exportieren auch eine Menge von nützlichen
// Konstanten, wie z. B. Wertebereiche für primitive Datentypen
   public void werteBereiche() {
      System.out.println("Der Datentyp byte nimmt Werte von "+
        Byte.MIN_VALUE+ " bis "+Byte.MAX_VALUE);
      System.out.println("Der Datentyp int nimmt Werte von "+
        Integer.MIN_VALUE+ " bis "+Integer.MAX_VALUE);
   }
// Einen Beitrag zum sicheren Rechnen mit Wrapper-Klassen liefern
// deren unendliche Werte
   public void unendlicheWerte() {
      double d1 = Double.MAX_VALUE;
      double dMax = 2*d1;
      double dMin = -1.0/0.0;
      double d = 0.0/0.0;
      System.out.println(dMax+" * "+dMin+" * "+d);
   }
}
```

Die Klasse WrapperKlassenTest

```
public class WrapperKlassenTest {
   public static void main(String args[]) {
      WrapperKlassen wrapper = new WrapperKlassen((byte)1, 2,'3',
        (short)4,(long)5,(float)1.0,2.0,true);
      System.out.print("Basistypwerte der Wrapper-Objekte: ");
      wrapper.objectTonumber();
      System.out.println(
       "Umwandlungen in Zahlensysteme mit einer anderen Basis:");
```

```
    wrapper.konvert(100);
    System.out.print("Typumwandlungen: ");
    System.out.print(" "+wrapper.charToUnicodeint('3'));
    System.out.print(" "+wrapper.charToint('3'));
    System.out.print(" "+wrapper.intTochar(3));
    System.out.println();
    wrapper.werteBereiche();
    wrapper.unendlicheWerte();
  }
}
```

Programmausgaben

```
Basistypwerte der Wrapper-Objekte:  1 2 5 4 1.0 2.0 3 true
Umwandlungen in Zahlensysteme mit einer anderen Basis:
Der Binaerwert von 100 = 1100100
Der Oktalwert von 100 = 144
Der Hexadecimalwert von 100 = 64
Typumwandlungen:  3 3 3
Der Datentyp byte nimmt Werte von – 128 bis 127
Der Datentyp int nimmt Werte von -2147483648 bis 2147483647
Infinity * -Infinity * NaN
```

Hinweise zum Lösungsvorschlag

Die als Lösung hinzugefügte Klasse ruft die Methoden von Hüllenklassen zu Typumwandlungen, Wechseln der Basis von Zahlensystemen und Ausgaben der Wertebereiche von primitiven Datentypen auf. Typumwandlungen beziehen sich auf das Verändern des Datentyps eines bestimmten Wertes. Java verhindert implizite Typumwandlungen, wenn eine fehlende Kompatibilität bei Zuweisungen zu einem Verlust von Daten führen könnte. Dann kann nur eine explizite Typumwandlung erfolgen oder man kann auf die entsprechenden Methoden der Wrapper-Klassen zurückgreifen. Beim Rechnen mit Bruchtypen (`float`, `double`) gibt es im Gegensatz zu den Ganzzahlentypen (`short`, `long`, `byte`) drei spezielle Werte »keine Zahl« (not a number, abgekürtzt `NaN`), »plus unendlich« (`POSITIVE_INFINITY`) und »minus unendlich« (`NEGATIVE_INFINITY`).

Lösung 1.18

Die Klasse WrapperKlassenmitAutoBoxing

```
public class WrapperKlassenmitAutoBoxing {
// Globale Referenzen vom Typ der Hüllenklassen
   private Byte einByte;
   private Character einCharacter;
   private Integer einInteger;
   private Short einShort;
   private Long einLong;
   private Float einFloat;
```

