Eine kleine Java-Rundreise – Teil 1

1. Intro

Dieses Kapitel zeigt Statements, Operatoren und Ausdrücke, die in einer durchschnittlichen Anwendungsprogrammierung häufig vorkommen. So begegnen uns etwa kombinierte Zuweisungen, Inkrement (c++) und Dekrement (c--) oder auch Stringverknüpfungen auf Schritt und Tritt. Dieses Kapitel zeigt und erklärt all die Java-Elemente, die für das Schreiben von einfachem prozeduralen Code typisch sind. Einige Entwickler sehen die genannten Operatoren skeptisch, weil es sich um verkürzende Schreibweisen handelt. Das vorliegende Buch möchte dagegen zeigen, dass kompakte Syntax und lesbarer Code gut zusammenpassen.

Umständlich geschriebener Code erscheint vielleicht Anfängern leichter fassbar, aber er ist für Übersichtlichkeit, Lesbarkeit und klare Strukturen schädlich. Lesbarkeit von Code wird nicht mit Ausführlichkeit und einem hohen Anteil von Leerzeilen erreicht, sondern durch die beständige Nutzung von immer gleichen Redewendungen. Diese werden sozusagen als elementarer Wortschatz in diesem Kapitel behandelt. Wenn sich ein Java-Neuling mit den häufigsten Redewendungen vertraut macht, hat er vermutlich einen einfacheren Einstieg, als wenn er sich durch alle Syntaxelemente mit gleichmäßiger Anstrengung hindurcharbeitet.

Der erste Teil der Java-Rundreise ist nur wenig mehr als ein Umherschweifen in hügeligem Gelände, um die Grundausrüstung kennenzulernen. Der zweite Teil erkundet dann ein größeres Gebiet und ist entsprechend anstrengender. Dazwischen erschließen wir uns angenehme Features wie die Lambdas, schwierige Tools wie die unvermeidbaren Exceptions, und schließlich auch die Philosophie, die Seele im Code, nämlich die Objektorientierung.

2. 90 Prozent - Ein Plan und ein Stück Code - Was ein Entwickler wissen muss

Wenn ein Entwickler einen Plan hat, kann er beginnen, Code zu schreiben. Die Pläne des Entwicklers im Kleinen betreffen dabei immer diese beiden Aufgaben:

- Das Organisieren von Code in Methoden
- Das Schreiben des Codes einer einzelnen Methode.

Die Rede ist von Funktionen. Funktionen heißen in Java Methoden. Beide Bezeichnungen werden weitgehend synonym verwendet. Das Denken eines Entwicklers dreht sich um Methoden. Jeglicher Code in Java läuft in Methoden ab. Und jede Methode ist Teil einer Klasse. Um Code zu schreiben, muss ein Entwickler eine Klasse bereitstellen und in dieser oder einer anderen Klasse eine main()-Funktion. Denn um ein Programm in Java auszuführen, braucht man eine main()-Funktion. Mit deren Aufruf wird das Programm gestartet. Ein Kopfdruck an der richtigen Stelle in der Entwicklungsumgebung und die Eingabe des Namens ‚Experiment' in einen Dialog der Entwicklungsumgebung erzeugen eine Klasse mit der gewünschten main()-Methode (siehe den ersten Abschnitt im Anhang, wenn Sie hier Unterstützung brauchen):

```
public class Experiment                          // Dieser Bezeichner ist beliebig
{
    public static void main(String[] args)       // main() jedoch muss es immer geben
    {
    }
}
```

Ein weiterer Knopfdruck bringt die Klasse zur Ausführung. Das lässt uns natürlich kalt, denn sie tut ja nichts und folglich sehen wir auch nichts. Sollten Sie diese Klasse nicht mit einer Entwicklungsumgebung erzeugen, sondern vollständig selbst erstellen, dann achten Sie darauf, dass die Klasse in einem File mit dem Namen der Klasse und der Namenserweiterung .java steht. Die Klasse Experiment muss in dem Java-File Experiment.java gespeichert sein. Jede öffentliche Klasse muss in einem eigenen File ihres Namens stehen. Sie sollten zudem darauf achten, dass die Kopfzeile der main()-Funktion genau das gezeigte Aussehen hat. Wenn Sie vom Ablauf der Klasse etwas sehen wollen, dann stellen Sie Ausgabe-Statements wie das folgende in die main()-Funktion:

```
System.out.println("Hello World");
```

Den Button, dessen Betätigung dieses erste Programm zum Ablauf bringt, finden Sie ebenfalls in Ihrer Entwicklungsumgebung. In Eclipse ist dies: *Run → Run As → Java Application*. Wenn Sie nicht mit einer Entwicklungsumgebung arbeiten, müssen Sie das File Experiment.java zuerst in der Konsole übersetzen (*javac Experiment.java*). Dann können Sie den Code vom Interpreter ausführen lassen (*java Experiment*). Informationen über Compiler und Interpreter finden Sie im Anhang. Um sicherzustellen, dass Sie in der Lage sind, eine Klasse zu erstellen, gibt es im Anhang zudem ein kurzes Kapitel, das sich den möglichen Schwierigkeiten bei der ersten Klasse widmet. Denn ohne eigene Experimente, mit denen Sie die hier vorgestellten Beispiele und Codefragmente erproben, macht das Lesen dieses Buches wenig Sinn.

Ein Entwickler, der sich eine bevorstehende Codierung zurechtlegt, denkt entweder darüber nach, wie er den Code in Methoden aufteilt, oder wie er den Code einer einzelnen Methode schreibt. Im ersten Fall hat er etwas in dieser Art vor Augen:

```
public class Experiment
{
    public static int getMyInput() {…}                  // Die geschweiften Klammern müssen
    public static int calculate(int number) {…}          //   noch mit Code gefüllt
    public static void writeItOut(int number) {…}        //   werden
    public static void main(String[] args)               // Programmausführung beginnt hier
    {
        int number = getMyInput();                       // Führe der Reihe nach diese drei
        int result = calculate(number);                  //   Methoden aus
        writeItOut(result);
    }                                                    // Mit dem Ende von main() ist auch
}                                                        //   Programmausführung beendet
```

Wir sehen hier ganz gut, wie der Entwickler sich die Sache denkt. Sein Plan nimmt Gestalt an und man kann den Plan lesen. Man muss sich das so vorstellen, dass wir hier zwar nur eine Klasse zeigen, dass in der Regel aber viele Klassen beteiligt sind, die allesamt sprechende Namen haben, und dass es in jeder dieser vielen Klassen viele Methoden mit ebenfalls sprechenden Namen gibt. Der Witz bei der Organisation von Code in Methoden liegt darin, dass der Code so aufgeteilt wird, dass die zu lösende Aufgabe in jeder einzelnen Methode möglichst übersichtlich und der Plan als Ganzes möglichst gut lesbar und zu verstehen ist.

Der ‚Plan als Ganzes' ist dabei eine schöne Idee. Entwickler verfassen Pläne, wenn sie Software schreiben. Man sieht jedoch bald, dass es nicht ganz einfach ist, einen Plan zu machen und ihn aufzuschreiben. Die Schwierigkeiten beginnen bereits mit der Frage, wie ein aufgeschriebener Plan denn aussehen oder wann man ihn verfassen soll. Einen funktionierenden Plan zu erstellen, ist bedeutend schwieriger als die anschließende Umsetzung in Code. Planung und Codierung haben aber andererseits eine Menge miteinander zu tun. Sie verhalten sich etwa so wie Denken und Sprache. Das eine geht nicht ohne das andere. Niemand kann ohne fun-

dierte Codekenntnisse einen vernünftigen Plan machen und umgekehrt ist eine Codierung ohne Plan genau das, wonach es manchmal aussieht – eben planlos.

