Kapitel 05: Arrays

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Bisher haben wir Variablen verwendet, um einzelne Datenwerte in Programmen zu repräsentieren. Um eine größere Anzahl an Werten zu verarbeiten, werden diese in Arrays kompakt repräsentiert. Wir werden in dieser Vorlesung das Deklarieren, Initialisieren, Lesen und Schreiben von Arrays kennenlernen. In diesem Zusammenhang führen wir außerdem das Konzept der `for`-Schleife ein, da dieser Schleifentyp häufig bei der Programmierung mit Arrays angewendet wird. Außerdem erläutern wir den Begriff Referenzvariablen. Dieser Abschnitt schließt mit einer kurzen Einführung in das Lesen und Schreiben auf Ein- und Ausgabeströme von Daten.

Lernziele dieses Kapitels:

- Ein- und mehrdimensionale Arrays zur Repräsentierung vieler Werte gleichen Typs in einem Programm zu verwenden.
- Verschiedene Schleifenkonstrukte zu unterscheiden und in geeignetem Kontext anzuwenden.
- Semantik von Referenzvariablen und Aliasen zu erläutern.
- Einfache wahrscheinlichkeitstheoretische Fragestellungen durch Simulation zu lösen.
- Dateineingabe und -ausgabe über die Standardeingabe und -ausgabe zu realisieren.

1 Arrays

Kapitel 1.4 aus R. Sedgewick, K. Wayne: Einführung in die Programmierung mit Java. 2011, Pearson Studium.

Arrays werden zur Speicherung und Verarbeitung größerer Datenmengen verwendet. Sie ermöglichen es mehrere Werte vom gleichen Typ kompakt zu adressieren und zu verwalten.

Definition Ein ***Array (dt. Feld)*** ist ein Behälter für eine feste Anzahl von Werten eines bestimmten Typs. Die Länge eines Arrays wird bei dessen Erstellung festgelegt und kann danach nicht mehr geändert werden. Auf die Komponenten (auch: Elemente) eines Arrays wird über Indizes zugegriffen.

Die genaue Syntax und Semantik von Arrays ist von Sprache zu Sprache verschieden. Wir betrachten hier zunächst die Umsetzung von Arrays in Java.

Mit `T[]` wird in Java der Typ von Arrays mit Komponenten vom Typ `T` bezeichnet.

Syntax für Array-Typen :

```
Typ → Typ []
```

1.1 Deklaration und Initialisierung von Arrays

Die Deklaration einer Variablen `a` für ein Array mit Elementen vom Typ `T` erfolgt wie gewohnt durch eine Deklarationsanweisung. Um beispielsweise eine Variable `a` für ein Array mit `int`-Werten zu deklarieren, schreiben wir:

```
int[] a;
```

Diese Art von Variable wird auch ***Referenzvariable*** genannt. Die Deklaration stellt den Speicherplatz *für die Variable* bereit, mit der das Array referenziert wird. Die Deklaration einer Variable allein erzeugt noch kein Arrayobjekt!

Syntax für das Erstellen von Arrays :

```
Ausdruck →
    new ArrayElementTyp [ Ausdruck ]
```

`ArrayElementTyp` → `PrimitiverTyp` | ≪ *Bezeichner* ≫

Ein neues Arrayobjekt mit n Komponenten vom Typ `int` wird von dem Ausdruck

```
new int[n]
```

erzeugt. `n` ist dabei ein Ausdruck, der zu einem ganzzahligen Wert auswertet. Der Ausdruck `new int[n]` liefert eine Referenz auf das neu erzeugte Array als Ergebnis. Analog gehen wir vor, wenn wir Arrays für Werte von anderen Typen erstellen wollen (z.b. `new double[n]`, `new String[n]`).
Die Komponenten des Arrays sind zunächst mit dem Initialwert des jeweiligen Typs initialisiert.

- Für alle numerischen Datentypen ist der Initialwert 0 bzw. 0.0.
- Für `boolean` ist er `false`.
- Für `String`s und andere nicht-primitive Typen ist er `null`.

Beispiel Folgende Anweisung erzeugt ein Array der Größe `m` mit Zahlen vom Typ `double`, die alle zunächst mit `0.0` initialisiert sind und dann auf `1.0` gesetzt werden.

```
int m = ...;
double[] a = new double[m];

// Veraendern der Array-Inhalte
int i = 0;
while (i < m) {
  a[i] = 1.0;
  i = i + 1;
}
```

Sei im Folgenden `exp` ein Ausdruck vom Typ `int`, der sich zu einem Wert k auswertet.

