Robert Sedgewick
Kevin Wayne

Einführung in die Programmierung mit Java

ALWAYS LEARNING
PEARSON

definiert main() als eine statische Methode, die ein Array von Strings als Argument übernimmt und nichts zurückliefert. Per Konvention sammelt das Java-System die Strings, die Sie nach dem Programmnamen im java-Befehl eintippen, in einem Array namens arg[ ] und ruft main() mit diesem Array als Argument auf. (Die meisten Programmierer verwenden den Namen args für die Argumentvariable, obwohl eigentlich jeder Name den gleichen Zweck erfüllen würde.) In main() können wir dieses Array dann so wie jedes andere Array verwenden. Die main()-Methode aus *Listing 1.13* (Newton) ist ein Beispiel für solchen Code.

Nebeneffekte bei Arrays

In vielen Fällen besteht die eigentliche Aufgabe von statischen Methoden, die ein Array als Argument übernehmen, darin, einen Nebeneffekt zu produzieren (die Werte von Arrayelementen zu ändern). Ein typisches Beispiel hierfür ist eine Methode, die die Werte zweier gegebener Indizes in einem gegebenen Array vertauscht. Wir können den Code, den wir zu Beginn von *Kapitel 1.4* betrachtet haben, folgendermaßen umschreiben:

```
public static void exch(String[ ] a, int i, int j)
{
  String t = a[i];
  a[i] = a[j];
  a[j] = t;
}
```

Diese Implementierung ergibt sich nahezu automatisch aus der Art und Weise, wie Arrays in Java repräsentiert werden. Die Argumentvariable von exch() ist ein Verweis auf das Array – und nicht etwa auf alle im Array enthaltenen Werte: Wenn Sie einer statischen Methode ein Array als Argument übergeben, ermöglichen Sie es der Methode, auf dem Array selbst (und nicht etwa auf einer Kopie des Arrays) zu operieren. Ein zweites typisches Beispiel für eine statische Methode, die ein Arrayargument übernimmt und Nebeneffekte erzeugt, ist eine Methode, die die Werte in einem Array mischt (in eine zufällige Reihenfolge bringt) und dazu die Version des Algorithmus verwendet, die wir in *Kapitel 1.4* betrachtet haben (sowie die hier definierten Methoden exch() und uniform()).

```
public static void shuffle(String[ ] a)
{
  int N = a.length;
  for (int i = 0; i < N; i++)
    exch(a, i, i + uniform(N-i));
}
```

Als Ergänzung werden wir in *Kapitel 4.2* Methoden betrachten, die Arrays sortieren (d.h. die Werte des übergebenen Arrays in eine geordnete Reihenfolge bringen). Alle diese Beispiele belegen, dass der Mechanismus zur Übergabe von Arrays in Java hinsichtlich des Arrayinhalts ein **Call-by-Reference**-Mechanismus ist. Im Gegensatz zu den Argumenten eines primitiven Typs sind daher die Änderungen, die eine Methode am Inhalt eines Arrays vornimmt, im Client-Programm zu sehen. Methoden, die ein Array als Argument übernehmen, können das Array selbst nicht ändern – der Verweis zeigt auf

dieselbe Speicherposition, die bei der Arrayerzeugung zugeteilt wurde, und die Länge ist der Wert, der bei der Erzeugung des Arrays festgelegt wurde – aber sie können den Inhalt des Arrays auf beliebige Werte setzen.

Typischer Code für die Implementierung von Funktionen mit Arrays

Den maximalen Arraywert ermitteln	`public static double max(double[ ] a)`
	`{`
	`  double max = Double.NEGATIVE_INFINITY;`
	`  for (int i  = 0; i < a.length; i++)`
	`  if (a[i] > max) max = a[i];`
	`  return max;`
	`}`

Skalarprodukt	`public static double dot(double[ ] a, double[ ] b)`
	`{`
	`  double sum = 0.0;`
	`  for (int i  = 0; i < a.length; i++)`
	`  sum += a[i] * b[i];`
	`  return sum;`
	`}`

Zwei Elemente in einem Array vertauschen	`public static void exch(String[ ] a, int i, int j)`
	`{`
	`  String t = a[i];`
	`  a[i] = a[j];`
	`  a[j] = t;`
	`}`