```
   private Double einDouble;
   Boolean einBoolean;
// Konstruktordefinition
   WrapperKlassenmitAutoBoxing(byte by, int i, char c, short s,
                    long l, float f, double d, boolean bo) {
// Instanzen von Hüllenklassen erzeugen
     einByte = by;
     einCharacter = c;
     einInteger = i;
     einShort = s;
     einLong = l;
     einFloat = f;
     einDouble = d;
     einBoolean = bo;
     }
// Lieferung des Basistyp-Wertes eines Wrapper-Objektes über
// Unboxing
   public void objectTonumber() {
     System.out.print(" "+einByte);
     System.out.print(" "+einInteger);
     System.out.print(" "+einLong);
     System.out.print(" "+einShort);
     System.out.print(" "+einFloat);
     System.out.print(" "+einDouble);
     System.out.print(" "+einCharacter);
     System.out.print(" "+einBoolean);
     System.out.println();
   }
// Das Autoboxing macht möglich, das der Methodenaufruf über
// konvert(boolean) erfolgen kann
   public boolean konvert(Boolean b) {
// Unboxing vom Wrapper-Typ Boolean zum primitiven Typen boolean
     return b;
   }
// Beispiele von Typumwandlungen in einem Ausdruck
   public void rechnen(float zahl1, double zahl2) {
// Die Werte von primitiven Detentypen werden eingehüllt,
     Float zahlFloat = zahl1;
     Double zahlDouble = zahl2;
// zum Rechnen entkapselt und als Ergebnis wieder eingehüllt
     Double ergebnis = zahlFloat*zahlDouble+zahlFloat/
       zahlDouble;
     System.out.println("Wert vor Inkrementieren: "+
       ergebnis);
// Das Wrapper-Objekt ergebnis wird entkapselt, der gewonnene
// Wert inkrementiert und wieder eingehüllt
     ergebnis++;
```

```
     System.out.println("Wert nach Inkrementieren: "+ergebnis);
     System.out.print(
       "Ausdruck vom Typ Integer in einer for-Schleife ");
     for(Integer i=0; i<=2; i++)
        System.out.print(i+" ");
     System.out.println();
   }
// Unboxing in Kontrollanweisungen
   public void vergleichen(Integer zahl1Integer,
                           Integer zahl2Integer) {
// Unboxing von Wrapper-Objekten
     int zahl1 = zahl1Integer;
     int zahl2 = zahl2Integer;
// Vergleich von Referenzen
     if(zahl1Integer == zahl2Integer){
        System.out.print("Gleiche Referenzen");
// Vergleich von primitiven Datentypen
        if(zahl1 == zahl2)
          System.out.println(" und gleiche Werte");
        else
          System.out.println(" und verschiedene Werte");
     }
     else {
        System.out.print("Verschiedene Referenzen");
        if(zahl1 == zahl2)
          System.out.println(" und gleiche Werte");
        else
          System.out.println(" und verschiedene Werte");
     }
   }
}
```

Die Klasse WrapperKlassenmitAutoBoxingTest

```
public class WrapperKlassenmitAutoBoxingTest {
   public static void main(String args[]) {
// Objekt der Klasse erzeugen
     WrapperKlassenmitAutoBoxing wrapper =
       new WrapperKlassenmitAutoBoxing
         ((byte)1, 2,'3',(short)4,(long)5,(float)1.0,2.0,true);
     System.out.print("Basistypwerte der Wrapper-Objekte:");
// und ihre Methoden aufrufen
     wrapper.objectTonumber();
     System.out.print("Auto(un)boxing in Methoden-aufrufen und "
       + "-rueckgaben: ");
     System.out.println(wrapper.konvert(true));
     System.out.println("Typumwandlungen in einem Ausdruck:");
     wrapper.rechnen(1, 2.0);
```

```
    System.out.println("Unboxing in Kontrollanweisungen:");
    wrapper.vergleichen(1, 2);
    wrapper.vergleichen(new Integer(1), new Integer(1));
    Integer zahl1 = new Integer(1);
    Integer zahl2 = zahl1;
    wrapper.vergleichen(zahl1,zahl2);
  }
}
```

Programmausgaben

```
Basistypwerte der Wrapper-Objekte: 1 2 5 4 1.0 2.0 3 true
Auto(un)boxing in Methoden-aufrufen und -rueckgaben: true
Typumwandlungen in einem Ausdruck:
Wert vor Inkrementieren: 2.5
Wert nach Inkrementieren: 3.5
Ausdruck vom Typ Integer in einer for-Schleife: 0 1 2
Unboxing in Kontrollanweisungen:
Verschiedene Referenzen und verschiedene Werte
Verschiedene Referenzen und gleiche Werte
Gleiche Referenzen und gleiche Werte
```

Lösung 1.19

Die Klasse ArrayTest1