Ausdrücke zum Lesen und Schreiben eines Arrays `a`:

- `a.length` liefert die Anzahl der Arraykomponenten.
- `a[exp]` ist als

  R-Wert: der in der k-ten Arraykomponente gespeicherte Wert;

  L-Wert: die k-te Arraykomponente,
  d.h. z.B. wird durch die Anweisung `a[exp] = 7;` der k-ten Arraykompente der Wert 7 zugewiesen.

Bemerkungen:

- Das erste Array-Element eines Arrays `a` ist mit Index 0 indiziert (`a[0]`), das zweite an Index 1 (`a[1]`), usw.
- Das letzte Element ist an Position `a.length-1` zu finden.
- Bei Zugriff auf ein Array-Element muss sichergestellt sein, dass der Indexwert zwischen 0 und `a.length-1` ist. Andernfalls passiert ein `ArrayIndexOutOfBounds`-Fehler, der die reguläre Ausführung des Programms abbricht.

Syntax zum Lesen und Schreiben von Arrays :

```
Ausdruck →
      Ausdruck . length
    | Ausdruck [ Ausdruck ]

Zuweisung → Ausdruck [ Ausdruck ] = Ausdruck;
```

1.2 Einschub: for-Anweisung (Zählanweisung)

Die for-Anweisung wird häufig zum Iterieren über Arrays verwendet. Wir diskutieren sie daher in diesem Kontext.
Syntax in Java:

`Anweisung` →
`for (forInit; Ausdruck; forUpdate) Anweisung`

`forInit` → `Typ` ≪ *Bezeichner* ≫ `=` `Ausdruck`

`forUpdate` →
≪ *Bezeichner* ≫ `=` `Ausdruck`
| `Ausdruck ++`
| `Ausdruck --`

- Die Initialisierungsanweisung forInit (deklariert und) initialisiert eine Zählvariable.
- Die Updateanweisung forUpdate ist typischerweise entweder
  - eine direkte Zuweisung (d.h. der Wert der Variablen wird auf den Wert eines Ausdrucks gesetzt und zurückgeliefert),
  - das Inkrementieren einer Variable durch den `++` Operator (d.h. der Wert der Variable wird um 1 erhöht und zurückgeliefert), oder
  - das Dekrementieren einer Variable durch den `--` Operator (d.h. der Wert der Variablen wird um 1 erniedrigt und zurückgeliefert)

Semantik:

- Es wird zunächst die Initialisierungsanweisung ausgeführt.
- (∗) Als nächstes wird der boolsche Ausdruck ausgewertet. Falls er zu `false` auswertet, ist die Ausführung der Schleife beendet. Falls er zu `true` auswertet, wird der Schleifenrumpf (Anweisung) ausgeführt. Danach wird die Updateanweisung ausgeführt.
- Die Auswertung wiederholt sich nun ab (∗).

Beispiele Arrays können dazu verwendet werden, um Vektoren zu implementieren. Das folgende Programm initialisiert zwei 3-elementige Vektoren mit Zufallszahlen, addiert sie und gibt das Ergebnis auf der Konsole aus.

```
1  public class VectorExample {
2    public static void main(String[] args){
3      int[] a = new int[3]; // 3-elementiger Vektor
4      int[] b = new int[3];
5      for (int i = 0; i < 3; i++){
6        a[i] = (int) (Math.random() * 10); //erzeugt Zufallszahl zwischen 0
           und 9
7        b[i] = (int) (Math.random() * 10);
8      }
9
10     int[] c = new int[3];
11     for (int i = 0; i < 3; i++){
12       c[i] = a[i] + b[i];
13     }
```

```
14      for (int i=0; i < 3; i++) {
15        System.out.print("a["+ i + "] = " +a[i]);
16        System.out.print(", b[" + i + "] = " +b[i]);
17        System.out.println(", c[" + i + "] = " +c[i]);
18      }
19    }
20  }
```

Hinweis: Die Ausgabe mit `System.out.println()` beendet die Ausgabe mit einem Zeilenumbruch, `System.out.print()` macht diesen Zeilenumbruch nicht.
Als weiteres Beispiel haben wir noch ein Code-Snippet, mit dem das Maximum der Elemente eines `double`-Arrays mit 10 Elementen ermittelt werden kann.

```
double[] a = new double[10];
... // Initialisierung von a mit Double-Werten

double max = a[0];
for (int i = 1; i < a.length; i++) {
  if(max < a[i]) {
    max = a[i];
  }
}
```