Ein eindimensionales Array (und seine Länge) ausgeben	`public static void print(double[ ] a)`
	`{`
	`  StdOut.println(a.length);`
	`  for (int i = 0; i < a.length; i++)`
	`  StdOut.println(a[i]);`
	`}`

Ein zweidimensionales Array von double-Werten (mit Dimensionen) in Row-Major-Anordnung (Zeile zuerst) einlesen	`public static double[ ][ ] readDouble2D()`
	`{`
	`  int M = StdIn.readInt();`
	`  int N = StdIn.readInt();`
	`  double[ ][ ] a = new double[M][N];`
	`  for (int i  = 0; i < M; i++)`
	`  for (int j = 0; j < N; j++)`
	`  a[i][j]  = StdIn.readDouble();`
	`  return a;`
	`}`

Arrays als Rückgabewerte

Methoden, die die Werte eines Arrays, das als Argument übernommen wurde, sortieren, umstellen oder anderweitig ändern, müssen keinen Verweis auf das Array zurückliefern, da sie ja direkt den Inhalt des Client-Arrays ändern (und nicht etwa den Inhalt einer Kopie). Es gibt aber auch viele Situationen, in denen es nützlich ist, wenn die statische Methode ein Array als Rückgabewert zurückliefert. Hierzu gehören vor allem statische Methoden, die Arrays erzeugen, um mehrere Werte desselben Typs an einen Client zurückzuliefern. So erzeugt zum Beispiel die folgende statische Methode ein Array, wie es von StdAudio (siehe *Listing 1.27*) verwendet wird. Die im Array gespeicherten Werte stammen von einer Sinuswelle einer gegebenen Frequenz (in Hertz) und Dauer (in Sekunden), die mit einer Samplingrate von 44100 Werten pro Sekunde abgetastet wurde.

```
public static double[ ] tone(double hz, double t)
{
  int sps = 44100;
  int N = (int) (sps * t);
  double[ ] a = new double[N+1];
  for (int i = 0; i <= N; i++)
    a[i] = Math.sin(2 * Math.PI * i * hz / sps);
  return a;
}
```

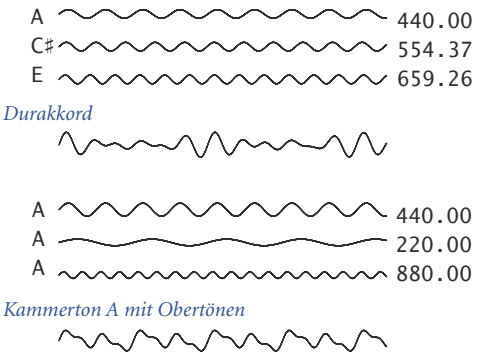
In diesem Beispiel hängt die Größe des zurückgelieferten Arrays von der Dauer ab: Wenn die gegebene Dauer t beträgt, ist die Größe des Arrays ungefähr 44100*t. Mit statischen Methoden wie dieser können wir Code schreiben, der eine Schallwelle als eine Einheit behandelt (ein Array mit den abgetasteten Werten), siehe das nachfolgende Beispiel mit *Listing 2.4*.

2.1.6 Beispiel Überlagerung von Schallwellen

Wie bereits in *Kapitel 1.5* angesprochen, bedarf das einfache Audio-Beispiel, das wir dort betrachtet haben, noch einiger Verfeinerungen, um einen Ton zu erzeugen, der realistischer klingt, d.h. eher wie der Ton eines Musikinstruments. Um dieses Ziel zu erreichen, gibt es viele Möglichkeiten der Verbesserung. Mit statischen Methoden können wir sie systematisch anwenden, um Schallwellen zu erzeugen, die weitaus komplizierter sind als die einfachen Sinuswellen aus *Kapitel 1.5*. Um den effektiven Einsatz von statischen Methoden zur Lösung eines interessanten Rechenproblems zu veranschaulichen, betrachten wir ein Programm, das im Wesentlichen die gleiche Funktionalität aufweist wie *Listing 1.27* (PlayThatTune), aber zusätzlich harmonische Schwingungen eine Oktave über und unter jedem Ton ergänzt, um einen realistischeren Klang zu erzeugen.