In diesem Buch geht es sehr viel um Denken und Sprache und um Plan und Code. Man könnte auch sagen, es geht darum, wie man Software konstruiert. Für Konstruktionen braucht man mehr als die Syntax einiger Java-Statements. Wir brauchen dazu die Fähigkeit der Konzeption, das ist Planung, und wir brauchen die Fähigkeit der Umsetzung, das ist Coding. Und zudem brauchen wir eine gewisse Grundschnelligkeit. Es geht nicht nur darum, irgendwann einmal nach langer Zeit und vielen Anläufen alles in schöne Ordnung gebracht zu haben. Es geht auch um die Fähigkeit, mit einer hohen Produktivität zu konstruieren, also robuste Konstruktionen in kurzer Zeit zu erstellen. Planung und Coding und ihre Verbindung müssen so aufgesetzt sein, dass Konstruktionen rational und damit schnell, sehr schnell, durchgeführt werden können.

In diesem Hauptkapitel geht es in erster Linie um Coding. Rationalität beginnt jedoch - in kleinen Ansätzen - bereits hier. Wir müssen nicht jedes Syntaxelement in gleicher Breite beschreiben. Wir konzentrieren uns bei den Coding-Themen auf diejenige Auswahl von Syntax und Redewendungen, die geschätzte 90 Prozent des ‚durchschnittlichen' Codes ausmachen. Wir betrachten dazu Tools wie Strings, Collections und Enums und eben elementare Syntax. Die 90-%-Regel gilt für elementare Syntax und Tools gleichermaßen. Mit einer Handvoll Tools erledigt man 90 Prozent der anfallenden Aufgaben in einer ‚durchschnittlichen' Entwicklung. Mit der Beherrschung der wichtigsten Sprachmittel und einer guten Kombination aus Planung und Coding beginnt das minimale Konstruktionswissen, über das ein Entwickler verfügen sollte.

Sieht man nochmals auf die obige Klasse Experiment mit ihren Methoden, so verschwimmt die Grenze zwischen Planung und Code. Ist die Aufteilung in Methoden ein Mittel der Planung oder ein Mittel der Codierung? Tatsache ist, dass der Methodenaufruf eines der wichtigsten Elemente ist, um den Programmablauf zu kontrollieren. Und die Organisation von Code in Methoden ist ein elementares und zugleich wichtiges Engineering-Mittel. Wir behandeln Methoden und ihre Aufrufe deshalb als erstes. Der Java-Entwickler hat daneben aber auch alle anderen gängigen Sprachmittel zur Verfügung, die man in modernen prozeduralen und objektorientierten Sprachen kennt. Unter anderem sind das diese:

Methodenaufrufe. Um Leistungen an andere Methoden zu delegieren:

```
public static int perform()
{
        int fromUser = getMyInput();
        int result = calculate(fromUser);
        writeItOut(result);
}
```

Bedingungen. Um Prüfungen durchzuführen:

```
public static int perform()
{
        int fromUser = getMyInput();
        if (fromUser < 0)                                   // Bedingung: Ist diese Zahl kleiner 0
                writeItOut("Info für Anwender – deine Eingabe: " + fromUser);
}
```

Logische Operatoren. Um Bedingungen zu verknüpfen:

```
public static int perform()
{
        int fromUser = getMyInput();
        if (fromUser == 0 && getResults() == null)                    // Logisches Und
        {
                writeItOut("Info für Anwender – null Eingabe");
                try {Thread.sleep(10_000);} catch(Throwable ignore) {}
                System.exit();
        }
}
```

Verzweigungen. Um mehrwertige Bedingungen zu behandeln:

```
public static int perform()
{
        int fromUser = getMyInput();
        …
        switch(fromUser)
        {
        case -1:
                writeItOut("Info für Anwender");
                break;
        case 0:
                writeItOut("Andere Kommentierung der Eingabe");
                break;
        case 1:
                writeItOut("Ergebnis … ");
                break;
        }
}
```

Iterationen. Um Elemente einer Menge zu durchlaufen:

```
public static int calculate(int number)
{
        int[] array = getArray(number);
        int index = 0;
        while(index < array.length)                                   // Iteriere über das Array
                writeItOut(array[index++]);
}
```

Einige vorkommende Methoden wie getResults() oder getArray() sind hier nicht wesentlich und werden deshalb nicht weiter erläutert. In der Praxis müssen natürlich alle Methoden, die aufgerufen werden, an anderer Stelle vereinbart sein. Im weiteren Verlauf des Kapitels greifen wir die gezeigte Basissyntax auf, erläutern Regeln, auf die es ankommt, und zeigen an Standard-Situationen, die man häufig antrifft, was man tun und

was man vermeiden sollte. Wir gehen dabei davon aus, dass die Leserinnen und Leser mit der grundsätzlichen Wirkungsweise einer Programmiersprache bereits halbwegs vertraut sind. Ein Java-Neuling, der von einer anderen Sprache her kommt, wie etwa Pascal, C oder C++, sollte mit dem gewählten Niveau gut zurecht kommen.

3. Methoden

3.1. Die Signatur von Methoden

Die Grundlage von elementarem Java sind Methoden. Methoden sind ein erster Ansatz von Engineering. Sie dienen dazu, Code zu organisieren und die zu erbringende Leistung in benennbare Einheiten aufzuteilen. Eine Methode besteht aus Methodenkopf und Methodenkörper:

```
public static int printLongArray(long[] array, String separator)   // Methodenkopf
{                                                                   // Ab hier Methodenkörper
        …                                                           // Die Klasse, zu der diese Methode
}                                                                   //   gehört, ist nicht dargestellt
```

Der Methodenkopf enthält alle formalen Deklarationen zu einer Methode und legt Ein- und Ausgang fest. Er bestimmt die Sichtbarkeit (private oder public), den Returntyp und die Argumente. Die Sichtbarkeit von printLongArray() ist public, was bedeutet, dass die Methode auch außerhalb ihrer Klasse gesehen und zugegriffen werden kann. printLongArray() hat zwei Argumente, nämlich array und separator mit den Standardtypen long[] und String. Die Argumente sind der Eingang einer Methode. Sie legen fest, was an eine Methode bei ihrem Aufruf übergeben wird. Das Gegenstück dazu ist der Returnwert. Dieser ist das Ergebnis, das eine Methode nach ihrer Ausführung an den Aufrufer zurückliefert. Der Methodenkopf von printLongArray() legt fest, dass der Returnwert den elementaren Typ int hat. Die geordnete Liste der Argumente mit ihren Typen zusammen mit Methodenname, Returntyp und Sichtbarkeit nennt man auch die Signatur einer Methode.