1.3 Fallbeispiel: Sieb des Eratosthenes

Um alle Primzahlen zu ermitteln, die kleiner oder gleich einem Eingabeparameter n sind, eignet sich der folgende Algorithmus, der als das "Sieb des Eratosthenes" bekannt ist.
Zunächst werden alle Zahlen 2, 3, 4, ... bis n aufgeschrieben. Die zunächst unmarkierten Zahlen sind alles potentielle Primzahlen. Die kleinste unmarkierte Zahl in diesem Verfahren ist immer eine Primzahl. Wenn eine Primzahl gefunden wird, werden alle Vielfachen dieser Primzahl als Nicht-Primzahlen markiert.
Für den Algorithmus bestimmt man jeweils die nächste nichtmarkierte Zahl. Da sie kein Vielfaches von Zahlen kleiner als sie selbst ist (sonst wäre sie markiert worden), kann sie nur durch eins und sich selbst teilbar sein. Folglich muss es sich um eine Primzahl handeln. Diese wird dementsprechend als Primzahl ausgegeben. Im nächsten Schritt streicht man alle Vielfachen dieser Zahl und führt das Verfahren fort, bis man am Ende der Liste angekommen ist.
Entwurf Als Eingabe erwarten wir die Zahl n; die Ausgabe soll die Primzahlen durch Leerzeichen getrennt ausgeben. Wir wollen in der Implementierung ein Array `isPrime` der Größe $n + 1$ mit boolschen Werten verwenden, das die Information verwaltet, welche der Zahlen zwischen 0 und n eine Primzahl ist. Für alle Positionen wird dieses Array zunächst mit `true` initialisiert. Ausgehend von der kleinsten Primzahl 2 wird nun dieses Array durchlaufen. Die nächste nichtmarkierte Primzahl, d.h. der nächste Index i, für den `isPrime[i]` den Wert `true` enthält, wird ausgegeben. Danach werden die Vielfachen von i als Nicht-Primzahl markiert, indem der Wert des Arrays an den entsprechenden Indizes auf `false` gesetzt wird.
Implementierung

```java
/* Berechnet die Primzahlen, die kleiner oder gleich einem
   Programmparameter n sind

   Programm-Parameter: Oberer Wert, bis zu dem
   Primzahlen ermittelt werden

   Testfaelle:

   10  -> 2 3 5 7
   30  -> 2 3 5 7 11 13 17 19 23 29
*/

public class Eratosthenes {
  public static void main(String[] args) {
    String eingabe = args[0];
    // Eingabeparameter n
    int n = Integer.parseInt(eingabe);
    boolean[] isPrime = new boolean[n+1];
    // Initialisierung des Arrays mit true
    for (int i = 0; i < isPrime.length; i++) {
        isPrime[i] = true;
    }
    // 2 ist die kleinste Primzahl
    for (int i = 2; i < isPrime.length; i++) {
      if (isPrime[i]) {
        System.out.print(i + " ");
        for (int j = 2; i * j < isPrime.length; j++) {
          // Alle Vielfachen von i koennen keine Primzahlen sein
          isPrime[i*j] = false;
        }
      }
    }
    System.out.println();
  }
}
```

Das Verfahren lässt sich folgendermaßen optimieren:

- Da mindestens ein Primfaktor einer Nicht-Primzahl immer kleiner gleich der Wurzel der Zahl sein muss, ist es ausreichend, nur die Vielfachen von Zahlen mit `false` zu markieren, die kleiner oder gleich der Wurzel von n sind.
- Es genügt beim Streichen der Vielfachen mit dem Quadrat der Primzahl zu beginnen, da alle kleineren Vielfachen bereits markiert worden sind.

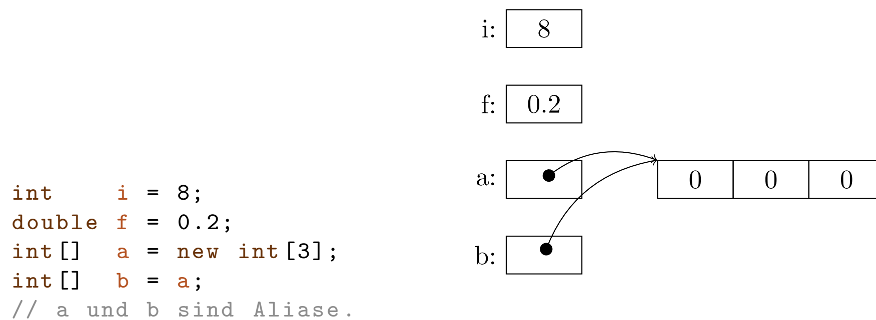
```java
/* Berechnet die Primzahlen kleiner oder gleich
   einer natuerlichen Zahl

   Programm-Parameter: Oberer Wert, bis zu dem
   Primzahlen ermittelt werden

   Testfaelle:
   10  -> 2 3 5 7
   30  -> 2 3 5 7 11 13 17 19 23 29
*/
```

```
11 public class ErathostenesOptimized {
12   public static void main(String[] args) {
13     int n = Integer.parseInt(args[0]);
14     boolean[] isPrime = new boolean[n+1];
15     for (int i = 0; i < n+1; i++) {
16         isPrime[i] = true;
17     }
18     // Mindestens ein Primfaktor einer Nicht-Primzahl muss
19     // immer kleiner gleich der Wurzel der Zahl sein
20     for (int i = 2; i*i < n+1; i++) {
21       if (isPrime[i]) {
22         // Es genuegt nur die Eintraege ab i zu betrachten,
23         // da die kleineren Vielfachen bereits markiert wurden!
24         for (int j = i; i*j < n+1; j++) {
25           isPrime[i*j] = false;
26         }
27       }
28     }
29     for (int i = 2; i < n+1; i++) { // Ausgabe
30         if (isPrime[i]) {
31             System.out.println(i + " ");
32         }
33     }
34   }
35 }
```