Akkorde und Obertöne

Töne wie der Kammerton A haben einen reinen Klang, der nicht sehr musikalisch klingt, da in den Tönen, an die Sie gewöhnt sind, normalerweise viele andere Töne mitschwingen. Der Klang der Gitarrensaiten hallt beispielsweise vom Holz des Instruments wider oder von den Wänden des Raums, in dem Sie sich befinden. Sie können sich solche Effekte als Schwingungen vorstellen, die die grundlegende Sinuswelle modifizieren. Die meisten Musikinstrumente erzeugen zum Beispiel Obertöne (d.h. gleiche Töne, aber in verschiedenen Oktaven und nicht so laut), oder Sie wollen vielleicht Akkorde (d.h. mehrere Töne gleichzeitig) spielen. Um mehrere Töne zu kombinieren, arbeiten wir mit **Überlagerung**: Wir addieren einfach die jeweiligen Wellen und skalieren dann neu, um sicherzustellen, dass alle Werte weiterhin zwischen −1 und +1 liegen. Wie sich herausstellt, können wir, indem wir Sinuswellen verschiedener Frequenzen auf diese Weise überlagern, beliebig komplizierte Wellen erzeugen. Tatsächlich war es einer der größten Triumphe der Mathematik des 19. Jahrhunderts, als man herausfand, dass jede glatte periodische Funktion als eine Summe von Sinus- und Kosinuswellen, einer sogenannten **Fourierreihe**, ausgedrückt werden kann – der mathematische Beleg für die alltägliche Erfahrung, dass wir mit Musikinstrumenten oder unseren Stimmbändern eine große Bandbreite von Tönen erzeugen können und dass alle Töne aus einer Zusammensetzung von mehreren oszillierenden Kurven bestehen. Jeder Klang entspricht einer Kurve und jede Kurve entspricht einem Klang, und durch Überlagerung können wir beliebig komplexe Kurven erzeugen.



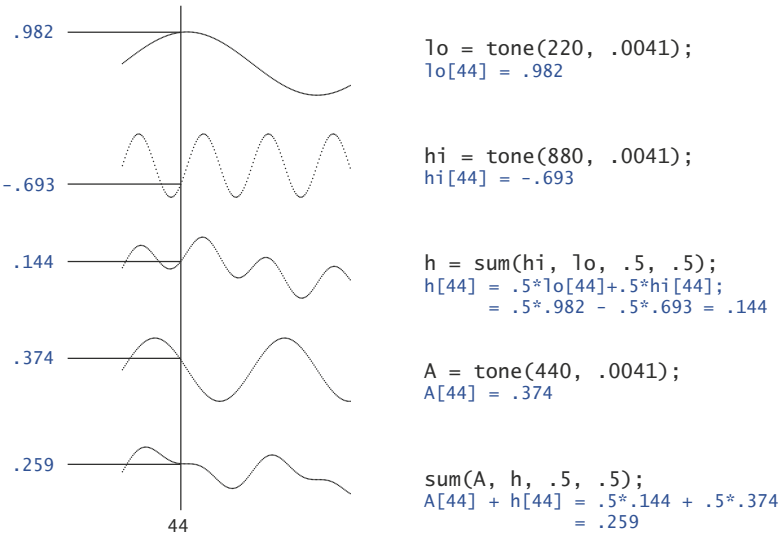
Überlagerung von Wellen zur Erzeugung zusammengesetzter Töne

Überlagerung

Da wir Schallwellen durch Zahlenarrays darstellen, die die Schallwellenwerte an den jeweils gleichen Abtastpunkten speichern, ist die Überlagerung einfach zu implementieren: Wir addieren einfach an jedem Abtastpunkt die abgetasteten Werte und erhalten so das zusammengesetzte Ergebnis, das wir dann neu skalieren. Um mehr Einfluss auf die Überlagerung nehmen zu können, weisen wir den beiden zu addierenden Wellen relative Gewichte zu, die positiv sein müssen und in der Gesamtsumme 1 ergeben. Wollen wir dann zum Beispiel erreichen, dass der erste Ton einen dreimal stärkeren Effekt hat als der zweite, brauchen wir nur dem ersten ein Gewicht von 0,75 und dem zweiten ein Gewicht von 0,25 zuzuweisen. Und wie sieht der zugehörige Code aus? Wenn die eine Welle durch ein Array a[ ] mit dem relativen Gewicht awt gegeben ist und die andere Welle durch ein Array b[ ] mit dem relativen Gewicht bwt, berechnen wir deren gewichtete Summe durch folgenden Code:

```
double[ ] c = new double[a.length];
for (int i = 0; i < a.length; i++)
  c[i] = a[i]*awt + b[i]*bwt;
```

