

**252-0027**

# **Einführung in die Programmierung**

## **Übungen**

## **Klassen**

**Henrik Pätzold**

**Departement Informatik**

**ETH Zürich**

# Heutiger Plan

- Änderungen bei Bewertungen (RequestFeedback.txt)
- Theorie
  - Felder
  - Methoden
  - Konstruktoren
  - **this**-Schlüsselwort
  - Sichtbarkeit
  - Getter- und Setter-Methoden
- Kurze Programmieraufgabe
- Kahoot

# Referenzen 101

- In Java läuft **alles** über Klassen
- Klassen sind „Baupläne“, die definieren, wie Daten gespeichert werden
- Objekte sind konkrete Realisierungen der Klasse
- Wie speichern wir Objekte?

```
1 public class Main {  
2     public static void main(String[] args){  
3         Coordinate coord = new Coordinate(2,3);  
4         System.out.println(coord);  
5     }  
6 }  
7 public class Coordinate {  
8     int x;  
9     int y;  
10    public Coordinate(int xcoord, int ycoord){  
11        x = xcoord;  
12        y = ycoord;  
13    }  
14 }
```

# Primitive D.t.

- speichern tatsächlichen Wert
- sind von der Größe begrenzt
- Standardwerte sind festgelegt
- Können niemals **NULL** sein
- Liegen auf dem *Stack*

# Objekte

- speichern Referenz auf Speicherort
- Größe ist variabel
- Standardwert ist **NULL** (keine Referenz in den Speicher)
- Referenz liegt auf dem *Stack*
- Objekt selbst liegt auf *Heap*

# Weitergabe von Referenzen

- In Java ist **alles** call-by-value
  - Entscheidend ist welches **value** gespeichert wird
- **numbers** speichert die Referenz, wo die tatsächlichen Werte liegen
- gib Referenz an **modifyArray** weiter
- **modifyArray** sucht Werte im Speicher, verdoppelt sie, terminiert
- **main** sucht mit derselben im Speicher und findet verdoppelte Werte **ohne**, dass wir sie zurückgeben mussten
- Call-by-value bedeutet unterschiedliches Verhalten von =, == bei primitiven gegenüber Objekten

```
1 public class ReferenceArrayDemo {
2     public static void main(String[] args) {
3         // Erstelle ein Array
4         int[] numbers = {1, 2, 3, 4, 5};
5         System.out.println("Vor der Änderung:");
6         System.out.println(Arrays.toString(numbers));
7         modifyArray(numbers); // Übergabe des Arrays
8         System.out.println("Nach der Änderung:");
9         System.out.println(Arrays.toString(numbers));
10    }
11    public static void modifyArray(int[] arr) {
12        for (int i = 0; i < arr.length; i++) {
13            arr[i] *= 2; // Verdopple jeden Wert im Array
14        }
15    }
```


# Ausnahmen - Unveränderlichkeit von Strings

- Strings sind immutable, also wirkt es, als ob sie „by value“ übergeben würden.
- bei Veränderung wird Kopie mit selbem Wert erstellt (mit neuer Referenz)
  - auch bei Anwendung von Funktionen (concat, substring) wird immer eine Kopie erstellt

```
1 public class StringDemo {  
2     public static void main(String[] args) {  
3         String a = "Hello";  
4         System.out.println(a); // Hello  
5         modifyString(a);  
6         System.out.println(a); // Hello  
7     }  
8     public static void modifyString(String a) {  
9         a = a + " World!";  
10        System.out.println(a); // Hello World!  
11    }  
12 }
```

# Wie definieren wir eine Klasse?


- Klassen sind „Baupläne“, die definieren, wie Daten gespeichert werden
- Wir benutzen das **class-Stichwort**, um eine Klasse zu definieren



```
1 public class Student {  
2     // Körper der Klasse  
3 }
```

# Erstellung von Attributen (Felder)

- Attribute definieren den Zustand oder die Eigenschaften einer Klasse
- Ihnen kann normal ein Wert zugewiesen werden
- Ohne Zuweisung durch Konstruktor erhalten Sie bei Instanziierung sonst den Standardwert des Datentyps (**null** für Objekte)