Diese Optimierung beschleunigt zwar die Berechnung, führt aber dazu, dass der Algorithmus weniger intuitiv und schwerer nachzuvollziehen ist.
Programmierer müssen immer gut abwägen, ob Optimierungen tatsächlich sinnvoll sind. Die Wiederverwendung von Variablen, um Deklarationen "zu sparen", ist beispielsweise keine sinnvolle Optimierung. Sie führt zu schwer verständlichem Programmtext ohne das Laufzeitverhalten tatsächlich zu verändern. Moderne Compiler können diese Art der Optimierung automatisch durchführen.

2 Referenzvariablen

Variablen speichern Werte, d.h. elementare Daten von primitiven Datentypen oder Referenzen auf Objekte (z.B. Arrays, Strings). Variablen, die Referenzen speichern, nennt man auch ***Referenzvariablen***. Sie enthalten als Werte Referenzen auf den Speicherbereich, der die eigentlichen Daten enthält. Diese Referenzen können Variablen zugewiesen werden, genauso wie Werte primitiver Datentypen. Dabei können auch mehrere Variablen den gleichen Speicherbereich / die gleichen Daten referenzieren. In diesem Fall spricht man von ***Aliasen***.



Ein spezieller Referenzwert ist `null`. Dieses Literal repräsentiert eine Referenz, die auf nichts verweist. Die Null-Referenz kann nicht dereferenziert werden, d.h. Lesen und Schreiben von Daten, die mit `null` referenziert werden, ist nicht möglich und führt in Java zu einem Laufzeitfehler (`NullpointerException`).
Achtung: Die Operationen auf Referenzvariablen, wie z.B. Vergleiche oder Zuweisungen, arbeiten mit den Referenzen, nicht mit den referenzierten Objekten.
Daher wird beim Vergleich von zwei Arrays oder zwei Strings mit `==` verglichen, ob beide Referenzen auf das selbe Objekt verweisen. Soll statt dessen der Inhalt geprüft werden, kann `==` nicht verwendet werden.
Für zwei Werte `x` und `y` vom Typ `String` kann dazu der Ausdruck `Objects.equals(x, y)` verwendet werden. Wenn `x` nicht `null` sein kann, wird auch oft der Ausdruck `x.equals(y)` verwendet.
Für zwei Array-Werte `x` und `y` können die Inhalte mit dem Ausdruck `Arrays.equals(x,y)` bzw. mit `Arrays.deepEquals(x,y)` verglichen werden. Bei der Verwendung von `deepEquals` werden auch die Inhalte von verschachtelten Arrays verglichen.
Um die Ausdrücke `Objects.equals`, `Arrays.equals` und `Arrays.deepEquals` verwenden zu können, muss zu Beginn der Java-Datei die Zeile `import java.util.Objects;` bzw. `import java.util.Arrays;` eingefügt werden.
Beispiel: Vergleichen und Kopieren von Arrays

> **Frage 1:** Was ist die Ausgabe des folgenden Programms?
> Führen Sie folgendes Beispiel im Java Visualizer[1] aus!

```
1 public class Arrayreferenz {
2   public static void main(String[] args) {
3     int n = 5;
4     int[] a = new int[n];
5     for (int i = 0; i < n; i++) {
6       a[i] = i;
7     }
8     int[] b = a; // a und b referenzieren das gleiche Objekt
9     System.out.println(a == b);
```

[1] Online unter `http://cscircles.cemc.uwaterloo.ca/java_visualize/`

```
10
11      int[] c = new int[n];
12      for (int i = 0; i < n; i++) {
13        c[i] = a[i]; // c referenziert eine Kopie von a
14      }
15      System.out.println(a == c);
16
17      b[2] = 100;
18      System.out.println(a[2] + " vs " + c[2]);
19    }
20  }
```

