Überblick

4. Imperative Programmierung
4.1 Grunddatentypen und Ausdrücke
4.2 Imperative Variablenbehandlung
4.3 Anweisungen, Blöcke und Gültigkeitsbereiche
4.4 Klassenvariablen
4.5 Reihungen
4.6 (Statische) Methoden
4.7 Kontrollstrukturen
4.8 ... putting the pieces together ...

Anweisungen

- In imperativen Programmen sind *Anweisungen* die elementaren Einheiten.
- Eine Anweisung steht für einen einzelnen Abarbeitungsschritt in einem Algorithmus.
- Anweisungen können unter anderem aus Ausdrücken gebildet werden.

Anweisungen in Java

- Im folgenden: Anweisungen in Java.
- Eine Anweisung wird immer durch das Semikolon begrenzt.

Die leere Anweisung

- Die einfachste Anweisung ist die leere Anweisung:

```
;
```

- Die leere Anweisung hat keinen Effekt auf das laufende Programm, sie bewirkt nichts.
- Oftmals benötigt man tatsächlich eine leere Anweisung, wenn von der Logik des Algorithmus *nichts* zu tun ist, die Programmsyntax aber eine Anweisung erfordert.

Der Block

- Ein Block wird gebildet von einer öffnenden geschweiften Klammer und einer schließenden geschweiften Klammer, die eine beliebige Menge von Anweisungen umschließen:

```
{
  Anweisung1;
  Anweisung2;
  ...
}
```

- Die Anweisungen im Block werden nacheinander ausgeführt.
- Der Block als ganzes gilt als eine einzige Anweisung, kann also überall da stehen, wo syntaktisch eine einzige Anweisung verlangt ist.
- Eine Anweisung in einem Block kann natürlich auch wieder ein Block sein.

Lebensdauer, Gültigkeit, Sichtbarkeit

- Die *Lebensdauer* einer Variablen ist die Zeitspanne, in der die virtuelle Maschine der Variablen einen *Speicherplatz* zu Verfügung stellt.
- Die *Gültigkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen der Name der Variablen dem Compiler durch eine Vereinbarung (*Deklaration*) bekannt ist.
- Die *Sichtbarkeit* einer Variablen erstreckt sich auf alle Programmstellen, an denen man über den Namen der Variablen auf ihren Wert zugreifen kann.

Gültigkeitsbereich von Variablen

- Eine in einem Block deklarierte (lokale) Variable ist ab ihrer Deklaration bis zum Ende des Blocks gültig und sichtbar.
- Mit Verlassen des Blocks, in dem eine lokale Variable deklariert wurde, endet auch ihre Gültigkeit und Sichtbarkeit.
- Damit oder kurz danach endet normalerweise auch die Lebensdauer der Variablen, da der Speicherplatz, auf den die Variable verwiesen hat, im Prinzip wieder freigegeben ist und für neue Variablen verwendet werden kann.
- Solange eine lokale Variable sichtbar ist, darf keine neue lokale Variable gleichen Namens angelegt werden.

Gültigkeitsbereich von Variablen

- Beispiel:

```
...
int i = 0;
{
  int i = 1;    // nicht erlaubt
  i = 1;        // erlaubt
  int j = 0;
}
j = 1;          // nicht moeglich
...
```

Ausdrucksanweisung

Aus einem Ausdruck `<ausdruck>` wird durch ein angefügtes Semikolon eine Anweisung. Allerdings spielt dabei der Wert des Ausdrucks im weiteren keine Rolle. Daher ist eine solche Ausdrucksanweisung auch nur sinnvoll (und in Java nur dann erlaubt), wenn der Ausdruck einen Nebeneffekt hat. Solche Ausdrücke sind:

- Wertzuweisung
- (Prä-/Post-)Inkrement
- (Prä-/Post-)Dekrement
- Methodenaufruf (werden wir später kennenlernen)
- Instanzerzeugung (werden wir später kennenlernen)

Überblick

4. Imperative Programmierung
4.1 Grunddatentypen und Ausdrücke
4.2 Imperative Variablenbehandlung
4.3 Anweisungen, Blöcke und Gültigkeitsbereiche
4.4 Klassenvariablen
4.5 Reihungen
4.6 (Statische) Methoden
4.7 Kontrollstrukturen
4.8 ... putting the pieces together ...

