Übersicht

Abschnitt 5.
Grundlagen der funktionalen & imperativen Programmierung

Übersicht

Übersicht

Übersicht

Übersicht

5.6 Imperative Algorithmen

5.6.3 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`**

5.6 Imperative Algorithmen

5.6.3 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
```

5.6 Imperative Algorithmen

5.6.3 Globale Größen

- 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.

5.6 Imperative Algorithmen

5.6.3 Globale Größen

- 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.

Nachtrag zur Sichtbarkeit von Variablen

In Kapitel 5.5. wurde Sichtbarkeit definiert als

- Die Sichtbarkeit einer Variablen erstreckt sich auf alle Programmstellen, an denen man über den Namen der Variablen auf Ihren Wert zugreifen kann.

Soll heissen:

- Die Sichtbarkeit einer Variablen erstreckt sich auf alle Programmstellen, an denen man über den Namen der Variablen auf die Variable zugreifen kann.

Insbesondere kann eine deklarierte, aber noch nicht initialisierte Variable sichtbar sein, also auch wenn sie noch keine Wert zugewiesen bekommen hat.

5.6 Imperative Algorithmen

5.6.3 Globale Größen

- 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 (vorläufig) vollständige Name der Konstanten `PI` aus der `Kreis`-Klasse ist also:

    ```
    Kreis.PI
    ```
  - Der entsprechende Name der Variablen `sum` aus der `GlobExample`-Klasse ist also:

    ```
    GlobExample.sum
    ```
  - Der entsprechende 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?
    }
}
```

5.6 Imperative Algorithmen

5.6.3 Globale Größen

– 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;
    }
}
```

5.6 Imperative Algorithmen

5.6.3 Globale Größen

- *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

5.6 Imperative Algorithmen

5.6.4 Verzweigung und Iteration

- Um den Fluss von Anweisungen innerhalb eines Algorithmus steuern zu können, gibt es die bereits angesprochenen Konzepte der *Verzweigung* (bedingte Anweisung) und der *Iteration* (Wiederholungsanweisung)
- Wir lernen im folgenden die Umsetzungen dieser Konzepte in Java kennen
- Durch bedingte Anweisungen kann man nun auch Prozeduren rekursiv formulieren

- Java erlaubt zwei Formen von bedingten Anweisungen:

  Eine *einfache* Verzweigung:

```
if (<Bedingung>) <Anweisung>
```

  Eine *zweifache* Verzweigung:

```
if (<Bedingung>)
    <Anweisung1>
else
    <Anweisung2>
```

- Wobei
  - `<Bedingung>` ein Ausdruck vom Typ **boolean** ist,
  - `<Anweisung>`, `<Anweisung1>` und `<Anweisung2>` jeweils eine einzelne (ggfs. auch leere) Anweisung oder einen Block mit mehreren Anweisungen darstellen (dann mit {}).

- Java erlaubt zwei Formen von bedingten Anweisungen:

  Eine *einfache* Verzweigung:

```
if (<Bedingung>) <Anweisung>
```

  Eine *zweifache* Verzweigung:

```
if (<Bedingung>)
    <Anweisung1>
else
    <Anweisung2>
```

- Wobei
  - `<Bedingung>` ein Ausdruck vom Typ **boolean** ist,
  - `<Anweisung>`, `<Anweisung1>` und `<Anweisung2>` jeweils eine einzelne (ggfs. auch leere) Anweisung oder einen Block mit mehreren Anweisungen darstellen (dann mit {}).

- Beispiel: Der (rekursive) Algorithmus zur Berechnung der Fakultät einer natürlichen Zahl $n \in \mathbb{N}_0$ kann in Java durch folgende Methode umgesetzt werden:

```
public static int fakultaet01(int n)
{
  if(n==0)
  {
    return 1;
  }
  else
  {
    return n * fakultaet01(n-1);
  }
}
```

- Bedingte Anweisungen mit mehr als zwei Zweigen müssen in Java durch Schachtelung mehrerer **`if`**-Konstrukte ausgedrückt werden:

```
if (<Bedingung1>)
      <Anweisung1>
else if (<Bedingung2>)
   <Anweisung2>
.
.
.
else if (<BedingungN>)
      <AnweisungN>
else
      <AnweisungN+1>
```

- *Achtung*: Fehlerquellen durch nicht gesetzte Blockklammern
  - Dangling Else