Beispiel: Vergleichen von Strings

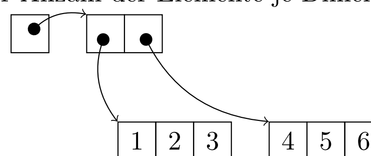
> **Frage 2:** Was ist die Ausgabe des folgenden Programms?
> Visualisieren Sie folgendes Beispiel im Java Visualizer und verwenden Sie dabei die Option “ Show String/Integer/etc objects, not just values”!

```
1  //Programm zur Erlaeuterung von String-Referenzen
2  public class Strings {
3    public static void main(String[] args) {
4      String a = "Hello, world!";
5      String b = "Hello, world!!".substring(0, 13);
6      String c = "Hello, ";
7      c = c + "world!";
8      String d = "Hello, w"+"orld!";
9      String e = a.substring(0, 13);
10     System.out.println((a == b) + " " + a.equals(b));
11     System.out.println((a == c) + " " + a.equals(c));
12     System.out.println((a == d) + " " + a.equals(d));
13     System.out.println((a == e) + " " + a.equals(e));
14   }
15 }
```

3 Mehrdimensionale Arrays

Mehrdimensionale Arrays sind “Arrays von Arrays”. Die Initialisierung erfolgt wie bei eindimensionalen Arrays durch Angabe der Anzahl der Elemente je Dimension.[2]

```
int[][] a = new int[2][3];
a[0][0] = 1;
a[0][1] = 2;
a[0][2] = 3;
a[1][0] = 4;
a[1][1] = 5;
a[1][2] = 6;
```



[2]Die formale Syntaxdefinition finden Sie in der Zusammenfassung am Ende.

Beispiel: Matrizen als mehrdimensionale Arrays
Um die Multiplikation zweier $n \times n$-Matrizen `a` und `b` zum implementieren, müssen verschachtelte for-Anweisungen verwendet werden.

```
double[][] a = ...; // Initialisierung
double[][] b = ...; // Initialisierung
double[][] c = new double[n][n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
  }
}
```

3.1 Ungleichförmige Arrays

In Java ist es nicht notwendig, dass alle Zeilen bzw. Subarrays die gleiche Größe haben. Der Umgang mit ungleichförmigen Arrays erfordert aber besondere Sorgfalt, um `ArrayIndexOutOfBounds`-Fehler zu vermeiden.
Im folgenden Beispiel wird ein mehrdimensionales Array konstruiert, dessen erster Eintrag auf ein Array der Größe 2, der zweite auf ein Array der Größe 1, und der dritte Eintrag auf ein Array der Größe 3 verweist. Bei der Iteration über die Einträge wird für jedes dieser Arrays die Größe ermittelt (`j < a[i].length;`).

```
int n = 3;
int[][] a = new int[n][];
a[0] = new int[2];
a[1] = new int[1];
a[2] = new int[3]
for (int i = 0; i < n; i++) {
  for (int j = 0; j < a[i].length; j++) {
    System.out.print(a[i][j]+ " ");
  }
  System.out.println();
}
```

3.2 Initialisieren von Arrays

Beim Erstellen eines Arrays können bereits initiale Werte für das Array angegeben werden.
Beispiele

```
int[] a = new int[] {1, 2, 3};
int[] b = {1, 2, 3};
int[][] c = {{1, 2, 3}, {4, 5}, {6}};
```

Die Anzahl der Elemente wird dann aus den angegebenen Werten abgeleitet.

Frage 3: Was ist das Ergebnis der folgenden Matrixmultiplikation?

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

Passen Sie den Code aus dem obigen Beispiel an, um die Berechnung zu implementieren!

4 Fallstudien: Simulationen

Simulationen werden in vielen Bereichen (Wirtschaft, Technik, Naturwissenschaften) eingesetzt, um Modelle zu erstellen und zu validieren. Sie ergänzen die (math.) Analyse und ersetzen sie bisweilen sogar in komplexen Situationen.

4.1 Beispiel: Sammelbilder

Seite 121 ff aus R. Sedgewick, K. Wayne: Einführung in die Programmierung mit Java. 2011, Pearson Studium.

Sammelkarten sind sehr beliebt bei Kindern, die gerne ihr Taschengeld dafür ausgeben ein Sammelheft mit Fussballern, Tierbabys oder ihren Lieblingsserienhelden zu füllen.

Wie viele Sammelkarten muss man im Mittel kaufen, um eine Serie von n Bildern zu erhalten (ohne Tauschen, alle Karten gleich wahrscheinlich)?