Die Bedingung, dass die Gewichte positiv sind und sich zu 1 aufaddieren, sorgt dabei dafür, dass unser Grundsatz, nach dem die Werte all unserer Wellen zwischen −1 und +1 liegen müssen, durch das Aufaddieren nicht verletzt wird.

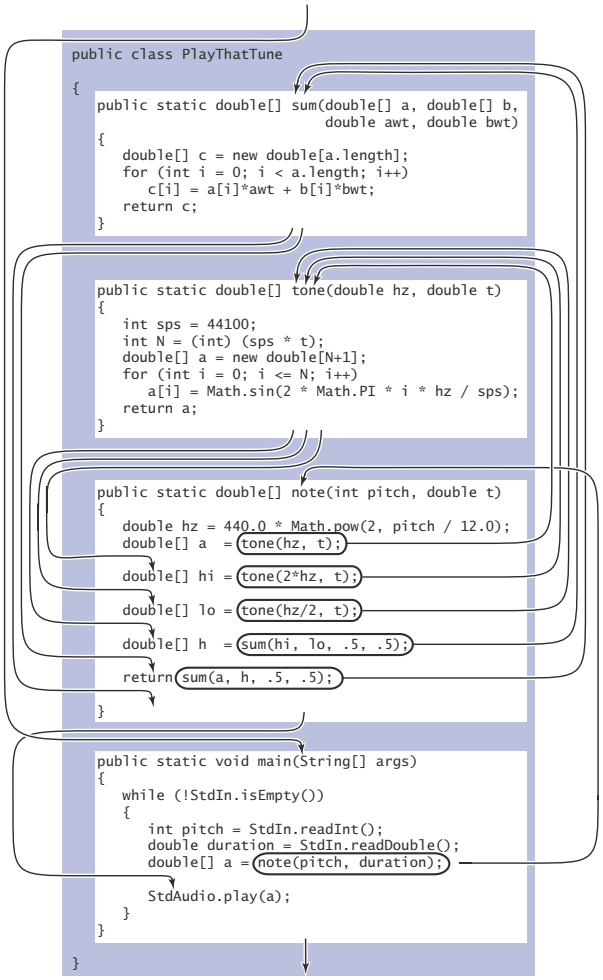


Hinzufügen von Obertönen zum Kammerton A (180 Samples bei 44100 SPS)

Listing 2.4 ist eine Implementierung, die mithilfe der dargestellten Konzepte einen wesentlich realistischeren Klang erzeugt als *Listing 1.27*. Funktionen helfen, die Berechnung in vier Teile zu zerlegen:

- Aus einer gegebenen Frequenz und Dauer einen reinen Ton (Sinuston) erzeugen.
- Aus zwei gegebenen Schallwellen und relativen Gewichten die Überlagerung der beiden Wellen berechnen.
- Aus einer gegebenen Tonhöhe und -dauer einen Ton mit Obertönen erzeugen.
- Eine Folge von Tonhöhe/Tondauer-Paaren über die Standardeingabe einlesen und abspielen.

Diese Aufgaben eignen sich bestens für die Implementierung als eine Gruppe aufeinander aufbauender Funktionen. Jede einzelne Funktion ist dabei wohl definiert und leicht zu implementieren. Allen Funktionen (inklusive StdAudio) ist gemein, dass sie Sound darstellen als eine Folge von diskreten Werten, die in einem Array gespeichert sind und dem Abtasten einer Schallwelle mit 44100 SPS (Samples pro Sekunde) entsprechen.