Eine jede Methode ist Teil einer Klasse. Auch printLongArray() muss in irgendeiner Klasse vereinbart sein. Nehmen wir an, dass es sich dabei um die Klasse Experiment handelt, die wir bereits kennen. Wie die anderen Experiment-Methoden hat auch printLongArray() den Qualifier static. Java-Methoden werden nicht grundsätzlich als static vereinbart. Bis wir die Instanzmethoden kennenlernen, die nicht statisch sind (siehe dazu im Kapitel Objekte), sollten Sie Ihre Experimente jedoch mit statischen Methoden ausführen. Die Reihenfolge der Schlüsselworte public und static spielt übrigens keine Rolle. Beide Qualifier müssen aber vor dem Returntyp stehen. Wenn eine (statische) Methode von beliebigen anderen Klassen aus aufgerufen werden soll, so muss sie dazu public sein und der Methodenaufruf muss dabei durch den Klassennamen qualifiziert werden:

```
Experiment.printLongArray(…);                                       // Aufruf über Klassengrenzen hinweg
```

Um eine statische Methode der eigenen Klasse aufzurufen, genügt es, den Namen der Methode zu nennen. Der Aufruf einer Methode muss in jedem Fall ihrer Signatur entsprechen. Dies bedeutet, dass alle Argumente mit dem richtigen Typ gegeben und in der richtigen Reihenfolge präsentiert werden müssen. Der Returnwert kann entgegen genommen werden oder auch nicht. Im ersteren Fall muss auch für ihn eine Variable mit dem richtigen Typ bereitgestellt werden. Hier ist eine kleine Reihe von Beispielstatements, welche die Aufrufregeln illustrieren:

```
public class Experiment
{
    public static void main(String[] args)
    {
        long[] longArray = new long[100];          // Zu Arrays gibt es einen eigenen
        Arrays.fill(longArray, 131313);            //   ausführlichen Abschnitt
        printLongArray(longArray, " : ");          // Klappt
        printLongArray(null, null);                // Okay für Compiler. Laufzeitfehler!!

        int[] intArray = new int[100];
        printLongArray(intArray, " : ");           // Error. int[] passt nicht auf long[]

        printLongArray(longArray);                 // Error. Nur ein Argument statt zwei
        printLongArray(longArray, 13);             // Error. Zweiter Arg-Typ passt nicht
        String result = printLongArray(longArray, " : ");   // Error. Returntyp passt nicht
    }

    public static int printLongArray(long[] array, String separator) { … }
}
```

Derartige im Kommentar markierte Fehler sind immer harmlos. Denn es handelt sich um Fehler, die der Compiler bemerkt. Der Compiler lehnt dann die Übersetzung des betreffenden Codes ab. Ein Fehler, den der Compiler bemerkt, ist deshalb harmlos, weil er vom Entwickler unmittelbar erkannt wird und behoben werden kann. Ein nicht-harmloser Fehler ist ein Fehler, der erst zur Laufzeit auftritt, alle Test übersteht, nur sporadisch auftritt und das erste Mal in Erscheinung tritt, wenn die Applikation gerade an 1000 Anwender verteilt worden ist.

Das Beispiel zeigt, dass beim Aufruf einer Methode die übergebenen Argumente in ihren Typen auf die deklarierten Argumente passen müssen. Ebenso muss der Returntyp zum Typ der aufnehmenden Variable passen. Dies ist in Java elementar. Der Compiler kann geringfügige Anpassungen automatisch vornehmen, etwa wenn durch den Typ eines Arguments ein long gefordert und beim Aufruf irgendein anderer ganzzahliger Typ wie short oder byte übergeben wird. Denn diese Typen lassen sich ohne Informationsverlust in ein long umwandeln (siehe unter Cast). Wenn beim Methodenaufruf die Erwartung an die Typen der Argumente und ihre Anzahl nicht erfüllt wird, lehnt der Compiler ab und übersetzt nicht. Da der Name der Methode nicht das einzige entscheidende Kriterium bei ihrem Aufruf ist, können verschiedene Methoden mit gleichem Namen vereinbart werden, wenn sie sich in Zahl, Reihenfolge oder Typ der Argumente unterscheiden. Zwei Methoden gleichen Namens in einer Klasse, die sich nur im Returntyp unterscheiden, sind dagegen nicht erlaubt.

Verschiedene Methoden können den gleichen Namen tragen, wenn sich ihre Argumente durch Anzahl, Typ oder Reihenfolge unterscheiden

Wenn mehrere Methoden mit gleichem Namen innerhalb einer Klasse vereinbart werden, so nennt man dies Overloading. Overloading (Überladen) ist etwas sehr Triviales. Gäbe es für Instanzmethoden nicht einen wichtigen ähnlichen Mechanismus, nämlich Overriding, so würde der Term ‚Overloading' hier gar nicht erwähnt werden. Das folgende Beispiel zeigt einige mögliche Überladungen in der Klasse Experiment:

```
public class Experiment
{
        public static void main(String[] args) {… }
        public static int printLongArray(long[] array, String separator) { … }
        public static int printLongArray(long[] array, String separator, boolean useSingleLine)  {… }
        public static void printLongArray(long[] array, String separator, short itemsPerLine)  {… }
        public static int printLongArray(int[] array, String separator) { … }
}
```

Wir sehen einfache Überladungen, die dadurch entstehen, dass die Zahl der Argumente variiert wird oder dass ein Argument einen anderen Typ erhält. Natürlich kann man Überladungen auch dadurch erreichen, dass man die Reihenfolge der Argumente vertauscht. Allerdings sollte für den Anwender der Methoden ohne Weiteres ersichtlich sein, was diese tun und worin ihr Unterschied liegt. Im obigen Beispiel gibt es da keinerlei Schwierigkeiten. Im folgenden Beispiel ist dagegen von außen nicht zu erkennen, was die beiden printLongArray()-Methoden unterscheidet:

```
public class Experiment                                    // Problematische Überladung
{
        public static void main(String[] args) {… }
        public static int printLongArray(int[] array, String separator) { … }
        public static int printLongArray(String separator, int[] array) { … }
}
```

Die Möglichkeit, mehrere Methoden in einer Klasse mit gleichem Namen zu verfassen, ist nicht nur theoretisch gegeben, sondern wird in der Praxis ausgiebig genutzt. Es ist sehr hilfreich, nicht beständig neue Methodennamen erfinden zu müssen, wenn man mehrere Methoden mit einer ähnlichen Leistung anbieten möchte. Welche Namen würde man für die Menge der printLongArray()-Methoden im obigen Beispiel wählen, wenn Overloading nicht zulässig wäre. Einige von uns würden es sicher mit printLongArray1(), printLongArray2() und so fort versuchen. Bei der Anwendung dieser Methoden würde man dann den unangenehmen Effekt entdecken, dass man bei jedem Methodenaufruf erst mühsam herausfinden muss, welche Zahl, beziehungsweise welcher Name zu welcher Argumentkombination gehört. Von daher ist Overloading zwar kein großes Feature, aber sehr praktisch.

3.2. Werden Argumente und Returnwerte kopiert?

In einer der obigen Versionen von printLongArray() sind drei Argumente gegeben: ein Array, ein String und ein boolean. Was passiert mit diesen Argumenten beim Methodenaufruf? Werden die Argumente kopiert? Oder sieht die gerufene Methode das Original? Ist es für den Aufrufer relevant, wenn die gerufene Methode ein Argument ändert? Die Antwort auf diese Fragen liegt in dem Unterschied zwischen elementaren Typen wie boolean und wirklichen Objekten (Instanzen) wie String und Array. Wir beginnen diese wichtige Erläuterung mit einer Nonsense-Implementierung von printLongArray():

```
public static void main(String[] ignore)                   // Dieses Array interessiert hier nicht
{
        long[] longArray = new long[100];                  // Erzeuge Array mit 100 Elementen
        Arrays.fill(longArray, 131313);                    // Jedes Element hat nun diesen Wert
        boolean singleLine = true;
```

```
        String delimiter =  " : ";
        printLongArray(longArray, delimiter, singleLine);      // Auf diesen Aufruf kommt es hier an
        int test = array[0];                                   // 23. Dieser Wert wurde verändert
        boolean flag = singleLine;                             // true. Dieser Wert ist unverändert
}

public static int printLongArray(long[] array, String separator, boolean useSingleLine)
{
        array[0] = 23;                                         // Änderung, die in main() sichtbar ist
        array = new long[1];                                   // Änderung der Array-Referenz
        useSingleLine = false;                                 // Änderung eines elementaren Typs
        separator.replace(" ", ".");                           // ! Änderung eines Strings. Vorsicht !
        …
}
```

Möglicherweise sind Sie ein ungeduldiger Mensch und möchten sofort wissen, was denn in Java ein boolean ist oder ein long oder was elementare Typen sind. Wenn Sie öfters von dieser Ungeduld getrieben werden, so gibt es für Sie zwei einfache Optionen. Diese heißen Glossar und Essentials. Beide sind Orte der Erklärung und Teil dieses Buches und befinden sich somit in Ihrem unmittelbaren Zugriff. Die genannten Fragen haben mit dem Typsystem von Java zu tun, das in einem späteren Teil dieses Buches behandelt wird. Eine kurze Übersicht ist hier hilfreich.