Um diese Frage zu beantworten, werden wir den Kauf im Programm simulieren. Die Karten werden dabei als Array von n boolschen Werten repräsentiert, initialisiert mit `false`, da zu Beginn noch keine Karten gekauft wurden. Es wird nun eine Zufallszahl zwischen 0 und $n-1$ erzeugt, um den zufälligen Kauf einer Karte zu modellieren. Der Wert an der entsprechenden Arrayposition wird dann auf `true` gesetzt (falls er noch nicht `true` ist). Der Simulationsdurchlauf endet, wenn alle Werte im Array `true` sind. Dabei wird gezählt, wie viele Versuche (= Käufe) dazu notwendig waren.

```
1  // Simulation von Sammelkartenkauf
2  // Programm-Parameter: Anzahl der Karten
3  // Ausgabe: Anzahl der benoetigten Kaeufe
4
5  public class Sammelkarten {
6    public static void main (String[] args) {
7      int n = Integer.parseInt(args[0]); // Anzahl der Karten
8      boolean[] karten = new boolean[n]; // Information, welche Karten
       erworben wurden
9      int anzahl = 0; // Anzahl der erworbenen unterschiedl. Karten
10     int versuche = 0; // Kaeufe
11
12     while (anzahl < n) {
```

```
13        // Simuliert zufaelligen Kauf
14        int naechste = (int) (Math.random() * n);
15        versuche++;
16
17        if (!karten[naechste]) {
18          anzahl++;
19          karten[naechste] = true;
20        }
21      }
22      System.out.println("Anzahl an Versuchen: " + versuche);
23    }
24  }
```

Eine anschauliche Erklärung zur mathematischen Analyse dieses Problems finden Sie im Spiegel Online Artikel "So funktioniert die Panini-Formel"[3] vom 23.04.2014.

4.2 Spielbankbesuche

Seite 87 ff aus R. Sedgewick, K. Wayne: Einführung in die Programmierung mit Java. 2011, Pearson Studium.

Wir betrachten hier zunächst eine Simulation von Gewinnwahrscheinlichkeiten beim Besuch von Spielbanken. Angenommen ein Spieler startet mit einem gegebenen Startkapital und will einen festgelegten Zielbetrag erreichen. Bei jeder Wette setzt er 1$ und kann entweder 1$ gewinnen oder verlieren. Wir nehmen an, dass das Kasino ein faires Spiel spielt (d.h. die Gewinnwahrscheinlichkeit liegt bei 50%). Das Spiel ist beendet, wenn der Spieler pleite ist oder wenn er den Zielbetrag erreicht hat.

Wie hoch sind die Chancen, dass der Spieler den Zielbetrag erreicht?
Wie viele Wetten sind dazu notwendig?

Folgendes Programm simuliert den Besuch in einer Spielbank. Es nimmt drei Programm-Parameter: den Einsatz/Startkapital, den Zielbetrag und die Anzahl an Versuchen. Es ermittelt die durchschnittliche Erfolgswahrscheinlichkeit und die Anzahl an Versuchen, die durchschnittlich nötig waren um zu gewinnen bzw. zu verlieren.

```
1   /* Programm zur Simulation von Wettspielen
2      Programm-Parameter:
3       einsatz  Geldeinsatz des Spielers
4       ziel         Betrag, der erreicht werden soll (Geldeinsatz + Gewinn)
5       versuche Anzahl der Simulationendurchlaeufe
6   */
7   public class Wettspiel {
8     public static void main (String[] args) {
9       int einsatz  = Integer.parseInt(args[0]);
10      int ziel     = Integer.parseInt(args[1]);
11      int versuche = Integer.parseInt(args[2]);
12
13      int wetten   = 0;
14      int gewinne  = 0;
```

[3]Artikel unter http://spon.de/aec54

```
15
16      for (int t = 0; t < versuche; t++) {
17        int cash = einsatz;
18        while( cash > 0 && cash < ziel) {
19          wetten++;
20          if (Math.random() < 0.5) {
21            cash++;
22          } else {
23            cash--;
24          }
25          if (cash == ziel) {
26            gewinne++;
27          }
28        }
29      }
30      System.out.println(100.0 * gewinne/versuche + "% Gewinne");
31      System.out.println("Durchschnittl. Anzahl Wetten: " + ((double)
        wetten)/versuche);
32    }
33  }
```

Bei der Simulation lassen sich folgende Beobachtungen machen:

- Die Erfolgswahrscheinlichkeit ist gegeben durch das Verhältnis von Einsatz zu Zielwert.
  Beispiel: Bei $500 Einsatz wird das Ziel $2500 in 20% aller Fälle erreicht.
- Die durchschnittliche Anzahl der Wetten ist gegeben durch das Produkt von Einsatz und Zielgewinn.
  Beispiel: Um aus $500 Einsatz $2500 Gewinn zu machen, braucht man im Durchschnitt etwa 1 Million Versuche.

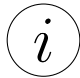
Frage 4: Die gegebene Implementierung terminiert nicht immer. Warum? Wie kann man die Implementierung abändern, so dass die Terminierung immer nach endlich vielen Schritten gewährleistet ist?