Ablauf der Programmausführung unter mehreren statischen Methoden

Bisher brachte uns der Einsatz von Funktionen vor allem eine besser zu handhabende Notation. *Listing 2.1* bis *Listing 2.3* weisen zum Beispiel einen ganz einfachen Programmablauf auf – jede Funktion wird im Code nur an einer Stelle aufgerufen. Im Vergleich dazu ist *Listing 2.4*, in dem die Funktionen jeweils mehrmals aufgerufen werden, ein überzeugendes Beispiel dafür, wie effektiv es sein kann, Funktionen zur Strukturierung von Berechnungen zu definieren. Nehmen wir zum Beispiel die Funktion note(), welche die Funktion tone() drei Mal und die Funktion sum() zwei Mal aufruft. Ohne statische Methoden bräuchten wir mehrere Kopien des Codes in tone() und sum(); mit statischen Methoden können wir mit Konzepten arbeiten, die der Anwendungsdomäne nahe sind. Ebenso wie Schleifen haben Methoden eine einfachen aber sehr wirkungsvollen Effekt: Sie bündeln eine Folge von Anweisungen (diejenigen in der Methodendefinition), die zur Laufzeit des Programms mehrmals ausgeführt wird – und zwar ein Mal für jeden Aufruf der Methode im Zuge der Ausführung von main().

Listing 2.4: PlayThatTune (überarbeitet)

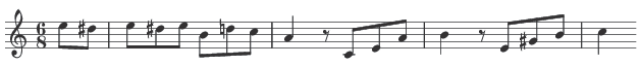
```
public class PlayThatTune
{
  public static double[ ] sum(double[ ] a, double[ ] b,
                              double awt, double bwt)
  {  // Ueberlagert a und b, gewichtet.
     double[ ] c = new double[a.length];
     for (int i      = 0; i < a.length; i++)
        c[i] = a[i]*awt + b[i]*bwt;
     return c;
  }

  public static double[ ] tone(double hz, double t)
     // siehe Text

  public static double[ ] note(int pitch, double t)
  {  // Spielt Ton einer gegebenen Tonhoehe mit Obertoenen ab.
     double hz   = 440.0 * Math.pow(2, pitch / 12.0);
     double[ ] a  = tone(hz, t);
     double[ ] hi = tone(2*hz, t);
     double[ ] lo = tone(hz/2, t);
     double[ ] h  = sum(hi, lo, .5, .5);
     return sum(a, h, .5, .5);
  }

  public static void main(String[ ] args)
  {  // Liest eine Melodie mit Obertoenen ein und spielt sie ab.
     while (!StdIn.isEmpty())
     {  // Liest einen Ton mit Obertoenen ein und spielt ihn ab.
        int pitch = StdIn.readInt();
        double duration = StdIn.readDouble();
        double[ ] a = note(pitch, duration);
        StdAudio.play(a);
     }
  }
}
```

hz	Frequenz
a[ ]	Sinuston (reiner Ton)
hi[ ]	oberer Oberton
lo[ ]	unterer Oberton
h[ ]	Ton mit Obertönen

Dieser Code verbessert die Klangfarbe des Tons, der von Listing 1.27 erzeugt wird, indem er mithilfe statischer Methoden Obertöne erzeugt, die einen realistischeren Klang erzeugen als die reinen Sinustöne.

```
% more elise.txt
7 .25
6 .25
7 .25
6 .25
7 .25
...
```

```
% java PlayThatTune < elise.txt
```



Statische Methoden sind wichtig, weil sie uns erlauben, die Möglichkeiten der Programmiersprache Java in unseren Programmen *zu erweitern*. Nachdem wir statische Methoden wie sqrt(), phi(), Phi(), mean(), abs(), exch(), shuffle(), isPrime(), H(), uniform(), sum(), note() und tone() implementiert und sorgfältig getestet haben, können wir sie fast genauso verwenden, wie wenn sie in Java integriert wären. Diese Flexibilität erschließt uns eine völlig neue Welt der Programmierung. Vorher reichte es, sich ein Java-Programm als eine Folge von Anweisungen vorzustellen. Jetzt müssen Sie sich ein Java-Programm als einen **Satz statischer Methoden** vorstellen, die sich gegenseitig aufrufen können. Innerhalb der statischen Methoden gibt es noch den vertrauten Ausführungsablauf von Anweisung zu Anweisung, der Ablauf von Programmen jedoch findet auf einer höheren Ebene statt und wird von den Aufrufen der statischen Methoden und ihren Rückgabewerten definiert. Dies erlaubt es uns, in Form von Operationen zu denken, die von der Anwendung angefordert werden, und nicht mehr nur in den einfachen arithmetischen Operationen auf primitiven Typen, die in Java vorgegeben sind.

