Übersicht

4.4 Imperative Algorithmen

4.4.4 Globale Größen

- Variablen und Konstanten, so wie wir sie bisher kennengelernt haben, sind *lokale Größen*, d.h. sie sind nur innerhalb des Blocks (z.B. Schleife, Methode), der sie verwendet, bekannt.
- Es gibt auch *globale* Variablen und Konstanten, die in mehreren Algorithmen (Methoden und sogar Klassen) bekannt sind.
- Diese globalen Größen sind z.B. für den Datenaustausch zwischen verschiedenen Algorithmen geeignet.
- Globale Variablen heißen in Java *Klassenvariablen*. Diese Variablen gelten in der gesamten Klasse und ggf. auch darüberhinaus. Eine Klassenvariable definiert man üblicherweise am Beginn einer Klasse
- Die Definition wird von den Schlüsselwörtern **`public`** und **`static`** eingeleitet. Deren genaue Bedeutung lernen wir später kennen.
- Klassenvariablen kann man auch als Konstanten definieren.
  Wie bei lokalen Variablen dient hierzu das Schlüsselwort **`final`**

4.4 Imperative Algorithmen

4.4.4 Globale Größen


```
public class Kreis {

    public static final double PI = 3.14159    // Klassen-Konstante

    /**
     * Berechnung des Umfangs eines Kreises mit gegebenem Radius.
     * @param radius Der Radius des Kreises.
     * @return Der Umfang des Kreises.
     */
    public static double kreisUmfang(double radius) {
            return 2 * PI * radius;
    }

    /**
     * Berechnung der Flaeche eines Kreises mit gegebenem Radius.
     * @param radius Der Radius des Kreises.
     * @return Die Flaeche des Kreises.
     */
    public static double kreisFlaeche(double radius) {
            return PI * radius * radius;
    }
}
```

- Beispiel mit Klassen-Variable

```
class Programm{

   static void add(int x) {
      int sum = 0;
      sum += x;
   }

   public static void main(String[] args)
   {
      int sum=0;
      add(3);
      add(5);
      add(7);
      System.out.println("Summe: "+sum);
   }

}
```

```
Summe: 0
```

Warum?

```
class GlobExample{
   static int sum = 0;

   static void add(int x){
      sum += x;
   }

   public static void main(String[] args)
   {
      add(3);
      add(5);
      add(7);
      System.out.println("Summe: "+sum);
   }
}
```

```
Summe: 15
```

- Im Gegensatz zu lokalen Variablen muss man Klassenvariablen nicht explizit initialisieren. Sie werden dann automatisch mit ihren Standardwerten initialisiert:

Typname	Standardwert
`boolean`	`false`
`char`	`\u0000`
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0`
`double`	`0.0`

- Klassenkonstanten müssen dagegen explizit initialisiert werden.

- Lokale Variablen innerhalb einer Klasse können genauso heißen wie eine Klassenvariable.

```
public class Sichtbarkeit
{
    public static int variablenname;

    public static void main(String[] args)
    {
        boolean variablenname = true;
    }
}
```

- Das bedeutet: Während bei lokalen Variablen *Sichtbarkeit* und *Gültigkeit* zusammenfallen, muss man zwischen beiden Eigenschaften bei Klassenvariablen prinzipiell unterscheiden.

- Das ist kein Widerspruch zum Verbot, den gleichen Namen innerhalb des Gültigkeitsbereichs einer Variable nochmal zu verwenden, denn genau genommen heißt die Klassenvariable anders:
- Zu ihrem Namen gehört der vollständige Klassenname (inklusive des Package-Namens, was ein Package ist, lernen wir mal wieder erst später kennen).
  - Der vollständige Name der Konstanten `PI` aus der `Kreis`-Klasse ist also:
    ```
    Kreis.PI
    ```
  - Der vollständige Name der Variablen `sum` aus der `GlobExample`-Klasse ist also:
    ```
    GlobExample.sum
    ```
  - Der vollständige Name der Variablen `variablenname` aus der `Sichtbarkeit`-Klasse ist also:
    ```
    Sichtbarkeit.variablenname
    ```
- Unter dem vollständigen Namen ist eine globale Größe auch dann sichtbar, wenn der innerhalb der Klasse geltende Name durch den identisch gewählten Namen einer lokalen Variable verdeckt ist.