```
import java.lang.reflect.Array;
import java.util.Arrays;
public class ArrayTest1 {
// Ein Array-Objekt muss mit dem new-Operator über die Angabe
// einer festen Grösse erzeugt werden
  private static int[]a = new int[3];
  private static int[][] x = new int [2][3];
// Alternative Deklarationen
  // private int a[] = new int[3];
  // private static [][] int x = new int [2][3];
  // private static int [] x [] = new int [2][3];
// Klassenmethode für die Anzeige eines zweidimensionalen Arrays
  public static void anzeige(int[][] x) {
    for(int i=0; i<2; i++) {
      for(int j=0; j<3; j++)
        System.out.print(x[i][j]+" ");
// Zeilenumbruch ausgeben
      System.out.println();
    }
    System.out.println();
  }
  public static void main(String args[]) {
// Eindimensionale Arrays initialisieren
```

```
// Arrayelemente können auch einzeln initialisiert werden
     int []y = new int[3];
     for(int i=0; i<3; i++)
       y[i]= 1;
// Eine kompakte Initialisierung für Arrayelemente wird vom
// Compiler in eine Initialisierung von einzelnen Elementen
// umgeformt und erst zur Laufzeit durchgeführt
     int[] z = {2,2,2};
// Initialisierungen mit einem konstanten Wert über die Methode
// fill() der Klasse Arrays
     Arrays.fill(a,7);
     System.out.println(
      "Eindimensionales Array von primitiven Datentypen");
// Mit der Methode toString() der Klasse Arrays das Array
// anzeigen
     System.out.println(Arrays.toString(a));
// Methoden der Klasse Array aufrufen
     Array.setInt(a, 0, 6);
     Array.setInt(a, 1, 6);
     Array.setInt(a, 2, 6);
     Integer integer = (Integer)Array.get(a, 0);
// Den Wert des Wrapper-Objektes ermitteln
     int n = integer.intValue();
     System.out.println(a[0] + " " + a[1] +" " +a[2]+"**"+n);
// Zweidimensionale Arrays über eindimensionale Arrays erzeugen
     System.out.println("Zweidimensionale Arrays sind Arrays"
      +" von eindimensionalen Arrays");
     for(int i=0; i<2; i++) {
       x[i] = new int [] {9, 9, 9};
// Mit der Methode toString() der Klasse Arrays die
// eindimsionalen Arrays anzeigen
        System.out.print(Arrays.toString(x[i]));
        System.out.println();
     }
     System.out.println("Zweidimensionale Arrays von primitiven"
      + " Datentypen");
// Das so erzeugte zweidimensionale Array ausgeben
     anzeige(x);
// Inititialisierungen von zweidimensionalen Arrays
     x = new int[][]{{1, 1, 1},{2,2,2}};
     anzeige(x);
     x = new int[][]{y, z};
     anzeige(x);
// Dynamisches Erzeugen eines Array mit der Methode
// newInstance() der Klasse Array
     int c[] = {2};
     Object array = Array.newInstance(int.class,c);
```

```
     Array.setInt(array, 0, 4);
     Array.setInt(array, 1, 4);
     System.out.println("Eindimensionales Array von primitiven"
       + " Datentypen");
     System.out.println(Array.getInt(array,0) + " "+
       Array.getInt(array,1));
// Elemente von Referenzarrays müssen immer einzeln instanziert
// werden
     Vektor [] v = new Vektor[2];
     Vektor [][] w = new Vektor[2][2];
     System.out.println("Eindimensionales Array vom Typ der "
       + "Klasse Vektor");
     for(int i=0; i<2; i++) {
        v[i] = new Vektor(1,2,3);
     }
// An den Vektor-Objekten wird die Methode anzeige() der Vektor-
// Klasse aufgerufen
     v[0].anzeige();
     v[1].anzeige();
// Zeilenumbruch definieren
     System.out.println();
     System.out.println("Zweidimensionales Array vom Typ der "
       + "Klasse Vektor");
     for(int i=0; i<2; i++) {
        w[i][0] = new Vektor(1,2,3);
        w[i][1] = new Vektor(4,5,6);
     }
     for(int i=0; i<2; i++) {
        w[i][0].anzeige();
        w[i][1].anzeige();
     }
   }
}
```

Programmausgaben

```
Eindimensionales Array von primitiven Datentypen
[7 7 7]
6 6 6**6
Zweidimensionale Arrays sind Arrays von eindimensionalen Arrays
[9 9 9]
[9 9 9]
Zweidimensionale Arrays von primitiven Datentypen
9 9 9
9 9 9