Wann immer sich ein Programm klar in Teilaufgaben zerlegen lässt, sollten Sie dies tun. Die Beispiele in diesem Abschnitt (und die Programme im ganzen Rest des Buches) zeigen deutlich, welche Vorteile es hat, an dieser Maxime festzuhalten. Mit statischen Methoden können wir:

- Eine lange Folge von Anweisungen in kleinere, voneinander unabhängige Teile zerlegen.
- Code wiederverwenden, ohne ihn kopieren zu müssen.
- Mit höheren Konzepten (wie Schallwellen) arbeiten.

Auf diese Weise erzeugen Sie Code, der leichter zu verstehen, warten und debuggen ist als ein langes Programm, das nur aus Java-Zuweisungen, -Bedingungen und -Schleifen besteht. Im nächsten Abschnitt geht es dann darum, wie man statische Methoden verwendet, die in *anderen Programmen* definiert wurden – wodurch wir eine noch höhere Stufe der Programmierung erreichen.

Fragen & Antworten

Frage: Warum muss ich den Rückgabetyp void verwenden? Warum kann ich diesen Rückgabetyp nicht einfach weglassen?

Antwort: Wir müssen diesen Rückgabetyp angeben, weil seine Angabe in Java zwingend erforderlich ist. Im Übrigen gilt: Wer eine Entscheidung, die von dem Entwickler einer Programmiersprache getroffen wurde, bezweifelt, befindet sich schon so gut wie auf dem Weg, selbst einer zu werden.

Frage: Kann ich aus einer void-Funktion mithilfe von return zurückkehren? Und wenn ja, welchen Rückgabewert soll ich verwenden?

Antwort: Ja. Verwenden Sie die Anweisung return; ohne Rückgabewert.

Frage: Was passiert, wenn ich das Schlüsselwort static vergesse?

Antwort: Wie schon erwähnt, lassen sich Fragen wie diese am besten dadurch beantworten, dass man es einfach ausprobiert und schaut, was passiert. Für den Fall, dass in Newton das Schlüsselwort static für sqrt() weggelassen würde, ergäbe sich zum Beispiel Folgendes:

```
Newton.java:13: non-static method sqrt(double)
cannot be referenced from a static context
        double x = sqrt(i);
                   ^
1 error
```

Nicht-statische Methoden unterscheiden sich von statischen Methoden. Erstere werden Sie in *Kapitel 3* kennenlernen.

Frage: Was passiert, wenn ich nach einer return-Anweisung noch weiteren Code vorsehe?

Antwort: Sobald eine return-Anweisung erreicht ist, geht die Kontrolle direkt wieder zurück an den Aufrufer, sodass jeder Code nach der return-Anweisung sinnlos ist. Der Java-Compiler identifiziert diese Situation als einen Fehler und meldet: unreachable code.

Frage: Was passiert, wenn ich keine return-Anweisung verwende?

Antwort: Kein Problem, wenn der Rückgabetyp void ist. In diesem Fall geht die Kontrolle automatisch nach Ausführung der letzten Anweisung zurück an den Aufrufer. Wenn der Rückgabetyp nicht void ist, meldet der Compiler einen Fehler (a missing return statement), falls er auf *irgendeinen* Pfad durch den Code stößt, der nicht mit einem return endet.

Fragen & Antworten

Frage: Das Thema „Nebeneffekte und Arrays, die als Argumente übergeben werden“ ist ziemlich verwirrend. Ist es wirklich so wichtig?