- Beispiel

```
public class Sichtbarkeit
{
    public static int variablenname;

    public static void main(String[] args)
    {
        boolean variablenname = true;
        System.out.println(variablenname); // Ausgabe: true
        System.out.println(Sichtbarkeit.variablenname); // Ausgabe?
    }
}
```

– Nochmal das Beispiel `Kreis` mit dem „echten“ Namen der Konstante `PI`

```
public class Kreis {

    public static final double PI = 3.14159

    public static double kreisUmfang(double radius) {
        return 2 * Kreis.PI * radius;
    }

    public static double kreisFlaeche(double radius) {
        return Kreis.PI * radius * radius;
    }
}
```

- *Achtung*: Globale Variablen möglichst vermeiden
  - Wenn globale Variable wirklich notwendig, dann nur einsetzen, wenn
    - sie in mehreren Methoden verwendet werden müssen
    - ihr Wert zwischen Methodenaufrufen erhalten bleiben muss
  - Wann immer möglich, lokale Variablen verwenden
    - Einfachere Namenswahl
    - Bessere Lesbarkeit:

      Deklaration und Verwendung der Variablen liegen nahe beieinander
    - Keine Nebeneffekte:

      Lokale Variablen können nicht durch andere Methoden versehentlich überschrieben werden

Übersicht

4.5 Reihungen und Zeichenketten

- Bisher haben wir einzelne Objekte der Grunddatentypen (Menge der bereitgestellten Sorten) verarbeitet
- Auf der anderen Seite wurde bei unseren Wechselgeldalgorithmen eine Menge von Objekten verarbeitet (die Ausgabe war eine Folge von EUR-Münzen)
- Unsere bisherigen Konzepte geben die Verarbeitung von Mengen von Objekten nicht her
- In einer Programmiersprache ist üblicherweise eine einfache Datenstruktur eingebaut, die es ermöglicht, eine Menge von Werten (typischerweise: gleichen Typs) zu modellieren.
- Im imperativen Paradigma ist das meist die *Reihung* (*Array*, *Feld*).
- Eine spezielle Form von Arrays sind Zeichenketten (*Strings*), bei denen die Objekte vom Typ *CHAR* bzw. **char** sind

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Eine Reihung (Feld, Array) ist ein Tupel von Komponentengliedern gleichen Typs, auf die über einen Index direkt zugegriffen werden kann.
- Formal handelt es sich um eine Folge, d.h. ein Array von Objekten eines bestimmten Typs T ist ein Element aus T*.
- Der Zugriff auf das *i*-te Element kann durch die Projektion formalisiert formalisiert werden.
- Eine Reihung mit *n* Komponenten vom Typ `<typ>` ist eine Abbildung von der Indexmenge I_n auf die Menge `<typ>`.
- **Achtung 1:**

  Bei den meisten Programmiersprachen beginnt die Indexmenge bei 0, d.h. für eine *n*-elementige Reihung gilt die Indexmenge $I_n = \{0, \ldots, n\text{-}1\}$
- **Achtung 2:**

  In Java sind Arrays *semidynamisch*, d.h., ihre Größe kann zur Laufzeit (=dynamisch) festgesetzt werden, danach aber nicht mehr geändert werden (=statisch). Dynamisches Wachstum ist nicht möglich!!!

- Beispiel: Eine **char**-Reihung `gruss` der Länge 13:

`gruss`:	'H'	'e'	'l'	'l'	'o'	','	' '	'W'	'o'	'r'	'l'	'd'	'!'
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12

ist formal eine Abbildung

$$\texttt{gruss} : \{0, 1, \ldots, 12\} \rightarrow \textbf{char}$$

$$i \mapsto \begin{cases} \text{'H'} & \text{falls} \quad i = 0 \\ \text{'e'} & \text{falls} \quad i = 1 \\ & \vdots \\ \text{'!'} & \text{falls} \quad i = 12 \end{cases}$$

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Der Typ eines Arrays, das Objekte vom Typ `<typ>` enthält, wird in Java als `<typ>[]` notiert.
- Beispiel: ein Array mit Objekten vom Typ **`int`** ist vom Typ **`int[]`**

- Variablen vom Typ `<typ>[]` können wie gewohnt vereinbart werden:

```
<typ>[] variablenName;
```

(Konstanten wie immer mit dem Zusatz **`final`**)

- Initialisierung (Erzeugung) eines Arrays kann auf verschieden Art und Weisen erfolgen. Die einfachste ist, alle Elemente der Reihe nach aufzuzählen:

```
<typ>[] variablenName = {wert1, wert2, …}
```

mit Werten vom Typ `<typ>`

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Zugriff (Projektion) auf das `i`-te Element eines Arrays `a` notiert man in Java als `a[i]`

  D.h. der Wert des Ausdrucks

  ```
  variablenName[i]
  ```

  ist der Wert des Arrays `variablenName` an der Stelle `i`.