Global – lokal

- Mit den bisherigen Konzepten können wir theoretisch einfache imperative Algorithmen und Programme schreiben.
- Wir werden später weitere Konzepte kennenlernen, um komplexere Algorithmen/Programme zu strukturieren:
  - Prozeduren (in Java statische Methoden genannt).
    Beispiel: die Prozedur (Methode) `main`. Ein Algorithmus ist in einer Prozedur (Methode) verpackt.
  - Module (in Java Pakete bzw. Packages genannt)
- Variablen, so wie wir sie bisher kennengelernt haben, sind *lokale* Variablen und Konstanten, d.h. sie sind nur innerhalb des Blocks (Algorithmus, Prozedur, Methode), der sie verwendet, bekannt.
- Darüberhinaus gibt es auch noch *globale* Variablen und Konstanten, die in mehreren Algorithmen (Prozeduren/Methoden) und sogar Modulen bekannt sind.
- Diese globalen Größen sind z.B. für den Datenaustausch zwischen verschiedenen Algorithmen geeignet.

Globale und lokale Größen in Java

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, z.B.:

```
public class HelloWorld
{
  public static String gruss = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(gruss);
  }
}
```

Die Definition wird von den Schlüsselwörtern **`public`** und **`static`** eingeleitet. Deren Bedeutung lernen wir später kennen.

Konstante Klassenvariablen

Klassenvariablen kann man auch als Konstanten definieren. Wie bei lokalen Variablen dient hierzu das Schlüsselwort **`final`**:

```
public class HelloWorld
{
  public static final String GRUSS = "Hello, World!";

  public static void main(String[] args)
  {
    System.out.println(GRUSS);
  }
}
```

Auch bei Klassenvariablen schreibt man Namen von Konstanten in Großbuchstaben.

Klasse und Programm

- Zur Erinnerung: Ein Java-Programm besteht aus Klassen – einer oder mehreren, in größeren Projekten oft hunderten oder tausenden.
- Da eine Klasse auch einen Block bildet, ist der Gültigkeitsbereich einer Klassenvariablen klar: Sie gilt im gesamten Programmcode innerhalb der Klasse nach ihrer Deklaration.
- Darüberhinaus gilt sie aber in der gesamten Programmlaufzeit, d.h. solange das Programm ausgeführt wird, “lebt” eine Klassenvariable.
- Die Sichtbarkeit in anderen als ihrer eigenen Klasse kann man aber einschränken.
- Eine Klasse gehört immer zu einem Package. Die Klassendatei liegt in einem Verzeichnis, das genauso heißt wie das Package.
- Der Package-Name gehört zum Klassennamen.

Nutzen von Klassenvariablen

Als Klassenvariablen definiert man z.B. gerne Werte, die von universellem Nutzen sind. Die Klasse `java.lang.Math` definiert die mathematischen Konstanten e und π:

```
package java.lang;

public final class Math {
  ...
  /**
    * The <code>double</code> value that is closer than any other to
    * <i>e</i>, the base of the natural logarithms.
    */
  public static final double E = 2.7182818284590452354;

  /**
    * The <code>double</code> value that is closer than any other to
    * <i>pi</i>, the ratio of the circumference of a circle to its
    * diameter.
    */
  public static final double PI = 3.14159265358979323846;
  ...
}
```


Automatische Initialisierung

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.

Namen von lokalen Variablen und Klassenvariablen

- 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.

Namen von lokalen Variablen und Klassenvariablen

- Das ist aber kein Widerspruch zum Verbot, den gleichen Namen innerhalb des Gültigkeitsbereichs einer Variable nochmal zu verwenden, denn genaugenommen heißt die Klassenvariable doch anders:
- Zu ihrem Namen gehört der vollständige Klassenname (inklusive des Package-Namens).
  Der vollständige Name der Konstanten `PI` aus der `Math`-Klasse ist also: `java.lang.Math.PI`
- Unter dem vollständigen Namen ist eine globale Variable auch dann sichtbar, wenn der innerhalb der Klasse geltende Name (z.B. `PI`) durch den identisch gewählten Namen einer lokalen Variable verdeckt ist.

Namen von lokalen Variablen und Klassenvariablen

```
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?
    }
}
```


Überblick

4. Imperative Programmierung
4.1 Grunddatentypen und Ausdrücke
4.2 Imperative Variablenbehandlung
4.3 Anweisungen, Blöcke und Gültigkeitsbereiche
4.4 Klassenvariablen
4.5 Reihungen
4.6 (Statische) Methoden
4.7 Kontrollstrukturen
4.8 ... putting the pieces together ...