1 1 1
2 2 2
```

```
1 1 1
2 2 2
Eindimensionales Array von primitiven Datentypen
4 4
Eindimensionales Array vom Typ der Klasse Vektor
(1, 2, 3) (1, 2, 3)
Zweidimensionales Array vom Typ der Klasse Vektor
(1, 2, 3) (4, 5, 6) (1, 2, 3) (4, 5, 6)
```

Lösung 1.20

Die Klasse StringTest

```
import java.util.Arrays;
public class StringTest {
   public static void main(String args[]) {
      String s1, s2, s3;
// Zuweisung eines Literals
      s1 = "Java";
// Den Defaultwert für eine Referenzvariable zuweisen
      s2 = null;
// String-Objekt erzeugen
      s3 = new String("Java");
// String anfügen
      s1 = s1.concat(" lernen");
// String-Objekt erzeugen
      s2 = new String(s1);
// Teilstring bilden
      s3 = s2.substring(2,8);
// Die Methode valueOf()der Klasse String aufrufen
      String s4 = String.valueOf(s1);
      String s5 = String.valueOf(s2);
// Addieren von String-Objekten
      String s6 = s4 + s5;
      System.out.println(s1+" * "+s2+" * "+s3+" * "+s4+" * "+s5+
        " * "+s6);
// Vergleichen von String-Objekten
      System.out.println(vergleichen(s4,s5));
// Konvertieren von Strings in numerische Werte
      System.out.print(Integer.parseInt("123"));
      System.out.print(" "+Double.parseDouble("12.3"));
      System.out.print(" "+Short.parseShort("123"));
      System.out.println();
// Konvertieren von numerischen Werten in Strings
      System.out.print String.valueOf (123+1));
      System.out.print(" "+ String.valueOf (12.3+0.1));
      System.out.print(" "+ String.valueOf (123>0));
```

```
     System.out.println();
// Umwandlungen von char-Arrays in String-Objekte
     char[] buchstaben = {'J','A','V','A'};
// Den Konstruktor der Klasse String aufrufen
     String s = new String(buchstaben);
     char []zahlen = {'0','1','2','3','4','5','6','7','8','9'};
// Mit dem zweiten und dritten Element des char-Arrays einen
// String erzeugen
     String zwoelf = String.copyValueOf(zahlen,1,2);
// und diesen in ein neues char-Array umsetzen
     char[] zahl1 = zwoelf.toCharArray();
     System.out.println(zwoelf);
     System.out.println(Arrays.toString(zahl1));
     String tausendzweihundertvierunddreisig = String.
       copyValueOf(zahlen,1,4);
     char[] zahl2 = tausendzweihundertvierunddreisig.
       toCharArray();
     System.out.println(tausendzweihundertvierunddreisig);
     System.out.println(Arrays.toString(zahl2));
// Definition eines String-Array
     String[] sArray = new String[3];
// Initialisieren der Arrayelemente
     sArray[0] = StringTest.vergleichen("Java", "JAVA");
     sArray[1] = StringTest.vergleichen("JAVA", "JAVA");
     sArray[2] = StringTest.vergleichen("JAVA", "Java");
     System.out.println(sArray[0]+" und "+sArray[1]+" und "
       +sArray[2]);
// String-Array am Beispiel der Kommandozeilenparameter
     for(int i=0; i<args.length; i++) {
        System.out.println("Parameter "+(i+1)+": " +args[i]);
     }
   }
// Klassenmethode zum Vergleichen von String-Objekten
   public static String vergleichen(String s1, String s2) {
     if(s1.compareTo(s2) > 0)
        return s1+" > "+s2;
     else if(s1.compareTo(s2) < 0)
        return s1+" < "+s2;
     else
        return s1+" = "+s2;
   }
}
```

Programmausgaben

```
Java lernen * Java lernen * va ler * Java lernen * Java lernen * Java
lernenJava lernen
Java lernen = Java lernen
123 12.3 123
```

```
124 12.4 true
12
[1, 2]
1234
[1, 2, 3, 4]
Java > JAVA und JAVA = JAVA und JAVA < Java
```

Der Aufruf der Applikation über: java StringTest Java lernen, führt zur zusätzlichen Ausgabe:

```
Parameter 1: Java
Parameter 2: lernen
```

Lösung 1.21

Die Klasse ArrayTest2

