Kapitel 1

Klassendefinition und Objektinstanziierung

1.1 Klassen und Objekte

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

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

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

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

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

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

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

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

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

Konstruktoren

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

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

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

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

Klassenfelder und Klassenmethoden

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

Klassenmethoden können ebenfalls über den Klassennamen angesprochen werden.

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

Instanzfelder und Instanzmethoden

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

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

Java-Programme

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

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

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

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

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

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

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

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

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

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

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

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

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

Aufgabe 1.1 ☆

Definition einer Klasse

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

Hinweise für die Programmierung

Ein Erzeugen von Instanzen der Klasse ist nicht erforderlich.

Achten Sie auf den richtigen Abschluss der Ausgabezeile.

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

Aufgabe 1.2 ☆

Objekt (Instanz) einer Klasse erzeugen

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

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

1.2 Das Überladen von Methoden

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

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

Aufgabe 1.3 ☆

Eine Methode überladen

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

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

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

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

1.3 Die Datenkapselung, ein Prinzip der objektorientierten Programmierung

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

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

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

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

Aufgabe 1.4 ☆

Zugriffsmethoden

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

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

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

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

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

Aufgabe 1.5 ☆☆

Konstruktordefinitionen

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

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

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

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

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

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

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

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

1.5 Die Wert- und Referenzübergabe in Methodenaufrufen

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

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

Aufgabe 1.6 ☆☆☆

Wertübergabe in Methoden (»call by value«)

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

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

Hinweise für die Programmierung

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

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

1.6 Globale und lokale Referenzen

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

Aufgabe 1.7 ☆☆

Der Umgang mit Referenzen

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

Hinweise für die Programmierung

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

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

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

1.7 Java-Pakete

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

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

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

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

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

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

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

Aufgabe 1.8 ☆

Die package-Anweisung

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

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

Hinweise zu den Programmaufrufen

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

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

Aufgabe 1.9 ☆

Die import-Anweisung

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

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

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

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

Java-Dateien: `kapitel1\paket1\KlassenTest.java`,
`kapitel1\paket1\paket2\Klasse.java`
Programmaufrufe im Verzeichnis `kapitel1`: `javac paket1\KlassenTest.java`
`java paket1/KlassenTest`

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

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

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

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

Aufgabe 1.10 ☆☆

Pakete und die Sichtbarkeit von Membern einer Klasse

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

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

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

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

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

1.9 Standard-Klassen von Java

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

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

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

Die Klasse System

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

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

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

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

Die Klasse File

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

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

Die Klassen Runtime, ProcessBuilder und Process

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

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

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

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

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

Die Klassen Exception und Error

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

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

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

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

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

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

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

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

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

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

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

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

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

Aufgabe 1.11 ☆☆

Das Auto(un)boxing

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

Prüfen Sie, ob einfache Zuweisungen von primitiven Werten für das Bilden der Instanzen von 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 der Instanzen von Wrapper-Klassen mit dem »==-Operator«.

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

Hinweise für die Programmierung

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

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

1.11 Das Paket java.lang.reflect

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

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

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

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

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

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

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

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

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

1.12 Arrays (Reihungen) und die Klassen Array und Arrays

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

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

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

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

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

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

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

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

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

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

Aufgabe 1.12 ☆☆

Der Umgang mit Array-Objekten

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

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

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

1.13 Zeichenketten und die Klasse String

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

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

Compact Strings

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

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

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

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

HTML-Dateien und JavaScripts

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

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

Ein HTML-Dokument besteht aus drei Bereichen:

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

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

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

Textblöcke

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

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

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

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

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

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

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

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

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

und

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

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

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

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

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

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

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

equivalent mit dem String-Literal:

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

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

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

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

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

so erscheint das String-Literal ohne Zeilenumbruch:

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

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

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

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

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

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

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

Interpretieren wir die nachfolgenden Beispiele nach diesen Regeln:

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

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

1. Beispiel:

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

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

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

2. Beispiel:

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

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

3. Beispiel:

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

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

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

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

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

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

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

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

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

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

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

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

nach dem Kompilieren der String:

```
          Hallo
Java
          13!
```

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

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

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

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

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

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

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

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