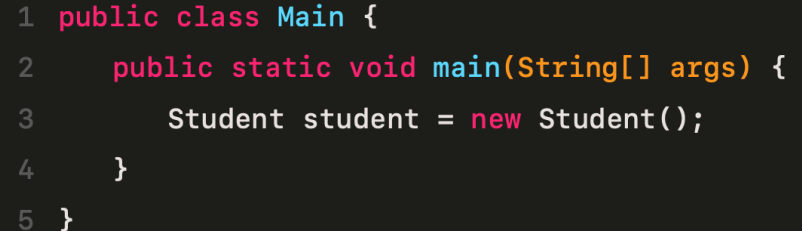


```
1 public class Student {  
2     String name;  
3     int alter;  
4     String legi;  
5     boolean istEingeschrieben = true;  
6 }
```



# Erzeugen eines Objekts aus einer Klasse

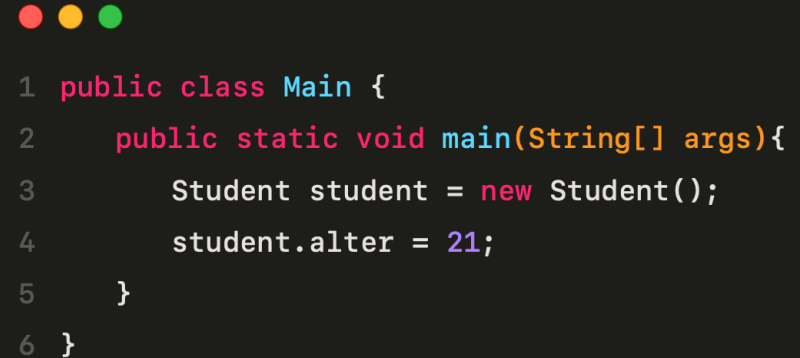
- Ein Objekt wird erstellt in dem der Konstruktor aufgerufen wird
- Wenn Klassen keinen Konstruktor haben, wird ein Default-Konstruktor verwendet
- Wenn der Default-Konstruktor verwendet wird, werden Instanzvariablen der Klasse auf Standardwerte gesetzt
  - Alle nicht initialisierten Felder werden vor Ausführung eines Konstruktors auf Standardwerte gesetzt
- Der Konstruktor erstellt das Objekt, das `new` liefert die Referenz auf das Objekt



```
1 public class Main {  
2     public static void main(String[] args) {  
3         Student student = new Student();  
4     }  
5 }
```

# Zugriff auf Felder eines Objektes


- Wenn die Sichtbarkeit es erlaubt, können wir auf ein Attribut eines Objekts mit der Punktnotation zugreifen: `objektname.attributname`
- wir bekommen über diese Notation Lese- und Schreibzugriff
- Ausnahme: `final` deklarierte Variablen (keine weiteren Veränderungen)



```
1 public class Main {  
2     public static void main(String[] args){  
3         Student student = new Student();  
4         student.alter = 21;  
5     }  
6 }
```

# Erstellung von Methoden

- Methoden sind Funktionen, die innerhalb einer Klasse definiert sind
- Methoden haben uneingeschränkten Zugriff auf die Attribute und andere Methoden der Klasse, in der sie definiert sind.




```
1 public class Student {  
2     String name;  
3     int alter;  
4     String legi;  
5     boolean istEingeschrieben = true;  
6     public void hasBirthday(){  
7         this.age += 1;  
8     }  
9 }
```

# Parametrisierte Konstruktor-Methoden

- **default**-Konstruktor kann mit expliziter Beschreibung überschrieben werden
- Konstruktoren können unterschiedliche Sichtbarkeit haben, sind aber meist **public**
- Methoden-Name ist gleich der Klasse
- Ein Konstruktor hat keinen Rückgabetyp (nicht einmal void).
- `new` erzeugt ein neues Objekt & liefert eine Referenz auf dieses Objekt.

