Algorithmen implementieren

Implementieren von Algorithmen

- Um Algorithmen ablaufen zu lassen, muss man sie als Programm darstellen (d.h. implementieren)
- Wie stellt man die algorithmischen Strukturelemente in einem Programm dar?

Was ist Implementierung?

- Die Umsetzung eines Algorithmus in eine Programmiersprache, die von einer Maschine gelesen und interpretiert werden kann, bezeichnet man als **Implementierung**.
- Zur Implementierung der algorithmischen Strukturelemente (Sequenz, Alternative, Wiederholung) stellt jede Programmiersprache spezielle Sprachelemente zur Verfügung, die man auch als algorithmische **Kontrollstrukturen** bezeichnet.

Sequenzen (Anweisungsfolgen)

```
temp = hoehe; hoehe = breite; breite = temp;
System.out.println("Höhe = " + hoehe);
System.out.println("Breite = " + breite );
```

- In Java werden die einzelnen Anweisungen einer Sequenz durch **Strichpunkte** voneinander getrennt.
- Schließt man eine Sequenz in geschweifte Klammern ein, so entsteht ein Block, den man anstatt einer einzelnen Anweisung verwenden kann.

Sequenzen in Java:

```
<Anweisung1>;
<Anweisung2>;
<Anweisung3>;
…
<AnweisungN>;
```

Bedeutung: Die Folge von Anweisungen wird unbedingt vollständig ausgeführt

Wiederholung mit Anfangsbedingung

```
neueZahl = 1;
while (neueZahl != 6){
    neueZahl = wuerfeln();
    System.out.println(neueZahl);
}
```

Die Methode
`public int wuerfeln()`
muss anderweitig definiert sein

Wiederholung in Java:

```
while (<Bedingung>) {
    <Sequenz>
}
```

Bedeutung:

1) Falls <*Bedingung*> den Wert `true` hat, wird <*Sequenz*> ausgeführt. Danach wird <*Bedingung*> erneut ausgewertet und wieder bei 1) begonnen.
2) Falls <*Bedingung*> den Wert `false` hat, erfolgt keine Aktion. Die gesamte `while`-Anweisung wird abgeschlossen.

Alternative

```
if (b != 0) {
    System.out.print("Das Ergebnis lautet: ");
    System.out.println(a/b);
} else {
    System.out.println("Fehler!");
}
```

Sonderfall:

Bedingte Anweisung in Java:

```
if (<Bedingung>) {
    <Sequenz1>
}
```

Alternative in Java:

```
if (<Bedingung>) {
    <Sequenz1>
} else {
    <Sequenz2>
}
```

Bedeutung: Falls <Bedingung> den Wert `true` hat, wird <*Sequenz1*> ausgeführt, andernfalls <*Sequenz2*>

Lokale Variablen

```
public int summeBis (int letzterSummand) {
    int summe = 0; int i = 1;
    while (i <= letzterSummand) {
        summe = summe + i;
        i = i + 1;
    }
    return summe;
}
```

Lokale Variablen `summe, i`

- In objektorientierten Programmen benötigt man neben den Attributen der Objekte oft Datenspeicher für **Zwischenergebnisse**, die nach dem Abschluss der jeweiligen Berechnung nicht weiter benötigt werden.
- Man verwendet dafür spezielle Variablen innerhalb von Methoden. Diese Variablen sind nur während des Ablaufs dieser Methode, d. h. nur lokal innerhalb der Methode definiert und heißen deshalb **lokale Variablen**.
- Nach dem Ende der Ausführung der Methode wird der dafür reservierte Speicherplatz wieder freigegeben.

Globale Variablen bzw. Attribute

- Speicherstrukturen wie Attribute oder Variablen bezeichnet man als **global bezüglich einer Methode**, wenn sie vor dem Aufruf dieser Methode schon definiert sind und auch nach dem Ablauf der Methode noch definiert bleiben.
- Im Programmtext werden solche globalen Variablen außerhalb der Methode in übergeordneten Strukturen deklariert, z. B. in der Klasse, in der die Methode definiert wird.

Global oder lokal?

