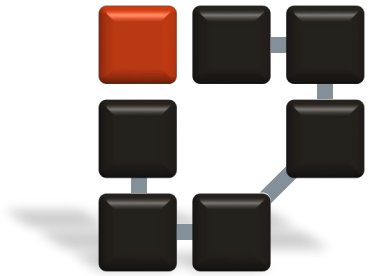
# Informatik 1 für Nebenfachstudierende Grundmodul

# Java – Datentypen und Variablen

Kai-Steffen Hielscher
Folienversion: 3. Dezember 2019



Informatik 7
Rechnernetze und
Kommunikationssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

# Inhaltsübersicht

- Kapitel 3 - Java
  - Einführung
  - Lexikalische Struktur
  - **Datentypen und Variablen**
  - Operatoren und Ausdrücke
  - Anweisungen und Ablaufsteuerung

# Datentypen und Variablen

- Variablen
- Konstanten
- primitive Datentypen
  - Wahrheitswerte (boolean)
  - Zeichen (char)
  - ganze Zahlen
    - byte
    - short
    - int
    - long
  - Fließkommazahlen
    - float
    - double
- Referenzdatentypen
  - Objekte (ganz kurz)
  - Zeichenketten
  - Arrays (kurz)

# Variablen

- Programme dienen der Verarbeitung von Daten
  - Dabei kann es sich um Zahlen (`3` oder `57.08`) wie um Zeichenketten `"Hallo Welt!"` oder die Hintergrundfarbe einer grafischen Darstellung handeln
  - Um Daten mittels eines Computer-Programms verarbeiten zu können, muss es eine Möglichkeit geben, wie man Daten im Programm zwischenspeichern kann
  - Dies geschieht im Arbeitsspeicher des Rechners
  - Er besteht aus einer Folge von einzelnen Speicherzellen, die fortlaufend nummeriert sind (Speicheradressen)
  - Versucht das Programm, ein Datum (Wert) im Speicher abzulegen, wird ein (je nach Typ des Datums) genügend großer Speicherplatz gesucht und das Programm merkt sich im Hintergrund die Anfangsadresse sowie die Größe des Speicherplatzes
  - Der Anwender muss sich nur (syntaktisch korrekte) Variablennamen ausdenken und diesen die Daten zuweisen. Beim Programmablauf werden die Variablennamen mit der entsprechenden Speicheradresse verbunden

# Variablen

- Variablen sind ein wesentlicher Bestandteil einer Programmiersprache
  - In Variablen werden Daten während der Laufzeit (Ausführung) eines Programms gespeichert
  - Die Daten können jederzeit verändert werden
    - durch Zuweisung anderer Daten bzw. Werte zu diesen Variablen
  - durch Berechnung mit verändertem Ergebniswert dieser Variablen
- Konstanten sind Variablen mit unveränderlichem Inhalt
  - damit eigentlich keine Variablen mehr
  - Sie werden nur einmal zugewiesen und lassen sich dann nicht mehr ändern

# Variablen

- Variablen dienen also zur Aufnahme veränderlicher Werte
  - Unsere Aufgabe als Programmierer:
    Repräsentation von Information durch Abbildung auf Variablen
  - Benutzung des Wertes der Variablen in
    - Operationen, z.B. `+`, `-`, `*`, `/`, …
    - Methoden, z.B.
      ```
      System.out.println("Heute hat es " + currTemp + " °C.");
      ```
- **Variablen: Basis der Datenspeicherung und Datenverarbeitung in Computern**

# Variablen in Java

- In Java muss man dem Compiler explizit mitteilen, dass man eine neue Variable erstellen will und welche Werte sie aufnehmen soll, damit dieser Speicherplatz dafür reservieren kann. Dies nennt man **Deklaration**. Variablen müssen **vor Benutzung** deklariert werden.
- Die Deklaration einer Variable hat die Form
  *`Variablentyp Variablenbezeichner;`*
  z.B. `float currTemp;`
- **Konvention**: Variablennamen beginnen in Java üblicherweise beginnend mit Kleinbuchstaben, jedes neue Wort innerhalb des Bezeichners beginnt mit Großbuchtstaben
  - z.B. `currTemp` oder `currentTemperature` oder `aktuelleTemperatur`