Es gibt in Java acht elementare Typen, die auch als primitive Typen bezeichnet werden. Diese umfassen die vier ganzzahligen Typen byte, short, int und long, die zwei Gleitkommatypen: float und double, den Typ char, um Einzelzeichen aufzunehmen, und schließlich den mit allen anderen inkompatiblen Typ boolean. Während sieben elementare Typen zumindest mit Zwang (Cast) ineinander umgewandelt werden können, hat boolean eine Sonderstellung. Eine boolean Variable kann nur die beiden Werte true und false (beides sind Schlüsselwörter) annehmen und kann in keinen anderen Typ umgewandelt werden.

Alle Größen in Java, die keinen elementaren Typ haben, sind Instanzen von Klassen. Während die elementaren Typen ein fester Teil der Sprache sind, sind Klassen im Prinzip selbst-vereinbarte Typen, die nach Belieben erzeugt und dem System Java hinzugefügt werden können. Obwohl man Klassen auch Benutzer-definierte Typen nennt, werden nicht alle Klassen vom Entwickler selbst bereitgestellt. Eine große Menge von Klassen wird unter der Bezeichnung ‚Standardklassen' mit Java mitgeliefert und ist Bestandteil des JDK.

Variable haben in Java also entweder einen elementaren Typ oder sie verweisen auf Instanzen von Klassen. Im letzteren Fall werden sie auch als Referenzen bezeichnet. Eine Referenz ist ein Zeiger, ein Pointer. Typen, die nicht elementar sind, sind Referenztypen. Und damit sind wir schon wieder mitten im Thema dieses Kapitels. Denn ob eine Variable bei der Übergabe an eine Methode kopiert wird oder nicht, hängt davon ab, ob sie ein Referenztyp oder ein elementarer Typ ist.

Wir beginnen mit dem dritten Argument in obigem Beispiel. Dieses hat in main() die Bezeichnung singleLine und in printLongArray() die Bezeichnung useSingleLine. Da es sich um einen elementaren Typ handelt, wird diese Variable bei der Übergabe an eine Methode (oder bei der Rückgabe aus einer Methode) kopiert. Eine Methode erhält bei ihrem Aufruf für einen elementaren Wert quasi ein Stück Papier ausgehändigt, auf das der elementare Wert übertragen wurde. Sie kann mit diesem Wert anstellen, was sie möchte, ihn verändern oder ihn löschen, die rufende Methode bekommt davon nichts mit, denn es gibt den betreffenden Wert ja zweimal: einmal in der rufenden und einmal in der gerufenen Methode. Eine Veränderung von useSingleLine in printLongArray() ist in der rufenden Methode main() folglich nicht zu sehen.

Anders ist dies bei der Übergabe von Referenzen beim Methodenaufruf. Eine Referenz ist ein Zeiger auf eine Instanz, der vier Bytes im Speicher belegt und damit ähnlich groß ist wie ein elementarer Wert, der ebenfalls

nur einige Bytes benötigt. Und ebenso wie ein elementarer Wert wird auch eine Referenz bei der Übergabe an eine Methode kopiert. Aber die Instanz, auf welche die Referenz zeigt, wird nicht kopiert. Wenn eine Referenz an eine Methode übergeben wird, können wir uns diese als eine Drachenschnur vorstellen. Da die Referenz kopiert wird, gibt es zwei Drachenschnüre. Eine hat die gerufene und die andere hat die rufende Methode in der Hand. Und beide zeigen auf die gleiche Instanz. Wenn die gerufene Methode auf das Objekt zugreift, das am Ende der Drachenschnur hängt, und dieses verändert, so ist die Veränderung für die gerufene Methode sichtbar. Das erste Statement in printLongArray() macht genau dieses, es ändert das Array-Objekt, und zwar das Element mit dem Index 0. Diese Änderung ist anschließend in main() zu sehen.

Arrays sind in Java echte Objekte, auch wenn es sich um Arrays mit elementarem Elementtyp handelt. Variable, die Arrays enthalten, sind also Referenzvariable. Das zweite Statement in printLongArray() erzeugt ein neues long-Array und stellt dessen Referenz in die Variable array. Die Methode lässt also die übergebene Drachenschnur los und nimmt stattdessen eine andere Drachenschnur in die Hand, an deren Ende ein anderes Objekt hängt. Ab diesem Punkt ist das übergebene Array für printLongArray() nicht mehr zugreifbar. Die betreffende Referenz wurde ersetzt. Und die Bedeutung für die rufende Methode ist offensichtlich. Wenn eine Methode das übergebene Objekt loslässt, so ist dies für die rufende Methode nicht relevant. Sie bekommt davon nichts mit.

Betrachten wir nun das zweite Argument, den String separator, der in main() delimiter heißt. Die Variable separator ist ebenfalls eine Referenz. Das String-Objekt wird deshalb wie alle echten Objekt im Original übergeben und die Änderungen, die printLongArray() an separator vornimmt, müssten folglich auch in main() zu sehen sein. Doch ein String ist ein ganz besonderes Objekt. Er kann nicht wirklich verändert werden. Ein String ist unveränderbar. Wäre separator ein anderer Typ, dann würde der replace()-Aufruf das Objekt verändern (wenn es diese Methode dann gäbe) und die Änderung wäre in printLongArray() und in main() zu sehen. Bei einem String jedoch erzeugt eine Manipulation ein neues Objekt. Dieses neue Objekt enthält die betreffende Änderung und wird als Ergebnis des Methodenaufrufs zurückgegeben. Da printLongArray() dieses Ergebnis nicht auffängt, ist das gezeigte Statement überflüssig, denn es hat auf diese Weise weder in printLongArray() noch in main() eine Wirkung. Richtig schreibt man Stringmanipulationen deshalb so:

```
separator = separator.replace(" ", ".");                    // Dieses Statement hat Wirkung
```

Nun wird die von String.replace() gelieferte Drachenschnur aufgenommen und das Statement macht Sinn. Mehr zu diesem grundlegenden Thema der Referenzen gibt es weiter unten im Teilkapitel über Objekte. Hier halten wir nach dieser ausführlichen Behandlung einer sehr wichtigen Eigenschaft von Java fest, dass im Standardfall bei der Übergabe von Werten zwischen Methoden keine Kopien erzeugt, sondern Originale ausgetauscht werden. Die unbedeutende Ausnahme dabei betrifft die wenigen elementaren Typen, die es in Java gibt.