5 Eingabe und Ausgabe

Programme interagieren und kommunizieren mit der Ausführungsumgebung. Bisher haben wir Eingaben als Programmparameter auf der Kommandozeile beim Programmstart übergeben und Ausgaben auf die Konsole gemacht. Es gibt viele weitere Möglichkeiten Informationsschnittstellen zu gestalten:

- Grafik
- Audio
- Video
- Drucker, etc...

In der Vorlesung SE1 werden wir uns allerdings auf einfache Interaktionsmöglichkeiten über die Kommandozeile beschränken.

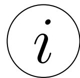


Die im Folgenden verwendeten Bibliotheken `StdOut` und `StdIn` finden Sie auf der Vorlesungsseite zum Download. Beim Übersetzen und Ausführen müssen diese Bibliotheken für Java auffindbar sein. Dazu können die Bibliotheks-Dateien in den gleichen Ordner gepackt werden.

5.1 Standardausgabe

Über die Standardausgabe kann ein Programm Zeichen ausgeben. In der Regel ist sie mit dem Konsolenfenster verbunden, so dass Ausgaben auf der Konsole ausgegeben werden. Bisher haben wir die Anweisung `System.out.println` verwendet, um Daten auf die Standardausgabe zu schreiben. Im Folgenden verwenden wir die Bibliothek `StdOut`, die verschiedene Prozeduren bereitstellt:

`void print(String s)`	Gibt den String `s` aus
`void println(String s)`	Gibt den String `s` gefolgt von einem Zeilenumbruch aus
`void println()`	Gibt einen Zeilenumbruch aus
`void printf(String f,...)`	Formatierte Ausgabe (⇒ siehe Übungen)



Die Signaturen in der Bibliothek `StdOut` weichen hiervon minimal ab, können aber genauso verwendet werden, wie hier beschrieben.

5.2 Standardeingabe

Analog zur Standardausgabe kann die Standardeingabe verwendet werden, um Zeichen in ein Programm einzulesen. Beim Lesen werden die Eingaben *verbraucht*, d.h. sie können nur einmal aus der Standardeingabe eingelesen werden. Die Standardeingabe kann leer sein, wenn alle eingegebenen Werte gelesen wurden und die Eingabe beendet wurde. Es kann auch sein, dass die Eingabe noch nicht beendet wurde, aber momentan kein Wert verfügbar ist. In diesem Fall wird gewartet, bis neue Werte eingegeben Werden oder die Eingabe beendet wird. Die entsprechenden Ausdrücke blockieren das Programm bis dies passiert.
Die Bibliothek `StdIn` bietet verschiedene Prozeduren, die es ermöglichen strukturiert Werte aus dem Eingabestream zu lesen, zum Beispiel Werte, die durch Leerzeichen, Tabulatoren, Zeilenumbruchzeichen, etc. von einander getrennt sind.
Hier ein Ausschnitt aus der Bibliothek `StdIn`:

`boolean`	`isEmpty()`	Testet, ob es weitere Werte gibt
`int`	`readInt()`	Liest einen Wert vom Typ `int`
`double`	`readDouble()`	Liest einen Wert vom Typ `double`
`boolean`	`readBoolean()`	Liest einen Wert vom Typ `boolean`
`String`	`readString()`	Liest einen Wert vom Typ `String`
`String`	`readLine()`	Liest den Rest der Zeile
`String`	`readAll()`	Liest den Rest des Textes

Bei fehlerhafter Eingabe wird ein `NumberFormatException`-Fehler geworfen (z.B. wenn versucht wird ein Integer einzulesen, aber kein Integer eingegeben wurde).

5.3 Interaktive Benutzereingaben

Alle Zeicheneingaben nach der Befehlszeile (Befehlszeilenparametern sind Teil der Befehlszeile) bilden den Eingabestream. Der Eingabestream wird *zeilenweise* verfügbar gemacht, d.h. erst nach Drücken der Eingabetaste (Enter) stehen die nächsten Werte des Streams zur Verfügung. Das **EOF**-Zeichen (End-of-File) gibt das Ende der Eingabe an. (Unter Linux kann dieses Zeichen erzeugt werden durch die Eingabe eines Zeilenumbruchs gefolgt von Strg-D[4], unter Windows durch Strg-Z gefolgt von einem Zeilenumbruch).