Bezüglich der Methode `uhr_verstellen` sind:

```
public class Uhr {
    int aktuelleStunde, aktuelleMinute;
    public void uhr_verstellen(int minuten) {
        int diffStunden, diffMinuten, uebertragStunden, uebertragMinuten;
        // Verstellung umrechnen in Stunden und Minuten
        diffStunden = minuten/60;
        diffMinuten = minuten-60*diffStunden;
        // Neue Minuten- bzw. Stundenwerte
        // mit Übertrag und Korrekturen
        aktuelleMinute = aktuelleMinute+diffMinuten;
        uebertragStunden = aktuelleMinute/60;
        aktuelleMinute = aktuelleMinute-uebertragStunden*60;
        aktuelleStunde = aktuelleStunde+diffStunden+uebertragStunden;
        uebertragStunden = aktuelleStunde/24;
        aktuelleStunde = aktuelleStunde-uebertragStunden*24;
    }
}
```

global

lokal

Seiteneffekte

- Einen schreibenden Zugriff auf globale Variable oder Attribute bezeichnet man auch als **Seiteneffekt** einer Methode, weil dadurch Werte zurückliefert, die (im Sinne einer Funktion) nicht explizit in ihrer Definition angegebenen sind.
- Diese Werte erscheinen ja weder im Kopf (d. h. der ersten Zeile) der Methodendefinition noch nach **return.**
- Dort ist also nicht erkennbar, welche Attributwerte die Methode verändert.
- Dies macht ein Programm unübersichtlich und fehleranfällig. Daher versucht man, Seiteneffekte möglichst zu vermeiden.

Seiteneffekte

```
public class Uhr {
    int aktuelleStunde, aktuelleMinute;

    public void uhr_verstellen(int minuten) {
        int diffStunden, diffMinuten, uebertragStunden, uebertragMinuten;
        // Verstellung umrechnen in Stunden und Minuten
        diffStunden = minuten/60;
        diffMinuten = minuten-60*diffStunden;
        // Neue Minuten- bzw. Stundenwerte
        // mit Übertrag und Korrekturen
        aktuelleMinute = aktuelleMinute+diffMinuten;
        uebertragStunden = aktuelleMinute/60;
        aktuelleMinute = aktuelleMinute-uebertragStunden*60;
        aktuelleStunde = aktuelleStunde+diffStunden+uebertragStunden;
        uebertragStunden = aktuelleStunde/24;
        aktuelleStunde = aktuelleStunde-uebertragStunden*24;
    }
}
```

globale Attribute

Seiteneffekt

Wiederholung mit fester Wiederholungsanzahl

```
public void Zinsen (double kapital, double zinssatz, int jahre) {
    double kontostand = kapital;
    for (int i=1; i<=jahre; i++){
        kontostand = kontostand*(1+zinssatz/100);
    }
    System.out.println("Kontostand nach "+jahre+" Jahren:");
    System.out.println(kontostand+" €");
}
```

Wiederholungen mit fester Anzahl von Durchläufen:

Lokale Zählvariable

Abbruchbedingung „wiederhole solange..“

Erhöhung des Wertes von i um 1 je Durchlauf

```
for (int i=1; i<=jahre; i++) { .. }
```

Äquivalent zu „Wiederhole für i=1 bis i = jahre mit Schrittweite 1“ oder

```
int i = 1;
while (i <=jahre) { .. i = i+1}
```

Algorithmen implementieren

Felder

- Mehrere Werte vom gleichen Typ können in Feldern abgespeichert werden.
- Felder können auch Objekte enthalten.

Wie kann man mehrere Werte für spätere Wiederverwendung speichern?