3.3. Returntyp und return-Statement

Eine Methode, die kein Ergebnis zurückliefert, hat den ‚Returntyp' void. void ist ein ganz eigentümliches Ding. void ist nicht wirklich ein Typ, es handelt sich weder um einen elementaren Typ noch um einen Benutzer-definierten Typ. Aber void ist ein Schlüsselwort. Und dieses bedeutet so viel wie ‚leer' und besagt, dass eine Methode nichts zurückgibt. Normale Methoden, die keinen Ergebniswert liefern, müssen als void deklariert werden. Im Falle von printLongArray() würde dies so aussehen:

```
public static void printLongArray(long[] array, String separator) {…}
```

Mit einer einzigen Ausnahme müssen alle benannten Methoden in Java den Typ ihres Returnwerts vereinbaren oder als void deklarieren werden. Die eine Ausnahme sind die Konstruktoren, denen wir bei anderer Gelegenheit wiederbegegnen. Eine Methode mit einem Returntyp ungleich void muss mindestens ein return-Statement enthalten. Hier sehen wir die Methode printLongArray() (jetzt wieder mit einem int-Ergebniswert) mit zwei return-Statements:

```
public static int printLongArray(long[] array, String separator)
{
    if (array == null)                          // Siehe Bedingung
        return 0;                               // Ein bedingtes return-Statement
    for (long element : array)                  // Siehe Iterationen
        System.out.print(element + separator);  // Siehe Stringverknüpfung
    return 1;                                   // Noch ein return-Statement
}
```

Ein return-Statement beendet die Ausführung einer Methode und gibt den Ergebniswert an den Aufrufer zurück. Jedes return-Statement muss dabei dem vereinbarten Returntyp der Methode entsprechen, es muss also einen Wert des Typs zurückgeben, der im Methodenkopf deklariert ist.

Eine void-Methode, also eine Methode mit dem Returntyp void, kann ebenfalls return-Statements enthalten, muss aber nicht. Bei void-Methoden ist das return-Statement leer:

```
public static void printLongArray(long[] array, String separator)  // void-Methode mit return-Statements
{
    if (array == null)
        return;                                 // Leeres return-Statement
    for (long element : array)
        System.out.print(element + separator);
    return;                                     // Der Compiler wird dies kritisieren
}
```

Ein leeres return-Statement am Ende einer Methode ist natürlich überflüssig, da die Methode hier sowieso zurückkehrt. Der Compiler wird deshalb einen entsprechenden Kommentar dazu abgeben (eine Warnung). Eine void-Methode kommt auch ohne return-Statements aus:

```
public static void printLongArray(long[] array, String separator)  // Methode ohne return-Statement
{
    for (long element : array)
        System.out.print(element + separator);
}
```

3.4. Aufteilung von Code in Methoden

Der springende Punkt an der Aufteilung von Code in Methoden besteht darin, den Code so zu organisieren, dass eine bestimmte Leistung nicht mehrmals geschrieben wird. In einem Programm, das mit long-Arrays hantiert und diese des Öfteren auf Konsole ausgibt, ist printLongArray() sicher hilfreich und vermeidet, dass

die gleichen Ausgabe-Statements an mehreren Stellen im Code auftauchen. In den folgenden Zeilen sehen wir ein printLongArray(), das die Array-Elemente auf Zeilen verteilt, die ausgegebenen Zeilen zählt und diese Zahl als Ergebnis zurückgibt:

```
public static int printLongArray(long[] array, String separator, int itemsPerLine)
{
        int lineCounter = 0;
        int itemCounter = 0;
        for (long element : array)                          // Siehe Iterationen
        {
                System.out.print(element + separator);
                itemCounter++;                              // Siehe Post-Inkrement
                if (itemCounter == itemsPerLine)            // Siehe Gleichheits-Operator
                {
                        System.out.println();               // Schreibe einen Zeilenwechsel
                        lineCounter++;                      // Siehe Post-Inkrement
                        itemCounter = 0;                    // Setze den Zähler zurück
                }
        }
        System.out.println();

        return lineCounter;
}
```

Wenn – unabhängig vom genauen Inhalt - dieser Code nicht in einer Methode isoliert ist, dann steht er in leicht variierenden Fassungen an verschiedenen Stellen im Programm. Man spricht dann von redundantem Code. Bei einer Fehlerkorrektur oder anderen Änderungen arbeitet man an mehreren Stellen und hat entsprechend mehrfachen Aufwand.

Code kann bequem in Methoden aufgeteilt werden. Eine der Richtlinien dabei heißt: strikte Vermeidung von Code-Redundanz

Die Verteilung von Code auf viele Methoden ist für einen Entwickler nicht schwierig, sondern erleichtert ihm das Leben. Es ist bequem, eine bestimmte Codemenge in einer größeren Zahl von Methoden zu strukturieren, weil Code damit flexibler und leichter zu handhaben wird und weil er mit der Einordnung unter Methodennamen auch leichter zu erfassen ist. Ein Stück Code in eine Methode zu stecken, ist ein ganz einfacher Weg, dieses Stück Code zu benennen und es damit der Modellierung, dem Nachdenken oder einer Diskussion zugänglich zu machen.

Dass man eine bestimmte Leistung nicht mehrfach implementieren möchte, ist naheliegend. Die Verpackung von Leistung in entsprechend benannte Methoden hat aber noch andere Vorteile. Beispielsweise kann eine Methode durch Argumente konfiguriert werden, was bei einem Codestück, das in eine umfangreiche Statementfolge eingebettet ist, nicht möglich ist.

Gut geschriebener Code besteht aus einer Vielzahl kleiner und kleinster Methoden

Ein weiterer gewichtiger Punkt, der für die Aufteilung von Code in viele Methoden spricht, ist der Umfang der einzelnen Methoden. Je größer die Gesamtzahl aller Methoden ist, desto kürzer sind die einzelnen Methoden. Kurze Methoden sind uns sehr viel lieber als lange Methoden. Ein Programm mit wenigen aber sehr umfangreichen Methoden ist schwierig in Pflege, Wartung und Weiterentwicklung, weil man große Probleme hat, das Programm zu verstehen. Ein Programm mit vielen kleinen Methoden erhält allein schon durch die Methodenköpfe eine Dokumentierung, die eventuell gut gelesen und verstanden werden kann.

Entwickler müssen sich hin und wieder klar machen, welchen Spielraum sie durch die Gestaltung der Methoden und Methodenköpfe haben, in die sie den Code aufteilen. Die Methodenköpfe bestimmen das Vokabular, mit dem Code formuliert wird

Bei einem Programm, das durch viele kleine Methoden strukturiert ist, weiß man eventuell sehr schnell, in welcher Methode man Änderungen durchführen muss, um einen gewünschten Effekt zu erreichen. Bei sehr umfangreichen Methoden ist es dagegen äußerst schwierig, alle Auswirkungen einer Änderung zu erfassen. Man muss selbst bei kleinsten Korrekturen die Methode in ihrer ganzen Länge lesen und verstehen. Der Aufwand kann hierbei so groß sein, dass man irgendwann dazu übergeht, laufenden Code gar nicht mehr anzurühren. Dann ist man bei einer sehr teuren aber nicht seltenen Codeform angelangt.