```
if (a)
  if (b)
      s1;
else
  s2;
```

Frage: Zu welchem if-Statement gehört der else-Zweig?

Antwort: Zur inneren Verzweigung `if (b)`. (Die (*falsche*!) Einrückung ist belanglos für den Compiler und verführt den menschlichen Leser hier, das Programm falsch zu interpretieren.)

Mit Blockklammern wäre das nicht passiert:

```
if (a) {
  if (b) { s1; }
  else { s2; }
}
```

oder

```
if (a) {
  if (b) { s1; }
}
else { s2; }
```

- *Achtung*: Fehlerquellen durch nicht gesetzte Blockklammern (Teil 2)
  - Beispiel: Ein Kunde hebt einen Betrag (`betrag`) von seinem Konto ab (die Variable `kontoStand` speichert den aktuellen Kontostand). Falls der Betrag nicht gedeckt ist, wird eine Überziehungsgebühr fällig (Konstante `UEBERZIEH_GEBUEHR`). Die fälligen Gebühren werden über einen bestimmten Zeitraum akkumuliert (`gebuehren`) und am Ende des Zeitraums in Rechnung gestellt. Was ist falsch in folgender Berechnung?

```
if (kontoStand >= betrag)
{
  double neuerStand = kontoStand - betrag;
  kontoStand = neuerStand;
}
else
kontoStand = kontoStand - betrag;
gebuehren = gebuehren + UEBERZIEH_GEBUEHR;
```

5.6 Imperative Algorithmen

5.6.4 Verzweigung und Iteration

- Problem im vorherigen Beispiel:
  Überziehungsgebühr wird immer verlangt, auch wenn das Konto gedeckt ist.
- Lösung: Blockklammern setzen.

```
if (kontoStand >= betrag)
{
  double neuerStand = kontoStand - betrag;
  kontoStand = neuerStand;
}
else
{
  kontoStand = kontoStand - betrag;
  gebuehren = gebuehren + UEBERZIEH_GEBUEHR;
}
```

- Nun wird die Überziehungsgebühr nur verlangt, wenn das Konto *nicht* gedeckt ist.

- Spezielle Mehrfachverzweigungen als *Sprunganweisungen*:
  - In Java gibt es eine weitere Möglichkeit, spezielle Mehrfachverzweigungen auszudrücken.
  - Die sog. **`switch`**-Anweisung funktioniert allerdings etwas anders als bedingte Anweisungen.
  - Syntax:

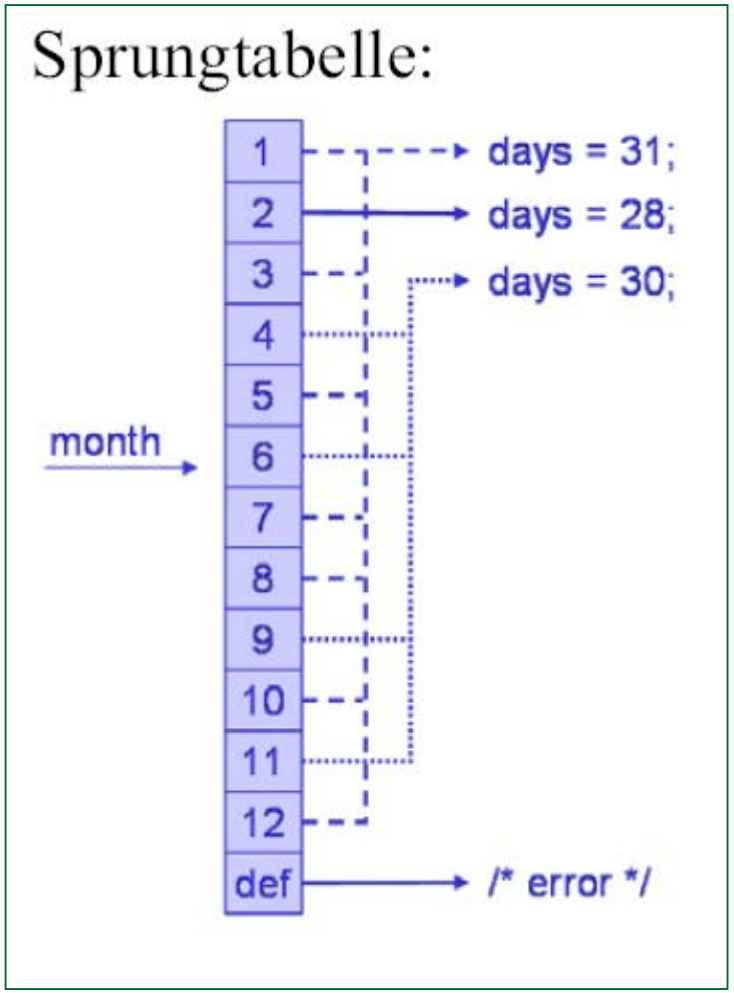
```
switch (<Ausdruck>)
{
  case <Konstante1> : <Anweisungsfolge1>
  case <Konstante2> : <Anweisungsfolge2>
    .
    .
    .
  default : <AnweisungsfolgeN>
}
```

- Bedeutung:
  - Abhängig vom Wert des Ausdrucks `<Ausdruck>` wird die *Sprungmarke* angesprungen, deren Konstante `<Konstante_i>` mit dem Wert von `<Ausdruck>` übereinstimmt.
  - Die Konstanten `<Konstante_i>` und der Ausdruck `<Ausdruck>` müssen den selben Typ haben.
  - Die Anweisungen nach der Sprungmarke werden ausgeführt.
  - Die optionale **`default`**-Marke wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird.
  - Fehlt die **`default`**-Marke und wird keine passende Sprungmarke gefunden, so wird keine Anweisung innerhalb der **`switch`**-Anweisung ausgeführt.

- Besonderheiten:
  - Der Ausdruck `<Ausdruck>` darf nur vom Typ **`byte`**, **`short`**, **`int`** oder **`char`** sein.
  - Die **`case`**-Marken sollten alle verschieden sein, müssen aber nicht.
  - ***Achtung***: Wird zu einer Marke gesprungen, werden alle Anweisungen hinter dieser Marke ausgeführt. Es erfolgt *keine* Unterbrechung, wenn das nächste Label erreicht wird, sondern es wird dort fortgesetzt! Dies ist eine beliebte Fehlerquelle!
  - Eine Unterbrechung kann durch die Anweisung **`break;`** erzwungen werden. Jedes **`break`** innerhalb einer **`switch`**-Anweisung verzweigt zum Ende der **`switch`**-Anweisung.
  - Nach einer Marken-Definition **`case`** muss nicht zwingend eine Anweisung stehen.

- Beispiel