- Ein Array hat immer eine feste *Länge*, die in einer Konstanten `length` festgehalten wird. Diese Konstante `length` wird mit einem Array automatisch miterzeugt. Der Name der Konstanten ist zusammengesetzt aus dem Namen des Arrays und dem Namen `length`

  Also die Länge des Arrays `a` ist der Wert des Ausdrucks `a.length`

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Beispiel

```
char a = 'a';
char b = 'b';
char c = 'c';
char[] abc = {a, b, c};
System.out.print(abc[0]);       // gibt den Character 'a'
                                // aus, den Wert des Array-
                                // Feldes mit Index 0.
                                // Allgemein: array[i] ist
                                // Zugriff auf das i-te
                                // Element
System.out.print(abc.length);   // gibt 3 aus
int[] zahlen = {1, 3, 5, 7, 9};
System.out.print(zahlen[3]);    // gibt die Zahl 7 aus
System.out.print(zahlen.length); // gibt 5 aus
```

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Oft legt man ein Array an, bevor man die einzelnen Elemente kennt. Die Länge muss man dabei angeben:

```
char[] abc = new char[3];
```

- Das Schlüsselwort new ist hier verlangt!!!
- Dann kann man das Array im weiteren Programmverlauf füllen:

```
abc[0] = 'a';
abc[1] = 'b';
abc[2] = 'c';
```

- Dass Arrays in Java *semidynamisch* sind, bedeutet: Es ist möglich, die Länge erst zur Laufzeit festzulegen.

```
// x ist eine Variable vom Typ int
// deren Wert bei der Ausfuehrung
// feststeht, aber nicht beim
// Festlegen des Programmcodes
char[] abc = new char[x];
```

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Wenn man ein Array anlegt:

```
int[] zahlen = new int[10];
```

  aber nicht füllt – ist es dann leer?

- Es gibt in Java keine leeren Arrays. Ein Array wird immer mit den Standardwerten des jeweiligen Typs initialisiert.
- Das spätere Belegen einzelner Array-Zellen ist also immer eine Änderung eines Wertes.

```
int[] zahlen = new int[10];
System.out.print(zahlen[3]);      // gibt 0 aus
zahlen[3] = 4;
System.out.print(zahlen[3]);      // gibt 4 aus
```

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Auch Array-Variablen kann man als Konstanten deklarieren. Dann kann man der Variablen keinen neuen Wert zuweisen:

```
final char[] ABC = {'a', 'b', 'c'};
final char[] DE = {'d', 'e'};
ABC = DE; // ungueltige Anweisung: Compilerfehler
```

Aber *Achtung*: einzelne Array-Komponenten sind normale Variablen. Man kann ihnen also einen neuen Wert zuweisen:

```
ABC[0] = 'd';
ABC[1] = 'e'; // erlaubt
System.out.print(ABC.length);     // gibt 3 aus
System.out.print(ABC[0]);         // gibt 'd' aus
System.out.print(ABC[1]);         // gibt 'e' aus
System.out.print(ABC[2]);         // gibt 'c' aus
```

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Wie passt das mit unserer Intuition der Zettel(-wirtschaft) zusammen?
- Zunächst ist eine Variable vom Typ `<typ>[]` ein Zettel auf dem wiederum weitere Zettel „liegen“
- Konstanten sind nicht radierbare Zettel, d.h. der Zettel auf dem die anderen Zettel liegen sind nicht radierbar, die Zettel, die darauf liegen aber schon

4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

- Beispiel: Suche im Array

  Aufgabe: Sei `a` ein Array `int[]` der Länge `n` und `q` $\in$ **`int`**
  - Annahme: es gibt keine Duplikate in `a`.
  - Suche `q` in `a` und gib, falls die Suche erfolgreich ist, die Position `i` mit `a[i] == q` aus. Falls die Suche erfolglos ist, gib den Wert `-1` aus.

- Einfachste Lösung: *Sequentielle Suche*
  - Durchlaufe `a` von Position `i` = 0, . . . , `n` – 1.
  - Falls `a[i] == q`, gib `i` aus und beende den Durchlauf.
  - Falls `q` nicht gefunden wird, gib `-1` aus.

- Methode in Java (Variante mit gezählter Schleife):

```
public static int sequentielleSuche(int q, int[] a)
{
  for(int i = 0; i < a.length; i++)
  {
    if(a[i]==q)
    {
      return i;
    }
  }
  return -1;
}
```

- Anzahl der Vergleiche (= Analyse der Laufzeit):
  - Erfolgreiche Suche: n Vergleiche maximal, n/2 im Durchschnitt.
  - Erfolglose Suche: n Vergleiche.