Eingebaute Datenstruktur

- In einer Programmiersprache ist üblicherweise eine einfache Datenstruktur eingebaut, die es ermöglicht, eine Reihe von Werten (gleichen Typs) zu modellieren.
- Im imperativen Paradigma ist das normalerweise das Array (Reihung, Feld).

  Vorteil: direkter Zugriff auf jedes Element möglich
  Nachteil: dynamisches Wachstum ist nicht möglich
  (In Java sind Arrays *semidynamisch*, d.h., ihre Größe kann zur Laufzeit (=dynamisch) festgesetzt werden, danach aber nicht mehr geändert werden.)

Reihung

Definition

Eine Reihung (Feld, Array) ist ein Tupel von Komponentengliedern gleichen Typs, auf die über einen Index direkt zugegriffen werden kann.

Darstellung

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

Mathematische Darstellung

Allgemein

Eine Reihung mit n Komponenten vom Typ `<typ>` ist eine Abbildung von der Indexmenge I_n auf die Menge `<typ>`.

Beispiel

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

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

Deklaration von Arraytypen und Arrayvariablen in Java

- Der Typ eines Arrays, das den Typ
  `<typ>`
  enthält, ist in Java:
  `<typ>[]`.
- Der Wert des Ausdrucks
  `<typ>[i]`
  ist der Wert des Arrays an der Stelle `i`.

Deklaration von Arraytypen und Arrayvariablen in Java

Ein Array wird zunächst behandelt wie Variablen von primitiven Typen auch. Wir können also z.B. deklarieren:

```
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
char d = 'd';
char e = 'e';
char[] de = {d, e};
abc = de;  // die Variable abc hat jetzt den gleichen
           // Wert wie die Variable de
System.out.print(abc[0]); // Ausgabe ?
```


Länge einer Reihung

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

```
char a = 'a';
char b = 'b';
char c = 'c';
char[] abc = {a, b, c};
System.out.print(abc.length); // gibt 3 aus
char[] de = {d, e};
abc = de; // gleicher Variablenname,
          // haelt aber als Wert ein
          // anderes Array
System.out.print(abc.length); // Ausgabe ?
```

Reihungen dynamisch anlegen

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

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

- 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];
```

- Dann kann man das Array im weiteren Programmverlauf füllen:

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


Leere Reihungen?

- 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 befüllt.
- 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
```

Array-Variablen als Konstanten

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 einzelne Array-Komponenten sind normale Variablen. Man kann ihnen also einen neuen Wert zuweisen. Die Länge des Arrays ändert sich dadurch nicht:

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


Array als Funktion

Rufen wir uns die abstrakte Betrachtungsweise eines Arrays als Funktion in Erinnerung:

```
final char[] ABC = {'a', 'b', 'c'};
```

$$\texttt{ABC} \;:\; \{0,1,2\} \rightarrow \textbf{char}$$

$$i \mapsto \begin{cases} \text{'a'} & \text{falls} \quad i = 0 \\ \text{'b'} & \text{falls} \quad i = 1 \\ \text{'c'} & \text{falls} \quad i = 2 \end{cases}$$

Was bedeutet die Neubelegung einzelner Array-Zellen?

```
ABC[0] = 'd';
ABC[1] = 'e';
```

Der Typ `String` als spezielles Array

- In der Deklaration und Initialisierung

```
public static String gruss = "Hello, World!";
```

  stellt der Ausdruck `"Hello, World!"` eine spezielle Schreibweise für ein konstantes Array `char[13]` dar, das in einen Typ `String` gekapselt wird.
- Durch die besonderen Eigenschaften des Typs `String` können die Komponenten dieses Arrays nicht mehr (durch Neuzuweisung) geändert werden.
- Der Typ `String` ist kein *primitiver* Typ, sondern eine Klasse von Objekten. Wir untersuchen diesen Typ daher erst in einem späteren Abschnitt der Vorlesung genauer.
- Werte dieses Typs können aber – wie bei primitiven Typen – durch Literale gebildet werden.
- Literale und komplexere Ausdrücke vom Typ `String` können durch den (abermals überladenen!) Operator `+` konkateniert werden.

Mehrdimensionale Reihungen

Da auch Arrays einen bestimmten Typ haben (z.B. `gruss` : `char[]`), kann man auch Reihungen von Reihungen bilden. Mit einem Array von Arrays lassen sich z.B. Matrizen modellieren:

```
int[] m0 = {1, 2, 3};
int[] m1 = {4, 5, 6};
int[][] m = {m0, m1};
```