Die Aufteilung in viele kleine Methoden ist auch für das Verfahren der Protokollierung vorteilhaft. Oftmals möchte man während einer Entwicklung, dass man von dem ablaufenden Code mehr sieht als nur die Ausgabe eines Resultats. Ein einfaches Verfahren protokolliert das Betreten und Verlassen von ausgewählten Methoden. Wenn das ganze Programm nur aus wenigen Methoden besteht, sieht man bei diesem Ansatz naturgemäß nur wenig. Wenn es dagegen viele kleine Methoden gibt, erhält man eine reichhaltige Wiedergabe der internen Abläufe. Hier sehen wir eine einfache Form der Protokollierung:

```
public static int printLongArray(long[] array, String separator)    // Methode mit Protokollierung
{
    System.out.println("Start of printLongArray ");                 // Das ist so nicht ideal
    …
    System.out.println("End of printLongArray ");
    return lineCounter;
}
```

Diese simple Protokollierungstechnik hat allerdings einen großen Hacken, der sich mit etwas Praxis deutlich und schnell bemerkbar macht. Eine Protokollierung dieser Art lässt sich nämlich nicht mehr abstellen. Es ist jedoch ganz wesentlich, dass man Codeteile nach einiger Zeit der Erprobung gezielt aus der Protokollierung wieder heraus nehmen kann. Hier gibt es wesentlich raffiniertere Ansätze, denen wir noch begegnen werden.

3.5. Variabel lange Argumentlisten

„Einer Versuchung sollte man nachgeben.
Wer weiß, wann sie wiederkommt.“

Oscar Wilde

Eine besondere Art der Vereinbarung von Argumenten findet man bei Methoden mit variabel langen Argumentlisten (Varargs). Im Aufruf sehen diese so aus:

```
print();
print(1, 2);
print(2, 3, 5, 7, 11, 13);
```

Die gezeigte print()-Methode gibt es nur einmal. Um eine Methode mit variabel vielen Argumenten zu vereinbaren, wird eine besondere Syntax verwendet, nämlich drei aufeinander folgende Punkte. Hier sehen wir eine Vereinbarung einer variabel langen Liste von int-Argumenten:

```
public static void print(int … array)                    // Variabel viele int-Argumente
{
        int count = array.length;                        // int… wird in int[] übersetzt
        int firstArg;
        if (count > 0)
                firstArg = array[0];
}
```

Die Methode print() kann mit beliebig vielen int-Argumenten aufgerufen werden, auch ohne Argumente. Der Compiler präsentiert innerhalb der Methode eine variabel lange Argumentliste als ein Array (siehe zu Arrays weiter hinten), wobei der Argumenttyp zum Elementtyp des Arrays wird. Der Aufrufer kann an print() aber auch gleich ein Array übergeben:

```
int[] arguments = new int[0]
print(arguments);                                        // Gültiger Aufruf von Vararg-Meth.
```

Die Gleichstellung dieser beiden Methoden aus Sicht des Compilers:

```
public static void print(int … array) {…}
public static void print(int[] array) {…}                // Error. Compiler sieht das als gleich
```

führt dazu, dass es nicht erlaubt ist, in einer Klasse zwei Methoden zu vereinbaren, deren einziger Unterschied in einer variabel langen Argumentliste und einem Array vom selben Typ besteht. Man beachte im Übrigen, dass es trotz dieser Gleichstellung durch den Compiler nicht möglich ist, die print()-Methode aus dem letzten Statement mit den eingangs gezeigten print()-Beispielen aufzurufen. Eine Methode mit einem Array-Argument kann nur mit einem Array aufgerufen werden. Eine Methode, an die nahezu beliebige Argument-Kombinationen übergeben werden können, ist die folgende:

```
public static void test(Object ... array)
{
        System.out.println(array.length);
}
```

Auch die eingangs gezeigten print()-Aufrufe würden von dieser Methode akzeptiert werden, wenn man die Aufrufe in test() umbenennen würde. Denn der Compiler wandelt die int-Argumente in Object-Instanzen um. Diese Umwandlung ist jedoch kein Cast, sondern Boxing. Dabei wird für jedes int ein echtes Objekt erzeugt, in das der elementaren Wert hineingestellt wird. Genauer wird dies als Auto-Boxing bezeichnet, weil der Compiler das Boxing automatisch anwendet, wenn es benötigt wird. Interessant ist, dass der Compiler diese folgende test()-Methode neben der obigen zulässt (Overloading):

```
public static void test(int ... array)
{
        System.out.println(array.length);
}
```

Erprobt man nun diese beiden test()-Methoden mit dem folgenden Aufruf:

```
test(new int[0]);
```

so wählt der Compiler test(int ...) und wir sehen die Ausgabe 0 auf der Konsole. Das haben wir so erwartet. Kommentieren wir nun aber diese Methode aus, so wählt der Compiler natürlich die verbleibende Methode test(Object ...) und wir sehen die Ausgabe 1 auf der Konsole. Der Compiler nimmt das Argument int[0] jetzt als Object und sieht somit ein Element des Arrays Object[], was entsprechend protokolliert wird.

Nun sind wir mit der Syntax-Erprobung der Varargs in Bereiche geraten, an die wir uns eigentlich nicht begeben wollten. Deshalb eine deutliche Warnung: Auch wenn der Compiler die beiden test()-Methoden nebeneinander akzeptiert, schreiben Sie niemals Code in dieser Art! Wenn Sie darüber grübeln müssen, welche Methode der Compiler mit einem Aufruf: *test(new int[0]);* verbindet, dann ist dies bereits schlecht. Wir wollen immer klaren und einfachen Code und keine Syntax-Gymnastik. Wenn wir hier die Syntax zu variablen Argument-Listen etwas ausgelotet haben, so nur deshalb, weil wir hin und wieder spontan dem Reiz des Unfugs nachgeben.

Methoden mit variabel langen Argumentlisten sind in Praxis auf Grund der Flexibilität, die sie bieten, nicht ganz selten (aber auch nicht häufig) und sind einfach zu verwenden. Variabel lange Argumentlisten können mit anderen Argumenten gemeinsam auftreten, aber es gibt in einer Methode maximal nur eine variabel lange Argumentliste und diese steht immer am Ende der Reihe der Argumente:

```
public static void print(String text, int … array) {… }          // Erlaubt
public static void print(int … array, String text) {… }          // Error
```

3.6. Argumentprüfung

Ein letzter Punkt zu Methoden im Allgemeinen betrifft die Argumentprüfung. Public Methoden sollten ihre Argumente prüfen, bevor sie mit ihrer eigentlichen Arbeit beginnen. Hier sehen wir, wie eine einfache Prüfung in printLongArray() aussehen könnte:

```
public static void printLongArray(long[] array, String separator)
{
        if (array == null)
                throw new NullPointerException("null is not allowed for the array argument");
```

```
        for (long element : array)                    // Ohne Prüfung fliegt die Exception hier
                System.out.print(element + separator);
}
```

Fehlerhafte Argumente werden mit Exceptions quittiert (mehr zu Exceptions in einem der folgenden Abschnitte). Die null-Überprüfung von separator kann hier unterbleiben, wenn es in Ordnung ist, dass eine für separator übergebene null in der Ausgabe als null-String erscheint. Wenn dagegen array als null übergeben und keine Prüfung durchgeführt wird, fliegt die Exception im for-Statement. Es sieht so aus, als würde es keinen Unterschied machen, wo die Exception fliegt. Tatsächlich ist der Unterschied jedoch groß. Wenn eine Exception im for-Statement auftritt, so ist dies die Schuld von printLongArray(). Fliegt die Exception in der Argumentprüfung, so ist dies die Schuld des Aufrufers. Argumentprüfungen helfen also bei Fehlerlokalisierung und Ursachenforschung. Argumentprüfung umfasst meistens mehr als nur die hier gezeigte Prüfung auf null. Public Methoden haben in der Regel eine Spezifikation, welchen Eingang sie erwarten, und sie testen ihre Argumente entsprechend diesen Vorgaben.