```
switch (month)
{
  case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
      days = 31; break;
  case 4: case 6: case 9: case 11:
      days = 30; break;
  case 2:
      if(leapYear)
      {
        days = 29;
      }
      else
      {
        days = 28;
      }
      break;
    default: /* error */
}
```



- Bei der Iteration gibt es zunächst die *bedingte Wiederholung* (*bedingte Schleifen*):
  - Java kennt mehrere Arten von bedingten Schleifen.
  - Schleifen mit dem Schlüsselwort `while`:
  - Die klassische While-Schleife:
    `while` `(<Bedingung>) <Anweisung>`
  - Die Do-While-Schleife:
    `do` `<Anweisung>` **`while`** `(<Bedingung>);`
  - Dabei bezeichnet `<Bedingung>` einen Ausdruck vom Typ **`boolean`** und `<Anweisung>` ist eine einzelne (ggfs. auch leere) Anweisung oder ein Block mit mehreren Anweisungen.
  - Unterschied: `<Anweisung>` wird einmal *vor* (Do-While) bzw. *nach* (While) der Überprüfung von `<Bedingung>` ausgeführt.
  - Ist `<Bedingung>` **`false`**, wird die Schleife verlassen.

- Beispiele: nicht-rekursive Algorithmen für die Fakultät mit While:

```
public static int fakultaet02(int n)
{
  int erg = 1;
  while (n > 0)
  {
    erg = erg * n;
    n--;
  }
  return erg;
}
```

  - Variante

```
public static int fakultaet03(int n)
{
  int erg = 1;
  int laufVariable = 1;
  while (laufVariable <= n)
  {
    erg = erg * laufVariable;
    laufVariable++;
  }
  return erg;
}
```

– Variante mit Do-While

```
public static int fakultaet04(int n)
{
  int erg = 1;
  int laufVariable = 1;
  do
  {
    erg = erg * laufVariable;
    laufVariable++;
  }
  while (laufVariable < n);
  return erg;
}
```

- Eine weitere bedingte Schleife kann in Java mit dem Schlüsselwort for definiert werden:

