Überblick

Konstanten und Variablen

Konstanten

Erinnerung *funktionale Programmierung*:

- Ergebnis wird durch Anwenden/Auswerten einer Funktion auf die Eingabewerte ermittelt, Funktionen können dabei andere (Hilfs-)Funktionen aufrufen.
- Konstanten sind null-stellige Funktionen.
- Beispiel:
  - Berechne die Funktion $f(x)$ für $x \neq -1$ mit $f : \mathbb{R} \rightarrow \mathbb{R}$ gegeben durch

$$f(x) = (x + 1 + \frac{1}{x+1})^2 \quad \text{für } x \neq -1$$

  - Übersichtliche Lösung durch Aufteilung des Terms in:

$y_1 = x + 1$
$y_2 = y_1 + \frac{1}{y_1}$
$y_3 = {y_2}^2$

Konstanten (cont.)

- Realisierung in SML:

```
fun f(x) =
    let val y1 = x + 1.0;
        val y2 = y1 + (1.0 / y1);
        val y3 = y2 * y2;
    in
        y3
    end;
```

- `y1`, `y2` und `y3` sind *Konstanten* (In Info 1 auch: *funktionale Variablen*.)
- Intuition des Auswertungsvorgangs:
  - y_1, y_2 und y_3 bzw. `y1`, `y2` und `y3` repräsentieren drei Zettel.
  - Auf diese Zettel werden der Reihe nach Rechenergebnisse geschrieben.
  - Bei Bedarf wird der Wert auf dem Zettel abgelesen.

Variablen

- Bei genauerer Betrachtung:
  - Nachdem der Wert von y_1 zur Berechnung von y_2 benutzt wurde, wird er im folgenden nicht mehr benötigt.
  - Eigentlich könnte der Zettel nach dem Verwenden (Ablesen) “radiert” und für die weiteren Schritte wiederverwendet werden.
  - In diesem Fall kämen wir mit einem Zettel y aus.
- y heißt *Variable*.
- Im Unterschied zu Konstanten sind Variablen “radierbar”.

Variablen (cont.)

- Die übersichtliche Lösung zur Berechnung von $f(x)$ mit einer Variable y:

  $y = x + 1$
  $y = y + \frac{1}{y}$
  $y = y^2$

- Variablen wurden in Info 1 unter dem Begriff *Zustandsvariablen* behandelt.

- Variablen und Konstanten können
  - *deklariert* werden, d.h. ein “leerer Zettel” (Speicherzelle) wird angelegt
  - *intialisiert* werden, d.h. ein Wert wird auf den “Zettel” (in die Speicherzelle) geschrieben.
- Der Wert einer Variablen kann später auch noch durch eine *Wertzuweisung* verändert werden.
- Die Initialisierung enspricht einer (initialen) Wertzuweisung.
- Nach der Deklaration kann der Wert einer Konstanten nur noch durch eine Wertzuweisung verändert (initialisiert) werden.
- Nach der Deklaration kann der Wert einer Variablen beliebig oft durch eine Wertzuweisung verändert werden.

Konstanten und Variablen in Java

- Eine Variablendeklaration hat in Java die Gestalt
  `Typname variablenName;`
  Konvention: Variablennamen beginnen mit kleinen Buchstaben.
- Eine Konstantendeklaration hat in Java die Gestalt
  `final Typname KONSTANTENNAME;`
  Konvention: Konstantennamen bestehen komplett aus großen Buchstaben.
- Eine Wertzuweisung (z.B. Initialisierung) hat die Gestalt
  `variablenName = NeuerWert;`
  bzw.
  `KONSTANTENNAME = Wert;` (nur als Initialisierung)
- Eine Variablen- bzw. Konstantendeklaration kann auch mit der Initialisierung verbunden sein, d.h. der ersten Wertzuweisung.
  `Typname variablenName = InitialerWert;`
  (Konstantendeklaration mit Initialisierung analog mit Zusatz **`final`**)