Antwort: Ja. Bei der Arbeit an größeren Systemen gehört es zu den wichtigsten Aufgaben des Programmierers, Kontrolle und Übersicht über die erzeugten Nebeneffekte zu behalten. Nehmen Sie sich also die Zeit und vergewissern Sie sich, dass Sie den Unterschied zwischen der Übergabe als Wert (wenn die Argumente primitiven Datentypen angehören) und der Übergabe als Referenz (wenn die Argumente Arrays sind) verstanden haben. Der Aufwand lohnt sich! Letzterer Mechanismus wird übrigens auch für alle anderen Datentypen verwendet – wie Sie in *Kapitel 3* erfahren werden.

Frage: Warum eliminieren wir nicht einfach die Möglichkeit von Nebeneffekten, indem wir alle Argumente, einschließlich der Arrays als Werte übergeben?

Antwort: Stellen Sie sich einfach einmal ein riesiges Array vor mit, sagen wir, Millionen von Elementen. Ist es in diesem Fall sinnvoll, alle diese Werte für eine statische Methode zu kopieren, die nichts anderes macht, als zwei davon zu vertauschen? Aus diesem Grund unterstützen die meisten Programmiersprachen die Übergabe von Arrays an Funktionen ohne Erzeugung einer Kopie der Arrayelemente (MATLAB ist in diesem Punkt eine bemerkenswerte Ausnahme).

2.1 Allgemeine Übungen

1 Schreiben Sie eine statische Methode max3(), die drei int-Werte als Argumente übernimmt und den Wert des größten Integers zurückliefert. Fügen Sie eine überladene Funktion hinzu, die das gleiche mit drei double-Werten macht.

2 Schreiben Sie eine statische Methode odd(), die drei Eingaben vom Typ boolean übernimmt. Wenn eine ungerade Anzahl der Eingaben true ist, soll die Methode true zurückliefern, ansonsten false.

3 Schreiben Sie eine statische Methode majority(), die drei Argumente vom Typ boolean übernimmt. Wenn mindestens zwei der Argumente den Wert true haben, soll die Methode true zurückliefern, ansonsten false. Verwenden Sie keine if-Anweisung.

4 Schreiben Sie eine statische Methode eq(), die zwei Integer-Arrays als Argumente übernimmt und true zurückliefert, wenn beide die gleiche Anzahl an Elementen enthalten und alle Elemente paarweise gleich sind.

5 Schreiben Sie eine statische Methode areTriangular(), die drei double-Werte als Argumente übernimmt und true zurückliefert, wenn die Werte die Seiten eines Dreiecks bilden könnten (d.h. kein Wert ist größer oder gleich der Summe der beiden anderen Werte). Siehe Übung 1.2.15.

6 Schreiben Sie eine statische Methode sigmoid(), die ein double-Argument x übernimmt und den double-Wert zurückliefert, der sich nach der Formel $1/(1-e^{-x})$ berechnet.

7 Wenn das Argument von sqrt() in *Listing 2.1* (Newton) den Wert Infinity hat, liefert Newton.sqrt() wie gewünscht den Wert Infinity zurück. Erklären Sie warum.

8 Ergänzen Sie *Listing 2.1* (Newton) um eine Methode abs(), ändern Sie sqrt() so, dass abs() anstelle von Math.abs() verwendet wird, und fügen Sie, wie im Text beschrieben, Ausgabeanweisungen zur Nachverfolgung der Funktionsaufrufe hinzu. *Hinweis*: Sie müssen jede Funktion um ein zusätzliches Argument erweitern, das die Ebene der Einrückung angibt.

9 Verfolgen Sie die Funktionsaufrufe für java Newton 4.0 9.0.

10 Schreiben Sie eine statische Methode lg(), die einen double-Wert N als Argument übernimmt und den Logarithmus von N zur Basis 2 zurückliefert. Sie können hierfür die Java-Bibliothek Math verwenden.

11 Schreiben Sie eine statische Methode lg(), die einen int-Wert N als Argument übernimmt und den größten int-Wert zurückliefert, der nicht größer ist als der Logarithmus von N zur Basis 2. Verwenden Sie *nicht* Math.

12 Schreiben Sie eine statische Methode signum(), die einen int-Wert N als Argument übernimmt und −1 zurückliefert, wenn N kleiner als 0 ist, 0, wenn N gleich 0 ist, und +1, wenn N größer als 0 ist.

Copyright