```
public class ArrayTest2 {
// Methodendefinitionen
   public static void anzeige() {
      int[] array1 = new int[2];
      int[][] array2 = new int[2][2];
      int[][][] array3 = new int[2][2][2];
// Initialisieren der Arrayelemente
      for(int i=0; i<2; i++) {
        array1[i] = i;
        for(int j=0; j<2; j++) {
          array2[i][j] = i+j;
          for(int k=0; k<2; k++) {
            array3[i][j][k] = i+j+k;
          }
        }
      }
// Anzeige der Arrayelemente
      for(int x : array1) {
        System.out.print(x+" ");
      }
      System.out.println();
      System.out.println();
      for(int x[] : array2) {
        for(int y : x) {
          System.out.print(y+" ");
        }
        System.out.println();
      }
      System.out.println();
      for(int x[][] : array3) {
        for(int y[] : x) {
          for(int z : y) {
```

```
            System.out.print(z+" ");
          }
        System.out.println();
        }
      }
   }
// Array als Rückgabewert in Mehtoden
   public static char[] rueckgabe(char a0, char a1, char a2) {
     char[] a = {a0, a1,a2};
     return a;
   }
   public static void main(String args[]) {
     System.out.println("char-Array");
// Die Methode rueckgabe() liefert ein char-Array zurück, und
// dessen Elemente werden in einer for-each-Schleife ausgegeben
     for(char x : rueckgabe('a','b','c'))
       System.out.print(x+" ");
     System.out.println();
     System.out.println("int-Arrays");
     anzeige();
   }
}
```

Programmausgaben

```
char-Array
a b c
int-Arrays
0 1

0 1
1 2

0 1
1 2
1 2
2 3
```

Lösung 1.22

Die Klasse MethodenmitVararg

```
public class MethodenmitVararg {
// Methode mit einer variablen Anzahl von Argumenten
   public static void varArg(char ... c) {
     System.out.println("Anzahl der variablen Argumente: "
       + c.length);
```

```
    System.out.print("Werte der variablen Argumente: ");
    for(char x : c)
      System.out.print(x+" ");
    System.out.println();
  }
// Überladen der Methode, indem der Typ der Parameter
// abgeändert wird
  public static void varArg(boolean ... c) {
    System.out.println("Anzahl der variablen Argumente: "
     + c.length);
    System.out.print("Werte der variablen Argumente: ");
    for(boolean x : c)
      System.out.print(x+" ");
    System.out.println();
  }
// Überladen der Methode, indem die Parameterliste erweitert wird
  public static void varArg(int i1, int i2, boolean ... c) {
    System.out.println("Anzahl der variablen Argumente: "
     + c.length);
    System.out.print("Werte der variablen Argumente: ");
    for(boolean x : c)
      System.out.print(x+" ");
    System.out.println();
    System.out.println("Werte der klassischen Argumente: "
     + i1 + " " + i2);
  }
  public static void main(String args[]) {
// Methodenaufrufe mit unterschiedlicher Anzahl von Argumenten
    varArg('a');
    varArg('a','b','c');
    varArg(false);
    varArg(true, true);
    varArg(1,2,false,true);
    varArg(1,2,true);
  }
}
```

Programmausgaben

```
Anzahl der variablen Argumente: 1
Werte der variablen Argumente: a
…
Anzahl der variablen Argumente: 2
Werte der variablen Argumente: false true
Werte der klassischen Argumente: 1 2
…
```

Lösung 1.23

Die Klasse InitialisierungInstanzfelder

```
// import java.util.Arrays;
public class InitialisierungInstanzfelder {
// Instanzfeld Initialisierer
   private byte einbyte;
   private char einchar = 'A';
   private int einint ;
   private short einshort;
   private long einlong = 1;
   private float einfloat;
   private double eindouble = 0.1;
   private boolean einboolean;
   private String einString ="Java 6.0";
   private Punkt einPunkt;
   private char [] einArray = new char[26];
// Instanzblock Initialisierer
   {
     for(int i = 0; i<26; i++) {
// buchstabe ist eine lokale Variable und kein Feld der Klasse
       char buchstabe = (char)('A'+ i);
       einArray[i] = buchstabe;
     }
   }
// Selbstreferenzierendes Instanzfeld
   InitialisierungInstanzfelder selfReferentialFeld;
// Konstruktordefinition
   InitialisierungInstanzfelder(String s) {
     einPunkt = new Punkt(1,1);
     einint = 50;
     System.out.println(s);
   }
// Methode zur InitIialisierung des selbstreferenzierenden
// Instanzfeldes
   public InitialisierungInstanzfelder setselfReferentialFeld() {
     return selfReferentialFeld =
       new InitialisierungInstanzfelder(
         "Initialisierung von Instanzfelder");
   }
   public static void main(String args[]) {
// Objekt der Klasse erzeugen
     InitialisierungInstanzfelder instanz =
       new InitialisierungInstanzfelder(
         "Initialisierung von Instanzfelder");
```