# Variablen in Java

- Wertzuweisung über Zuweisungsoperator `=`
  - z.B. `currTemp = -10.0;`
  - mögliche Werte hängen vom Typ der Variablen ab
  - **Initialisierung**: erste Wertzuweisung einer Variable
  - In Java ist die Initialisierung gleich bei der Deklaration möglich
    - anstelle von zwei Schritten
      ```
      float currTemp;
      currTemp = 10.0;
      ```
    - kann dies in einem Schritt geschehen
      ```
      float currTemp = 10.0;
      ```

# Primitive Datentypen

- Der **Typ** einer Variable legt fest, welche Daten die Variable aufnehmen kann
  - damit ist auch die Größe des zugehörigen Speicherplatzes festgelegt
  - ebenso, wie die Inhalte der Speicherzellen zu interpretieren sind
- aus primitiven (einfachen) Datentypen lassen sich **komplexe Datentypen** zusammensetzen, z.B. Objekte oder Arrays
- Ort der Deklaration legt den **Gültigkeitsbereich** (auch Geltungsbereich, scope) einer Variable fest
  - nur Programmteile im Gültigkeitsbereich können auf die Variable zugreifen, nur dort ist sie sichtbar
  - **globale Variablen** mit globalem Gültigkeitsbereich
  - **lokale Variablen** mit lokalem Gültigkeitsbereich

# Konstanten

- werden in Java ähnlich deklariert wie Variablen
- ihr Wert kann nach der ersten Wertzuweisung nicht mehr geändert werden
- Schlüsselwort **`final`**
  - z.B.

```
final int ZWEI = 2;
final double PI = 3.141592654;
```

- Konvention: Bezeichner von Konstanten in Großbuchstaben

# Primitive Datentypen

- Wahrheitswerte, boolesche Werte
  - Typ `boolean`
  - mögliche Werte `true` oder `false`
  - Größe im Speicher nicht festgelegt, eigentlich reicht ein Bit
  - Beispiel
    ```
    boolean rain = false;
    ```

# Primitive Datentypen

- Zeichen
  - Typ `char`
  - mögliche Werte: ein Unicode-Zeichen mit 16 Bit, kann auch als numerischer Wert zwischen 0 und 65535 zugewiesen werden
  - Größe im Speicher 16 Bit
  - Beispiel
    ```
    char zeichen = 'a';
    ```
    oder
    ```
    char zeichen = 'µ';
    ```
    oder als ASCII- bzw. Unicode-Wert
    ```
    char zeichen = 65;
    ```

# Primitive Datentypen

- ganze Zahlen
  - unterschiedliche primitive Datentypen für unterschiedlich große ganze Zahlen
    - unterschiedlicher Wertebereich
    - unterschiedliche Größe im Speicher

# Primitive Datentypen

- ganze Zahlen
  - Typ `byte`
  - mögliche Werte: -128 bis 127
  - immer mit Vorzeichen (Zweierkomplement-Wert)
  - Größe im Speicher 8 Bit
  - Beispiel
    ```
    byte b = 7;
    ```
    oder
    ```
    byte b = -7;
    ```
    oder als Hexadezimal-Literal
    ```
    byte b = 0xEE;
    ```

# Primitive Datentypen

- ganze Zahlen
  - Typ `short`
  - mögliche Werte: -32768 bis 32767
  - immer vorzeichenbehaftet (Zweierkomplement-Wert)
  - Größe im Speicher 16 Bit
  - Beispiel
    ```
    short si = 700;
    ```
    oder
    ```
    short si = -700;
    ```
    oder als Hexadezimal-Literal
    ```
    short si = 0x11AA;
    ```

# Primitive Datentypen

- ganze Zahlen
  - Typ `int`
  - mögliche Werte: -2147483648 bis 2147483647
  - immer vorzeichenbehaftet (Zweierkomplement-Wert)
  - Größe im Speicher 32 Bit
  - Beispiel
    ```
    int i = 1;
    ```
    oder
    ```
    int i = -65600;
    ```

# Primitive Datentypen

- ganze Zahlen
  - Typ `long`
  - mögliche Werte: $-2^{63}$ bis $2^{63}-1$
  - vorzeichenbehaftet (Zweierkomplement-Wert) , ab Java 8 auch vorzeichenlos möglich (Wertebereich 0 bis $2^{64}-1$)
  - Größe im Speicher 64 Bit
  - Beispiel

```
long l = 13456789000;
```