Konstanten und Variablen in Java

- Konstanten:

```
final <typ> <NAME> = <ausdruck>;

final double Y_1 = x + 1; //Achtung: was ist x???
final double Y_2 = Y_1 + 1 / Y_1;
final double Y_3 = Y_2 * Y_2;
final char NEWLINE = '\\n';
final double BESTEHENSGRENZE_PROZENT = 0.5;
final int GESAMTPUNKTZAHL = 120;
```

- Variablen:

```
<typ> <name> = <ausdruck>;

double y = x + 1; //Achtung: was ist x???
int hausaufgabenPunkte = 17;
boolean klausurZulassung =
        ((double) hausaufgabenpunkte) /
          GESAMTPUNKTZAHL >= BESTEHENSGRENZE_PROZENT;
```

Zustandsänderungen von Variablen

Beispiel:

Anweisung	x	y	z
`int x;`			
`int y = 5;`		5	
`x = 0;`	0	5	
`final int z = 3;`	0	5	‖ 3 ‖
`x += 7 + y;`	12	5	‖ 3 ‖
`y = y + z - 2;`	12	6	‖ 3 ‖
`x = x * y;`	72	6	‖ 3 ‖
`x = x/z;`	26	6	‖ 3 ‖

Legende: [ ] = radierbar [‖ ‖] = nicht radierbar

Zusammenfassung

- Konstante
  - Entspricht in gewisser Weise im funktionalen Paradigma der funktionalen Variablen.
  - Zustand nicht veränderbar.
- Variable
  - Entspricht in gewisser Weise im funktionalen Paradigma der Zustandsvariablen (Referenz).
  - Zustand veränderbar.

Überblick

Anweisungen

- In der funktionalen Programmierung ist die elementare Einheit der *Ausdruck* (mit einem Wert).
- In imperativen Programmen sind *Anweisungen* die elementaren Einheiten.
- Eine Anweisung steht für einen einzelnen Abarbeitungsschritt in einem Algorithmus.

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.

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 sog. Lebensdauer. Das bedeutet, dass danach der entsprechende Speicherplatz, auf den die Variable verwiesen hat, im Prinzip wieder freigegeben ist (d.h. der Zettel wird weggeworfen).
- Solange eine lokale Variable sichtbar ist, darf keine neue lokale Variable gleichen Namens angelegt werden. Es gibt also bei lokalen Variablen (im Gegensatz zu SML) kein Überschatten.

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
- Inkrement
- Dekrement
- Methodenaufruf
- Instanzerzeugung (werden wir später kennenlernen)

Überblick

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), die den Funktionen der funktionalen Programmierung entsprechen. 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

Neben lokalen Variablen gibt es in Java auch globale Variablen, sog. *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 (und Konstanten) muss man Klassenvariablen (und Klassenkonstanten) 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`

Namen von lokalen Variablen und Klassenvariablen

Lokale Variablen innerhalb einer Klasse können genauso heißen wie eine Klassenvariable. Das ist aber keine Überschattung, 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`

Überblick

Eingebaute Datenstruktur

In einer Programmiersprache ist üblicherweise eine einfache Datenstruktur eingebaut, die es ermöglicht, eine Reihe von Werten (gleichen Typs) zu modellieren.

- In funktionalen Sprachen ist das normalerweise die Liste (vgl. SML).
  - Vorteil: dynamisches Wachstum
  - Nachteil: direkter Zugriff nur auf das erste Element möglich
- 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`.
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]); // gibt den Character 'd' aus
```


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); // gibt 2 aus
```

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 abstrakt gesehen?

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

Mehrdimensionale Reihungen

Da auch Arrays einen bestimmten Typ haben (z.B. `gruss` : **`char`**`[]`), kann man auch Reihungen von Reihungen bilden (wie wir in SML Listen von Listen bilden konnten). 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};
```