Wir kennen bisher nur public Methoden. Wichtig im Zusammenhang mit der Argumentprüfung ist, dass diese auf public Methoden beschränkt ist. Bei privaten Methoden, die ja nur innerhalb ihrer Klasse aufgerufen werden können, ist eine Argumentprüfung nicht notwendig beziehungsweise sinnvoll, da bei diesen Aufrufen nur die eine Klasse beteiligt ist und eine Fehlerzuordnung sich daher erübrigt.

4. Lokale Variable, Blöcke, Scope und final

Das ist gültiger Java-Code:

```
public void emptyBlock()
{
        {

        }
}
```

Ein Code-Block wird mit geschweiften Klammern eröffnet und wieder geschlossen. Er fasst in der Regel eine Folge von Statements zu einer Einheit zusammen. Und er kann überall dort vereinbart werden, wo auch ein anderes Statement geschrieben werden darf.

Blöcke sind ein elementares Mittel, um Einheiten und Strukturen zu bilden. Selbst eine Methode kann als benannter (und aufrufbarer) Block aufgefasst werden. Die folgenden Beispielzeilen, die wir schon kennen, zeigen neben dem Methodenblock zwei weitere Blöcke. Der erste wird von einem for-Statement kontrolliert, der zweite von einem if-Statement:

```
public static int printLongArray(long[] array, String separator)
{
        int lineCounter = 0;                          // Variablenvereinbarung lineCounter
        int counter = 0;                              // Variablenvereinbarung counter
        for (long element : array)                    // Variablenvereinbarung element
        {
                System.out.print(element + separator);
```

```
            if (++counter == 20)                    // Pre-Inkrement, Magic-Number
            {
                System.out.println();
                lineCounter++;                      // Post-Inkrement
                counter = 0;
            }
        }
        System.out.println();

        return lineCounter;
    }
```

Der äußere Block wird für alle Elemente des long-Arrays durchlaufen. Der innere Block wird nur dann durchlaufen, wenn die Variable counter die gegebene Bedingung erfüllt und den Wert 20 annimmt. Die Bildung von Einheiten aus aufeinander folgenden Statements, um sie gemeinsam in eine Kontrollstruktur zu stellen, ist der Hauptzweck von Blöcken. Blöcke sind ein unkompliziertes Hilfsmittel. Die einzige Komplikation betrifft den Gültigkeitsbereich von Variablen. Zudem gilt die eher triviale Regel, dass Blöcke sich nicht nur teilweise überschneiden dürfen. Blöcke können beliebig ineinander gestellt werden. Aber jeder Block muss grundsätzlich in dem umgebenden Block wieder geschlossen werden, in dem er begonnen wird.

Variable können überall in einem Block beziehungsweise in einer Methode vereinbart werden. Es gibt keinen Grund, alle Variablen, die in einer Methode benötigt werden, gleich in den ersten Zeilen zu deklarieren. Im Gegenteil, Variable sollten erst dort vereinbart werden, wo sie gebraucht werden. Variable haben einen Gültigkeitsbereich, der auch Scope genannt wird. Dieser reicht von der Zeile, in der eine Variable deklariert wird, bis zum Ende ihres Blocks (also bis zum Ende des Blocks, in dem sie vereinbart wird).

Woran erkennt man die Vereinbarung einer Variablen? Ein Variablen-Statement beginnt immer mit einer Typangabe, also mit Angaben wie long oder String, dann folgt der Name der Variablen und ein Semikolon. Zwischen Namen und Semikolon kann sich die Zuweisung eines Wertes befinden. Wenn eine Variable ohne initialen Wert vereinbart wird, hat sie bis zur ersten Zuweisung den 0-Wert ihres Typs. Bei boolean ist dies false, bei den anderen elementaren Typen 0, beziehungsweise 0.0, und bei den Referenztypen null. Häufig wird Variablen in ihrer Vereinbarung ein 0-Wert dennoch explizit zugewiesen. Dies geschieht zuallererst der Klarheit zuliebe, dann um bei späteren Bearbeitern des Codes die Unsicherheit zu vermeiden, ob die Initialisierung übersehen wurde, und schließlich um dem Compiler zuvorzukommen, der manchmal eine scheinbar vergessene Initialisierung anmahnt.

In printLongArray() haben wir fünf Variable gegeben, genau genommen sind es drei lokale Variable und zwei Argumente. Aber wir unterscheiden (hier) nicht zwischen lokalen Variablen und Argumenten, denn Argumente haben den gleichen Status wie lokale Variable (sie existieren im ablaufenden Programm nur in der Zeitspanne, in der die betreffende Methode durchlaufen wird). Lokale Variable sind Variable, die innerhalb von Methoden vereinbart werden. Die fünf lokalen Variablen im Beispiel oben sind also: array, separator, lineCounter, counter und element. Der Scope von vier dieser Variablen erstreckt sich dabei bis zum Ende von printLongArray(). Einzig element ist nur innerhalb des Blocks der for-Schleife gültig. Wir können element hinter der for-Schleife nicht mehr zugreifen. Würde man auch die Variable lineCounter innerhalb der for-Schleife vereinbaren, könnte man sie im return-Statement nicht verwenden:

```
for (long element : array)
{
    int lineCounter = 0;                    // Demonstration eines kleinen Scopes
    …
}                                           // Ende des Scopes von lineCounter
```

```
return lineCounter;                    // Error. lineCounter ist nicht sichtbar
```

Eine lokale Variable kann außer in ihrem eigenen Block auch in allen anderen Blöcken verwendet werden, die sich in ihrem eigenen Block und hinter ihrer Deklaration befinden. Für die Statements nach ihrem Block ist eine lokale Variable allerdings nicht mehr vereinbart. Dies bedeutet, dass ihr Name dort wieder verwendet werden darf. Innerhalb des Scopes einer lokalen Variablen darf eine zweite lokale Variable mit dem gleichen Namen dagegen nicht vereinbart werden. Das folgende Beispiel zeigt in Bezug auf Scope und Variablenbenennung gültigen Code:

```
for(int i = 0; i < 2;) {;}             // Der Scope von erstem i endet hier
for(int i = 0; i < 2;) {;}             // Okay? Die Namensgleichheit schon
```

Spaßigerweise meldet der Compiler für die zweite for-Schleife einen Fehler, wenn man in der ersten for-Schleife die Bedingung (i < 2) weglässt (Unreachable Code). So, wie die beiden Schleifen jetzt geschrieben sind, ist die zweite Schleife immer noch nicht erreichbar, aber der Compiler ist zufrieden. Dies erinnert nebenbei daran, dass die Fähigkeiten von Compilern differieren und sich zudem im Laufe der Zeit wandeln. Die Syntaxregeln der Sprache sind jedoch auf allen Systemen die gleichen.