Lösung: Felder bzw. indizierte Variable

- Mathematische Schreibweise mit Index: $kontostand_i$ bezeichnet den Kontostand nach i Jahren
  z.B.: $kontostand_1$, $kontostand_2$,

- In Programmiersprachen als Feld, z.B. Java: `kontostand[i]`
  z.B. **`kontostand[1], kontostand[2], …`**

- Ein **Feld** besteht aus einer Menge gleichartiger (typgleicher) Variablen bzw. Attribute, hier z.B. vom Typ **`double`**:

  `kontostand[0], .. , kontostand[10]`

Deklaration und Erzeugen von Feldern

1. Felder müssen zunächst wie alle anderen Variablen oder Attribute **deklariert** werden:
   ```
   double[] kontostand;
   ```

2. Dann muss jedes Feld (ähnlich einem Objekt) **angelegt** werden:
   ```
   kontostand = new double[6];
   ```

   Bedeutung: Erzeuge die folgenden 6 indizierten Variablen/Attribute:
   ```
   kontostand[0], kontostand[1], .. kontostand[5]
   ```

Vorsicht:
`new double[n]` erzeugt Feldelemente mit den Indizes `0` bis `n-1`

Die Anlage wird oft auch gleich zusammen mit der Deklaration erledigt:
```
double[] kontostand = new double[6];
```

Belegen mit Werten

3. Schließlich müssen die einzelnen Elemente mit Werten **belegt** werden, z. B:

```
kontostand[0] = 10000;
kontostand[1] = 10000 * (1 + zinssatz/100);
…
```

Beispiel

```
public void zinsen2 (double kapital, double zinnsatz, int jahre) {
    double kontostand[]= new double[jahre+1];
    double aktuell = kapital;

    kontostand[0]= kapital;
    for (int i=1; i<=jahre; i=i+1){
        aktuell = aktuell * (1 + zinnsatz/100);
        kontostand[i] = aktuell;
    }
    // Ausgabe
     for (int i=0; i<=jahre; i=i+1){
         System.out.println("Kontostand nach " + i + " Jahren:");
         System.out.println(kontostand[i] + " €");
        }
}
```

Anwendung: Bubblesort

```
public void bubblesort() {
    String temp;
    for (int i=0;i<9;i++){
        for (int k=0; k<9; k++) {
            if(kundenname[k].compareTo(kundenname[k+1]) > 0){
                temp = kundenname[k];
                kundenname[k] = kundenname[k+1];
                kundenname[k+1] = temp;
            }
        }
    }
}
```

`compareTo(String s2)` ist eine Methode der Klasse `String`, die den Wert des aufgerufenen Objektes (z.B. `s1`) mit dem Wert von `s2` vergleicht. Das Ergebnis ist negativ, wenn `s1` alphabetisch vor `s2` steht, Null bei Gleichheit und positiv, wenn `s1` nach `s2` steht.

Objektwertige Felder

- Auch (zusammengesetzte) Objekte kann man mit Hilfe von Feldern verwalten. z. B. eine Reihe von Girokonten der Klasse `Konto` mit den folgenden Attributen

```
int nummer;
int kundennummer;
String art;
double kontostand;
```

Aufgabenstellung: Überzogene Konten suchen:

Wiederhole für alle Girokonten der SparBank
Überprüfe Kontostand
Falls Konto überzogen,
gib Warnung aus
EndeFalls
Ende Wiederhole

Objektwertige Felder

- Wir definieren dazu einfach ein Feld, das als Typ seiner Elemente die Klasse `Konto` hat:

```
Konto[] giro = new Konto[10];
```

- Dabei handelt es sich genau genommen um ein **Feld aus Referenzen**:

  Die einzelne Elemente des Feldes enthalten nicht Objekte der Klasse Konto, sondern lediglich **Referenzen** auf diese Objekte.

Explizites Anlegen der Feldelemente

- Bei der vorherigen Deklaration wird lediglich ein Feld für die **Referenzen auf die Objekte** erzeugt, **nicht die Objekte** selbst:

```
Konto[] giro = new Konto[10];
```

- Die einzelnen Objekte müssen jeweils noch mit `new` angelegt werden, z.B.:

```
giro[0] = new Konto(100122, 22343, "BestGiro", 1012.99)
```

Felder aus atomaren Typen vs. Objektwertige Felder

```
int[] fib = new int[10]
fib[0] = 0;
fib[1] = 1;
...
```

```
Konto[] giro = new Konto[10]
giro[0] = new Konto(100122, 22343, "BestGiro", 1012.99)
...
```