```
// Initialisierung des selbstreferenzierenden Feldes
     System.out.println(instanz.setselfReferentialFeld());
// Anzeige der Werte  von primitiven Typen
     System.out.println("einbyte: "+instanz.einbyte);
     System.out.println("einint: "+instanz.einint);
     System.out.println("einshort: "+instanz.einshort);
     System.out.println("einfloat: "+instanz.einfloat);
     System.out.println("einboolean: "+instanz.einboolean);
     System.out.println("einchar: "+instanz.einchar);
     System.out.println("einlong: "+instanz.einlong);
     System.out.println("eindouble: "+instanz.eindouble);
// Anzeige der Werte von Referenztypen
     System.out.println("einString: "+instanz.einString);
     System.out.print("einPunkt: ");
// Die Methode der Klasse Punkt, an einer Instanz der Klasse
// aufrufen
     instanz.einPunkt.anzeige();
// Die Anzeige der Elemente von einArray kann nicht über den
// Methodenaufruf Arrays.toString(einArray) erfolgen, weil auf
// ein Instanzfeld, nicht aus einem statischen Kontext, direkt
// zugegriffen werden kann
   // System.out.println(Arrays.toString(einArray)); // Fehler
     System.out.print("[ ");
     for (int i = 0; i<26; i++) {
        System.out.print(instanz.einArray[i]+" ");
     }
     System.out.println("]");
   }
}
```

Programmausgaben

```
Initialisierung von Instanzfelder
Initialisierung von Instanzfelder
InitialisierungInstanzfelder…
einbyte: 0
einint: 50
einshort:  0
einfloat: 0.0
einboolean: false
einchar: A
einlong: 1
eindouble:  0.1
einString: Java 6.0
inPunkt: (1.0,1.0)
[ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z ]
```

Lösung 1.24

Die Klasse InitialisierungKlassenfelder

```
import java.util.Arrays;
public class InitialisierungKlassenfelder {
// Klassenfeld Initialisierer
   private static byte einbyte;
   private static char einchar = 'A';
   private static int einint ;
   private static short einshort;
   private static long einlong = 1;
   private static float einfloat;
   private static double eindouble = 0.1;
   private static boolean einboolean;
   private static String einString = "Java 6.0";
   private static Punkt einPunkt;
   private static char [] einArray = new char[26];
// Klassenblock Initialisierer
   static {
     for(int i = 0; i<26; i++) {
// buchstabe ist eine lokale Variable und kein Feld der Klasse
       char buchstabe = (char)('A'+ i);
       einArray[i] = buchstabe;
     }
   }
// Selbstreferenzierendes Klassenfeld
   static InitialisierungKlassenfelder selfReferentialFeld =
     new InitialisierungKlassenfelder(
       "Initialisierung von Klassenfelder");
// Konstruktordefinition
   InitialisierungKlassenfelder(String s) {
     einPunkt = new Punkt(1,1);
     einint = 1;
     System.out.println(s);
   }
   public static void main(String args[]) {
// Anzeige der Werte von primitiven Typen
     System.out.println(selfReferentialFeld);
     System.out.println("einbyte: "+einbyte);
     System.out.println("einint: "+einint);
     System.out.println("einshort: "+einshort);
     System.out.println("einfloat: "+einfloat);
     System.out.println("einboolean: "+einboolean);
     System.out.println("einchar: "+einchar);
     System.out.println("einlong: "+einlong);
     System.out.println("eindouble: "+eindouble);
// Anzeige der Werte von Referenztypen
     System.out.println("einString: "+einString);
     System.out.print("einPunkt: ");
// Methode der Klasse Punkt an einer Instanz der Klasse aufrufen
```

```
     einPunkt.anzeige();
// Anzeige der Werte von Arraytypen mit der Methode der Klasse
// Arrays
     System.out.println(Arrays.toString(einArray));
   }
}
```

Programmausgaben