Es ist möglich, Variable durch das Schlüsselwort final zu qualifizieren. Einer final Variablen wird wie anderen Variablen ein initialer Wert zugewiesen. Danach kann ihr aber kein weiterer Wert mehr zugewiesen werden. Im folgenden deklarieren wir einige der Variablen in printLongArray() als final und sehen uns an, was das bedeutet:

```
public static int printLongArray(final long[] array, final String separator)
{
    final int lineCounter;
    int counter = 0;
    lineCounter = 0;                   // Okay. Wird erst hier initialisiert
    for (final long element : array)   // Okay
    {
        System.out.print(element + separator);
        …
        lineCounter++;    …            // Error. lineCounter nicht änderbar
    }

    array[0] = 12345;                  // Okay
    return lineCounter;
}
```

Die final-Setzung von array, separator und element hat keine Auswirkungen auf den bisherigen Code, da diesen Variablen kein neuer Wert zugewiesen wird. Diese Variablen werden im Code nicht verändert. Die final-Deklaration kann man hier deshalb mit der Motivation vornehmen, genau dies auszudrücken und um zugleich diese Variablen vor einer versehentlichen Neu-Zuweisung zu schützen.

Unmittelbar vor dem return-Statement haben wir das erste Element von array verändert. Der Compiler erlaubt dies, obwohl array final ist. final-Objekte können verändert werden. final bedeutet nicht in erster Linie Konstanz, sondern dass an final-Variable nach ihrer Initialisierung nicht erneut zugewiesen werden kann. Bei Variablen mit elementarem Typ erreicht man eine Wertveränderung jedoch nur durch Zuweisung. Bei diesen Variablen ist final deshalb gleichbedeutend mit konstant.

Die final-Deklaration von lineCounter im letzten Beispiel verhindert, dass printLongArray() weiterhin korrekt funktioniert, denn der Zähler kann nun nicht mehr verändert werden. Das Beispiel zeigt zudem, dass der Compiler schlau genug ist, die Initialisierung, die erst zwei Zeilen nach der Deklaration erfolgt, zu erlauben, während er eine weitere Zuweisung ablehnt. Betrachten wir nun eine weitere Variable in printLongArray(), die wir ebenfalls als final vereinbaren:

```
public static int printLongArray(long[] array, String separator)
{
    final int ITEMS_PER_LINE = 20;                    // Okay. Aber nicht wirklich gut
    …
    if (counter == ITEMS_PER_LINE) {…} …
}
```

Da final Variable mit elementarem Typ in keiner Weise mehr verändert werden können, können wir sie auch als Konstanten auffassen. Nach einer verbreiteten Konvention, die man tunlichst beachten sollte, sind Konstanten, wie hier zu sehen, an ihrer vollständigen Großschreibung zu erkennen. Warum aber sollte man die Konstante ITEMS_PER_LINE überhaupt verwenden? Hierzu gibt es zwei gute Gründe. Der erste ist, dass wir keine Magic-Numbers wollen. Ein Code wie dieser:

```
if (counter == 20) {…}                                // Magic-Number. Vermeiden!
```

kontrolliert den zugeordneten Block mit einer zufällig oder magisch erscheinenden Zahl, für die wir an dieser Stelle nicht erkennen können, warum genau diese Größe gewählt wurde. Hier ist die Zahl 20 nicht so entscheidend. Übertragen Sie das Ganze jedoch einmal in Bereiche aus Ihrem Arbeitsumfeld. Irgendwo in den Tiefen des Codes taucht plötzlich die Zahl 141 auf. Warum genau 141, warum nicht 140 oder 130? Mit ‚magic' soll ausgedrückt werden, dass ein bestimmter Wert (Zahlenwert) zwar begründet sein kann, dass wir aber den Zusammenhang nicht sehen können. Der zweite Grund ist, dass die Konstante bereits durch ihren Namen den Zweck des if-Statements beschreibt, der mit einer bloßen Zahl so nicht zu erkennen ist. Für uns ist im Beispiel bereits klar, was an dieser Stelle kontrolliert wird und welchen Zweck die betreffende Zahl hat. In Codeteilen, die Ihnen neu sind, muss das keineswegs so sein. Die Verwendung einer Konstante statt einer bloßen Zahl trägt dazu bei, dass Code besser verstanden werden kann.

Die Ersetzung von Magic-Numbers durch Konstanten weist in die Richtung eines Coding-Stils, der in diesem Buch sehr nachdrücklich propagiert wird. Wir wollen nach Möglichkeit flüssig lesbaren Code, der ohne weitere Zutaten für sich alleine verständlich ist. Dazu gehören sorgfältig gewählte Namen und Bezeichner und Formulierungen wie diese:

```
if (counter == ITEMS_PER_LINE) {…}
```

Was wir nicht wollen, ist ein Code, der wie dieser aussieht:

```
if(ctr == 20)
    ftp();
```

Nun gehen wir noch einmal einige Zeilen zurück: Was stört uns oben an der Deklaration von ITEMS_PER_LINE? Was hier nicht in Ordnung ist, ist der Ort, an der die Konstante vereinbart wird. Wir möchten im Allgemeinen die Parameter, die irgendeinen Aspekt der Applikation bestimmen, nicht irgendwo im Code verstreut, sondern an zentraler Stelle versammelt haben. Für die Konstante ITEMS_PER_LINE wäre es deshalb besser, wenn sie nicht innerhalb einer Methode vereinbart wäre, sondern an einer mehr hervorge-

hobenen Stelle. Darauf werden wir wieder zurückkommen, wenn wir mehr über die Möglichkeiten von Klassen und Interfaces wissen.

Eine gute Alternative für printLongArray() wäre es, die Einstellung der auszugebenden Elemente pro Zeile über ein Argument festzulegen:

```
public static int printLongArray(final long[] array, final String separator, final int itemsPerLine) { …}
```

Diese Alternative bewahrt uns allerdings nicht vor der Aufgabe, (an anderer Stelle) darüber nachzudenken, wo man am besten Konstanten vereinbart. Denn in der Praxis ist man oft in der Situation, dass mehrere überladene Versionen einer Methode nebeneinander bestehen. Die eine wird über Argumente konfiguriert, die anderen arbeiten mit Defaultwerten. Und Defaultwerte wiederum sind Konstanten, die irgendwo vereinbart sein müssen:

```
public static int printLongArray(final long[] array, final String separator, final int itemsPerLine) { …}
public static int printLongArray(final long[] array, final String sep) { …}     // Eventuell mit Defaultwert
public static int printLongArray(final long[] array) { …}                        // Eventuell mit Defaultwerten
```

Dass im Normalfall Konstanten nicht innerhalb von Methoden deklariert werden sollen, hat auch mit einer grundsätzlichen Haltung gegenüber Methoden zu tun. Wenn Methoden wie printLongArray() erst einmal erstellt sind, wollen wir uns nicht mehr mit ihrer Implementierung befassen. Wir sehen ihrer Deklaration an, was sie leisten, und wir verwenden sie entsprechend. Aber wir kramen nicht mehr in ihren Implementierungen herum und drehen dort an Knöpfen und Konstanten. Eine Methode wird zusammen mit ihren Argumenten so verständlich formuliert, dass sie problemlos verwendet werden kann und dass es keinen weiteren Bedarf mehr gibt, sich mit ihrem Innenleben auseinander zu setzen.

Und schließlich noch ein letzter Punkt zu Codeblöcken. Bei Blöcken, die nur ein einziges Statement umfassen wie dieser:

```
for (long element : array)
{
        System.out.print(element);
}
```

können die geschweiften Klammern auch weggelassen werden:

```
for (long element : array)
        System.out.print(element);
```

Dieses Weglassen der geschweiften Klammern nutzen wir häufig, um Code kompakt zu schreiben.

5. Die Grundregel für zusammengesetzte Ausdrücke

Der folgende Codeabschnitt zeigt die Grundregel zum Aufbau von Ausdrücken aus Teilausdrücken:

```
int i = 0;
boolean test = (i = getNumber()) > 0;                // Kann in die unten stehenden Teile aufgelöst werden
```