# Primitive Datentypen

- Fließkommazahlen (Gleitkommazahlen)
  - Typ `float`
  - floating point number
  - mögliche Werte: $-3.40282347 \cdot 10^{38}$ bis $3.40282347 \cdot 10^{38}$
  - kleinstmöglicher positiver Wert: $1.40129846432 \cdot 10^{-45}$
  - Größe im Speicher 32 Bit
  - beschränkte Genauigkeit: Rundung
  - Beispiel

```
float pi = 3.14;
float x = 1.78e-12;
float f = 1.0f;
```

# Primitive Datentypen

- Fließkommazahlen (Gleitkommazahlen)
  - Typ `double`
  - doppelt lange Fließkommazahl
  - mögliche Werte: $-1.7976931348623157 \cdot 10^{308}$ bis $1.7976931348623157 \cdot 10^{308}$
  - kleinstmöglicher positiver Wert: $4.940656458412465 \cdot 10^{-324}$
  - Größe im Speicher 64 Bit
  - beschränkte Genauigkeit: Rundung
  - Beispiel

```
double pi = 3.141592654;
double n = 6.022140857E23;
double d = 1.0d;
```

# Primitive Datentypen

- für all diese primitiven Datentypen existieren Wrapper-Klassen
  - umhüllen die primitiven Datentypen
  - ermöglichen es, primitive Datentypen als Objekte zu behandeln
  - bieten zusätzliche Funktionalität in den Methoden
  - Name: oft wie primitiver Datentyp, nur mir erstem Buchstaben als Großbuchstaben, manchmal ausgeschrieben
    - `Boolean`
    - `Character`
    - `Byte`, `Short`, `Integer`, `Long`
    - `Float`, `Double`

## Referenzdatentypen

- eine Variable für einen **primitiven Datentyp** verweist auf einen Bereich im Hauptspeicher, der **direkt den Wert der Variable** enthält
- eine Variable für einen **Referenzdatentypen** verweist auf einen Bereich im Speicher, der die **Adresse des referenzierten Objekts** enthält

# Objekte – ganz kurze Einführung

- Objekte sind Instanzen einer Klasse
- Deklaration einer Variable für ein Objekt einer vorhandenen Klasse wie bei Variablen mit primitivem Datentyp
  `Klassenname Variablenbezeichner;`
- dabei entsteht eine **Referenz** auf diese Objekt, noch kein Objekt
- z.B.
  `Integer i;`
- Erzeugung eines neuen Objekts dieser Klasse: **Instantiierung** mittels `new`
- ruft eine spezielle Methode der Klasse auf, den **Konstruktor**
  - legt u.a. die Initialwert fest
  - oft **Argument** für den Konstruktor nötig, der den initialen Wert angibt
  - z.B.
    `i = new Integer(5);`
- Deklaration und Instantiierung auch in einem Schritt möglich, z.B.
  `Integer i = new Integer(5);`

# Objekte

- Objekte können Daten enthalten
- Komponentenvariablen
  - Instanzvariablen (individueller Wert für jede Instanz)
  - Klassenvariablen (gleicher Wert für die ganze Klasse)
- Zugriff auf Komponentenvariablen einer Instanz (einem Objekt)

  *Objektbezeichner.Variablenname*

# Objekte

- z.B.
  - Deklaration einer Klasse mit Instanzvariablen

```
public static class Hausanschrift {
    public String strasse;
    public int hausnummer;
}
```

  - Erzeugung einer Instanz

```
Hausanschrift whhh;
whhh = new Hausanschrift();
```

  - Zugriff auf Komponentenvariablen

```
whhh.strasse = "Martensstr.";
whhh.hausnummer = 3;
```

# Objekte

- Neben Daten enthält eine Klasse auch Anweisungen in Methoden
- Methodenaufruf an einer Instanz (einem Objekt)
  ```
  Objektbezeichner.Methodenname([Argument]);
  ```
- z.B.
  ```
  i.toHexString();
  ```
  oder
  ```
  i.parseString("3.14");
  ```

# Objekte

- vordefinierte Klassen in den Bibliotheken von Java
- Klassen sind in Paketen (packages) organisiert
- müssen bei Bedarf importiert werden
  - `import`-Anweisung
  - erlaubt die Nutzung im eigenen Programm, bindet die Bibliothek ein
  - Pakete sind hierarchisch gegliedert, Trennung der Ebenen durch Punkte
  - z.B.
    ```
    import java.util.StringTokenizer;
    ```
- Dokumentation: Java-API-Dokumentation
  https://docs.oracle.com/javase/8/docs/api/
  (API: Application Programming Interface)
- es können auch eigene Pakete erstellt werden, die eigene Klassen enthalten