```
for (<Initialisierung>; <Bedingung>; <Update>)
  <Anweisung>
```

- Alle drei Bestandteile im Schleifenkopf sind Ausdrücke (nur `<Bedingung>` muss vom Typ **`boolean`** sein).
- *Vorsicht*: Dieses Konstrukt ist keine klassische gezählte Schleife (auch wenn es for-Schleife genannt wird).

5.6 Imperative Algorithmen

5.6.4 Verzweigung und Iteration

- Der Ausdruck `<Initialisierung>`
  - wird einmal vor dem Start der Schleife aufgerufen
  - darf Variablendeklarationen mit Initialisierung enthalten (um einen Zähler zu erzeugen); diese Variable ist nur im Schleifenkopf und innerhalb der Schleife (`<Anweisung>`) sichtbar aber nicht außerhalb
  - darf auch fehlen
- Der Ausdruck `<Bedingung>`
  - ist ähnlich wie bei den While-Konstrukten die Testbedingung
  - wird zu Beginn jedes Schleifendurchlaufs überprüft
  - die Anweisung `<Anweisung>` wird nur ausgeführt, wenn der Ausdruck `<Bedingung>` den Wert **`true`** hat
  - kann fehlen (dies ist dann gleichbedeutend mit dem Ausdruck **`true`**)
- Der Ausdruck `<Update>`
  - verändert üblicherweise den Schleifenzähler (falls vorhanden)
  - wird am Ende jedes Schleifendurchlaufs ausgewertet
  - kann fehlen

- Eine gezählte Schleife wird in Java wie folgt mit Hilfe der for-Schleife notiert:

```
for (<Zaehler>=<Startwert>;
     <Zaehler> <= <Endwert>;
     <Zaehler> = <Zaehler> + <Schrittweite>)
<Anweisung>
```

- Beispiel: Fakultät

```
public static int fakultaet05(int n)
{
  int erg = 1;
  for(int i=1; i<=n; i++)
  {
    erg = erg * i;
  }
  return erg;
}
```

5.6 Imperative Algorithmen

5.6.4 Verzweigung und Iteration

- In Java gibt es Möglichkeiten, die normale Auswertungsreihenfolge innerhalb einer **`do`**-, **`while`**- oder **`for`**-Schleife zu verändern.
- Der Befehl **`break`** beendet die Schleife sofort. Das Programm wird mit der ersten Anweisung nach der Schleife fortgesetzt.
- Der Befehl **`continue`** beendet die aktuelle Iteration und beginnt mit der nächsten Iteration, d.h. es wird an den Beginn des Schleifenrumpfes "gesprungen".
- Sind mehrere Schleifen ineinander geschachtelt, so gilt der **`break`** bzw. **`continue`**-Befehl nur für die aktuelle (innerste) Schleife.

5.6 Imperative Algorithmen

5.6.4 Verzweigung und Iteration

- Mit einem *Sprungbefehl* kann man an eine beliebige Stelle in einem Programm "springen".
- Die Befehle **`break`** und **`continue`** können in Java auch für (eingeschränkte) Sprungbefehle benutzt werden.
- Der Befehl **`break`** `<label>;` bzw. **`continue`** `<label>;` muss in einem Block stehen, vor dem die Marke `<label>` vereinbart ist. Er bewirkt einen Sprung an das Ende dieses Anweisungsblocks.
- Beispiel:

```
int n = 0;
loop1:
for (int i=1; i<=10, i++)
{
  for (int j=1, j<=10, j++)
  {
    n = n + i * j;
    break loop1;
  }
}
System.out.print(n); // n = 1
```

Der Befehl **`break`** `loop1;` erzwingt den Sprung an das Ende der äußeren **`for`**-Schleife.

Anmerkung:
Durch die Sprungbefehle (vor allem bei Verwendung von Labeln) wird ein Programm leicht unübersichtlich und eine Korrektheitsüberprüfung wird schwierig. Wenn möglich, sollten Sprungbefehle daher vermieden werden.

- Sog. *unerreichbare Befehle* sind Anweisungen, die (u.a.)
  - hinter einer **`break`**- oder **`continue`**-Anweisung liegen, die ohne Bedingung angesprungen werden,
  - in Schleifen stehen, deren Testausdruck zur Compile-Zeit **`false`** ist.
- Solche unerreichbaren Anweisungen sind in Java nicht erlaubt, sie werden vom Compiler nicht akzeptiert.
- Einzige Ausnahme sind Anweisungen, die hinter der Klausel

```
if (false)
```

stehen. Diese Anweisungen werden von den meisten Compilern nicht in den Bytecode übernommen, sondern einfach entfernt. Man spricht von *bedingtem Kompilieren*.