```
Initialisierung von Klassenfelder
InitialisierungKlassenfelder…
einbyte: 0
einint: 1
einshort:  0
einfloat: 0.0
einboolean: false
einchar: A
einlong: 1
eindouble:  0.1
einString: Java 6.0
einPunkt: (1.0,1.0)
[A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]
```

Lösung 1.25

Die Klasse PrivaterKonstruktor

```
import java.util.*;
public class PrivaterKonstruktor {
// Selbstreferenzierendes Klassenfeld
   private static PrivaterKonstruktor datumunduhrzeit;
// Privater Konstruktor der Klasse
   private PrivaterKonstruktor() {
// Instanz der Klasse GregorianCalendar erzeugen und deren
// Methoden aufrufen
     GregorianCalendar calendar = new GregorianCalendar();
     System.out.println("Datum: "+calendar.get(Calendar.DATE)+
       "."+(calendar.get(Calendar.MONTH)+1)+"."+
         calendar.get(Calendar.YEAR));
     System.out.println("Uhrzeit: "+calendar.get(
       Calendar.HOUR_OF_DAY)+"."+calendar.get(Calendar.MINUTE)+
         "."+calendar.get(Calendar.SECOND));
   }
// Klassenmethode, welche ein Objekt der Klasse erzeugt
   public static PrivaterKonstruktor getInstanz() {
     if(datumunduhrzeit == null)
       datumunduhrzeit = new PrivaterKonstruktor();
     return datumunduhrzeit;
   }
}
```

Die Klasse PrivaterKonstruktorTest

```
public class PrivaterKonstruktorTest {
   public static void main(String args[]) {
      PrivaterKonstruktor datum=PrivaterKonstruktor.getInstanz();
   }
}
```

Programmausgaben

```
Datum: 29.6.2007
Uhrzeit: 18.11.36
```

Lösung 1.26

Die Klasse WochenTage

```
public class WochenTage {
// Selbstreferenzierende Klassenfelder können in ihrer
// Deklaration initialisiert werden
   public final static WochenTage MONTAG = new WochenTage(1);
   public final static WochenTage DIENSTAG = new WochenTage(2);
   public final static WochenTage MITWOCH = new WochenTage(3);
   public final static WochenTage DONNERSTAG = new WochenTage(4);
   public final static WochenTage FREITAG = new WochenTage(5);
   public final static WochenTage SAMSTAG = new WochenTage(6);
   public final static WochenTage SONNTAG = new WochenTage(7);
   public String tag;
// Privater Konstruktor der Klasse
   private WochenTage(int i) {
      if(i == 1)
         tag = "Der 1.Wochentag ist der Montag";
      else if(i == 2)
         tag = "Der 2.Wochentag ist der Dienstag";
      else if(i == 3)
         tag = "Der 3.Wochentag ist der Mitwoch";
      else if(i == 4)
         tag = "Der 4.Wochentag ist der Donnerstag";
      else if(i == 5)
         tag = "Der 5.Wochentag ist der Freitag";
      else if(i == 6)
         tag = "Der 6.Wochentag ist der Samstag";
      else if(i == 7)
         tag = "Der 7.Wochentag ist der Sonntag";
   }
}
```

Die Klasse WochenTageTest

```
public class WochenTageTest {
   private WochenTage wt;
   private static WochenTageTest woche;
// Konstruktordefinition
   WochenTageTest(WochenTage wtage) {
      wt = wtage;
      System.out.println(wt.tag);
   }
   public static void main(String args[]) {
      woche = new WochenTageTest(WochenTage.MONTAG);
      woche = new WochenTageTest(WochenTage.DIENSTAG);
      woche = new WochenTageTest(WochenTage.MITWOCH);
      woche = new WochenTageTest(WochenTage.DONNERSTAG);
      woche = new WochenTageTest(WochenTage.FREITAG);
      woche = new WochenTageTest(WochenTage.SAMSTAG);
      woche = new WochenTageTest(WochenTage.SONNTAG);
   }
}
```

Programmausgaben

```
Der 1.Wochentag ist der Montag
Der 2.Wochentag ist der Dienstag
Der 3.Wochentag ist der Mitwoch
Der 4.Wochentag ist der Donnerstag
Der 5.Wochentag ist der Freitag
Der 6.Wochentag ist der Samstag
Der 7.Wochentag ist der Sonntag
```