- Geht das besser?
- Ja, falls das Array sortiert ist, funktioniert auch die sog. *Binäre Suche*
  - Betrachte den Eintrag `a[i]` mit `i = n/2` in der Mitte des Arrays
    - Falls `a[i] == q`, gib `i` aus und beende die Suche.
    - Falls `a[i] > q`, suche in der linken Hälfte von `a` weiter.
    - Falls `a[i] < q`, suche in der rechten Hälfte von `a` weiter.
  - In der jeweiligen Hälfte kann ebenfalls mit binärer Suche gesucht werden (oh, das riecht nach Rekursion?!?!).
  - Falls die neue Hälfte des Arrays leer ist, gib `-1` aus.

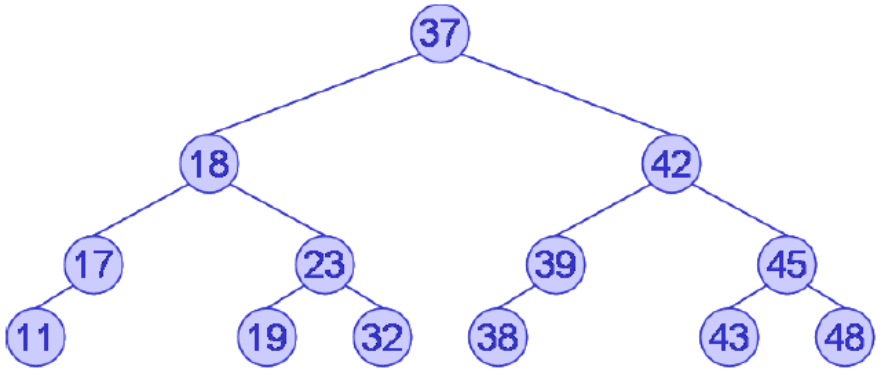
4.5 Reihungen und Zeichenketten

4.5.1 Reihungen (Arrays)

Beispiel: A:

11	17	18	19	23	32	37	38	39	42	43	45	48

- Entscheidungsbaum:
-



- Analyse der Laufzeit: Maximale Anzahl von Vergleichen entspricht Höhe h des Entscheidungsbaums: $h = \lceil \log_2(n+1) \rceil$
- Vergleich der Verfahren:
  - n = 1.000:
    Sequentielle Suche: 1.000 Vergleiche – Binäre Suche: 10 Vergleiche
  - n = 1.000.000:
    Sequentielle Suche: 1.000.000 Vergleiche – Binäre Suche: 20 Vergleiche

- Rekursiver Algorithmus in Java?
  - Berechne `mitte = a.length / 2;`
  - Wenn `a[mitte] == q` dann gib `mitte` aus (Basisfall)
  - Wenn `a[mitte] > q` dann suche im Teilarray mit der Indexmenge
    $I_n = \{$`0`, ..., `mitte-1`$\}$,

    d.h. rufe den Alg. rekursiv mit diesem Teilarray auf
  - Ansonsten suche im Teilarray mit der Indexmenge
    $I_n = \{$`mitte+1`, ..., `a.length`$\}$

    d.h. rufe den Alg. rekursiv mit diesem Teilarray auf
  - *ABER:* Wie wird das Teilarray erzeugt?

```
public int[] teilArray(int[] a, int first, int last) {
  int [] erg = int[last-first];
  for(int i = 0; i<erg.length; i++) {
          erg[i] = a[first+i];
  }
}
```

```
public int binSearchRek(int q, int[] a) {
 int mitte = a.length / 2;
 if (a[mitte] == q) {
        return mitte;
 } else if (a[mitte] > q) { // suche im vorderen Teil
        int[] aNeu = teilArray(a, 0, mitte-1);
        binSearchRek(q, aNeu);
 } else { // suche im hinteren Teil
        int[] aNeu = teilArray(a, mitte+1, a.length);
        binSearchRek(q, aNeu);
 }
}
```

- Alternativer iterativer Algorithmus in Java:

```
public static int binaereSucheIterativ (int q, int[] a)
{
  int first = 0;
  int last = a.length - 1;
  while (last >= first)
  {
    int m = (last + first) / 2;
    if (q == a[m])
    {
      return m;
    }
    else if (q < a[m])
    {
      last = m - 1;
    }
    else /*q > a[m]*/
    {
      first = m + 1;
    }
  }
  return -1;  // not found
}
```