# Zeichenketten

- können ein Folgen von Zeichen (`char`) aufnehmen
- Zeichenketten in Java sind Objekte der Klasse `String`
- initiale Zuweisung eines Wertes direkt möglich, kein expliziter Aufruf des Konstruktors nötig
  - z.B.
    ```
    String hello = "Hello World!";
    ```
- String-Objekte sind unveränderlich, will man z.B. weiteren Text anhängen, wird eine Kopie erzeugt
- als Objekte besitzen Strings Methoden, die man aufrufen kann
  - z.B.
    ```
    int laenge = hello.length();
    ```

# Arrays (Felder)

- Ein Array ist eine Folge von Variablen mit gleichem Typ
  - elementare Datentypen
  - Instanzen einer Klasse
- feste Länge
- Deklaration
  ```
  Datentyp[] Arraybezeichner;
  ```
  - z.B.
    ```
    int[] array;
    ```
- hierdurch wird eine **Referenz** auf das Array angelegt
- danach muss das eigentliche Array mit `new` erzeugt werden, dabei wird die Anzahl der Elemente angegeben
  ```
  Arraybezeichner = new Datentyp[Elemente];
  ```
  - z.B.
    ```
    array = new int[10];
    ```

# Arrays

- meist in einem Schritt
  - z.B.
    ```
    int[] array = new int[10];
    ```
- Jetzt wird auch klar, warum hier eine Referenz benötigt wird: Vor Erzeugung des Arrays mit `new` ist nicht klar, wie groß das Array sein wird, d.h. wie viel Speicher dafür benötigt wird
- Zugriff auf Element `i` mittels *`array[i]`*
  - in obigem Beispiel kann `i` Werte von 0 bis 9 annehmen
  - z.B.
    ```
    array[0] = 1;
    array[5] = 4711;
    array[9] = 42;
    ```
- **Achtung: die Nummerierung der Elemente beginnt bei 0.**

# Arrays

- es können auch Arrays von Objekten einer Klasse erzeugt werden
- beim Erzeugen des Arrays entsteht eine Reihe von Objektreferenzen, die Objekte müssen danach erst instantiiert werden
- z.B.

```
Double[] d = new Double[3];
d[0] = new Double(0.0);
d[1] = new Double(1.0);
d[2] = new Double(2.0);
```

# Arrays

- auch mehrdimensionale Arrays möglich
- z.B.

```
double[][] matrix = new double[3][3];
matrix[0][0] = 1.0;
matrix[0][1] = 0.0;
matrix[0][2] = 0.0;
matrix[1][0] = 0.0;
matrix[1][1] = 1.0;
matrix[1][2] = 0.0;
matrix[2][0] = 0.0;
matrix[2][1] = 0.0;
matrix[2][2] = 1.0;
```

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

# Arrays

- auch Arrays besitzen Methoden und Komponentenvariablen
- z.B. Anzahl der Elemente eines Arrays als Instanzvariable

```
int[] array = new int[10];
int numelem = array.length;
```

# Arrays

- Deklaration und Initialisierung in einem Schritt möglich, dann kein `new` nötig
- Werte des zu erstellenden Arrays in geschweiften Klammern
  - z.B.
    ```
    int[] feld = {0, 1, 2, 3, 4};
    ```

# Arrays

- Zuweisungen erzeugen nur eine neue Referenz, keine Kopie der Elemente des Arrays
  - z.B.
    ```
    int[] feld = {0, 1, 2, 3, 4};
    int[] feld2 = feld;
    feld2[0] = 5;
    ```
  - im Beispiel verweisen dann sowohl `feld` als auch `feld2` auf die gleichen Elemente, sowohl `feld2[0]` als auch `feld[0]` hat nun den Wert `5`.

## Referenzdatentypen

- bei Referenzdatentypen müssen die Objekte mit `new` erzeugt werden (bei Strings nicht explizit)
- dabei wird ein Speicherbereich für diese Objekte reserviert
- **Garbage Collector** von Java gibt den Speicher automatisch frei, wenn er nicht mehr benötigt wird
  - wenn keine Referenz mehr auf diesen Speicherbereich verweist