Beispiel: Verarbeitung von Eingaben beliebiger Länge Im folgenden Beispiel wird eine unbestimmte Anzahl von Double-Werten eingelesen und deren arithmetischer Mittelwert berechnet.

```
1  // Berechnet den Mittelwert aller Eingaben
2  public class Mittelwert {
3    public static void main(String[] args) {
4      double sum = 0.0;
5      int count = 0;
6      while (!StdIn.isEmpty()) {
7        double value = StdIn.readDouble();
8        sum = sum + value;
9        count++;
10     }
11     StdOut.println("Mittelwert: " + sum/count);
12   }
13 }
```

Das Programm kann folgendermaßen verwendet werden:

```
> java Mittelwert
10.0 5.0 6.0
3.0
7.0 32.0
Strg-D
Mittelwert: 10.5
```

5.4 Umleiten von Eingaben und Ausgaben

Oft sind die Daten zu umfangreich für manuelle Eingabe, oder es sollen Ergebnisse zur späteren Verwendung in Dateien gespeichert werden.
Man kann dazu die Standardausgabe in eine Datei umleiten (hier: in Datei `data.txt`):

```
java Mittelwert > data.txt
```

Ebenso lassen sich Daten aus einer Datei in die Standardeingabe umleiten (hier: aus Datei `data.txt`):

[4] Das heißt, man drückt die Steuerungstaste (Strg oder Crtl) und gleichzeitig die Taste mit dem Buchstaben D

```
java Mittelwert < data.txt
```

Soll die Ausgabe eines Programms direkt an ein anderes Programm weitergeleitet werden, so kann dazu der Pipe-Operator verwendet werden:

```
java Range 0 100 2 | java Mittelwert
```

Zusammenfassung: Neue Sprachelemente

```
Typ →
      ...
    | Typ []

Ausdruck →
      ...
    | new ArrayElementTyp  [ Ausdruck ]
    | new ArrayElementTyp  ArrayLaengen ArrayTypKlammern
    | new ArrayElementTyp ArrayTypKlammern ArrayKonstante
    | null
    | Ausdruck . length
    | Ausdruck [ Ausdruck ]

ArrayElementTyp → PrimitiverTyp | ≪ Bezeichner ≫

Zuweisung →
      ...
    | Ausdruck [ Ausdruck ] = Ausdruck;

Anweisung →
      ...
    | for (forInit; Ausdruck; forUpdate) Anweisung

forInit → Typ  ≪ Bezeichner ≫  = Ausdruck

forUpdate →
      Zuweisung
    | Ausdruck ++
    | Ausdruck --


ArrayLaengen →
      [ Ausdruck ] ArrayLaengen
    | [ Ausdruck ]

ArrayTypKlammern →
      []
    | ϵ


Deklaration →
      ...
    | Typ  ≪ Bezeichner ≫  =  ArrayKonstante;


ArrayKonstante →
```

```
    { ArrayEintragListe }
  | {}

ArrayEintragListe →
    Ausdruck , ArrayEintragListe
    ArrayKonstante , ArrayEintragListe
  | Ausdruck
  | ArrayKonstante
```

Hinweise zu den Fragen

Hinweise zu Frage 1: Die Ausgabe ist:

```
true
false
100 vs 2
```

Hinweise zu Frage 2: Die Ausgabe ist:

```
false true
false true
true true
true true
```

Hinweise zu Frage 3: Die Antwort enthält man durch Ausführung des folgenden Programms:

```
public class MatrixMultiplikation {
  public static void main(String[] args) {
    int n = 2;
    int k = 4;
    int m = 3;
    int[][] a = {{1, 2, 3, 4},{5, 6, 7, 8}}; // Initialisierung
    int[][] b = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}}; //
    Initialisierung
    int[][] c = new int[n][m];
    for (int i = 0; i < n; i++)
      for (int j = 0; j < m; j++)
        for (int h = 0; h < k; h++)
          c[i][j] = c[i][j] + a[i][h] * b[h][j];

    // Kompakte Ausgabe der Array-Inhalte
    System.out.println(java.util.Arrays.deepToString(a));
    System.out.println(java.util.Arrays.deepToString(b));
    System.out.println(java.util.Arrays.deepToString(c));
  }
}
```

Hinweise zu Frage 4: Das Programm verwendet zwei Schleifen, die möglicherweise nicht terminieren.

- Die `for`-Schleife terminiert für alle Werte von `versuche`. Falls `versuche` größer 0 ist, wird in jedem Schleifendurchlauf `t` um 1 erhöht, bis die Abbruchbedingung erfüllt ist. Andernfalls (`versuche <= 0`) wird die Schleife gar nicht ausgeführt.

- Die `while`-Schleife wird ausgeführt, solange `cash > 0 && cash < ziel` ist. In jedem Schleifendurchlauf wird `cash` mit 50%-Wahrscheinlichkeit um 1 erhöht oder um 1 verringert. Daher ist die Terminierung dieser Schleife nicht zwangsläufig gegeben! [Eine mathematische Analyse zeigt allerdings, dass die Wahrscheinlichkeit einer Nichtterminierung bei 0% liegt.]