# Parametrisierte Konstruktor-Methoden



```
1 public class Student {  
2     String name;  
3     int alter;  
4     String legi;  
5     boolean istEingeschrieben = true;  
6  
7     public Student(String StudentName, int StudentAlter, String StudentLegi) {  
8         name = StudentName;  
9         alter = StudentAlter;  
10        legi = StudentLegi;  
11    }  
12 }
```

```
public int add(int a, int b){ ...
```

**public** **int** **add**(**int** a, **int** b){ ...

The diagram illustrates the components of the function signature `public int add(int a, int b){ ...}` using colored brackets and labels:

- A red bracket under `public` is labeled **Sichtbarkeit** (Visibility).
- A green bracket under `int` is labeled **Rückgabewert** (Return value).
- A yellow bracket under `add` is labeled **Name** (Name).
- A yellow bracket under `(int a, int b)` is labeled **Funktionsparameter** (Function parameter).

Methodensignatur

**public** **int** **add**(**int** a, **int** b){ ...

Sichtbarkeit Rückgabewert Name Funktionsparameter

The diagram illustrates the components of a Java method signature. The signature is 'public int add(int a, int b){ ...'. Brackets are used to group parts of the signature: a red bracket under 'public' is labeled 'Sichtbarkeit'; a green bracket under 'int' is labeled 'Rückgabewert'; a yellow bracket under 'add' is labeled 'Name'; and a yellow bracket under '(int a, int b)' is labeled 'Funktionsparameter'. A large black bracket above the entire signature is labeled 'Methodensignatur'.




# Method-Overloading

- Eine Funktion kann überladen werden, in dem die Funktionsparameter verändert werden
- trotz gleichem Namen hat sie dann eine andere Signatur
- **add** kann durch Überladung mit unterschiedlichen Parametern aufgerufen werden, hat nun unterschiedliche Rückgabetypen

```
1 public class Main {  
2     // Erste Version: Addiert zwei int-Werte  
3     public static int add(int a, int b) {  
4         return a + b;  
5     }  
6     // Überladene Version: Addiert drei int-Werte  
7     public static int add(int a, int b, int c) {  
8         return a + b + c;  
9     }  
10    // Überladene Version: Addiert zwei double-Werte  
11    public static double add(double a, double b) {  
12        return a + b;  
13    }  
14 }
```

**Methoden (auch Konstruktoren) können  
überladen werden**

# Parametrisierte Konstruktor-Methoden




```
1 public class Student {  
2     String name;  
3     int alter;  
4     String legi;  
5     boolean istEingeschrieben = true;  
6  
7     public Student(String StudentName, int StudentAlter, String StudentLegi) {  
8         name = StudentName;  
9         alter = StudentAlter;  
10        legi = StudentLegi;  
11    }  
12    public Student(String StudentName, int StudentAlter) {  
13        name = StudentName;  
14        alter = StudentAlter;  
15    }  
16 }
```

# Das **this**-Schlüsselwort

- **verweist auf das aktuelle Objekt, auf dem eine Methode oder ein Konstruktor ausgeführt wird.**
- **Wir können this verwenden, um:**
  - auf die **Attribute (Felder)** des aktuellen Objekts zuzugreifen
  - **Methoden** desselben Objekts aufzurufen
  - einen **anderen Konstruktor** derselben Klasse aufzurufen (`this(...)`)
- **Besonders nützlich, wenn Parameter denselben Namen wie Felder haben. (schafft Klarheit)**

# this-Schlüsselwort für Zuweisung von Feldern



```
1 public class Student {  
2     String name;  
3     int alter;  
4     String legi;  
5     boolean istEingeschrieben = true;  
6  
7     public Student(String name, int alter, String legi) {  
8         this.name = name;  
9         this.alter = alter;  
10        this.legi = legi;  
11    }  
12 }
```

# this-Schlüsselwort für Aufrufen von Methoden



```
1 public class Student {  
2     String name;  
3     int alter;  
4     String legi;  
5     boolean istEingeschrieben = true;  
6  
7     public Student(String name, int alter, String legi) {  
8         this.Student(String name, int alter);  
9         this.legi = legi;  
10    }  
11    public Student(String name, int alter){  
12        this.name = name;  
13        this.alter = alter;  
14    }  
15 }
```

# Sichtbarkeit von Methoden und Feldern

- nützliches Konzept, um bestimmte Informationen zu kapseln
  - gewährleistet korrektes und sicheres Verhalten von Objekten
- **private** deklarierte Strukturen sind nur innerhalb der Klasse sichtbar
- auf **public** deklarierte Strukturen kann von jeder anderen Klasse aus zugegriffen werden
- Ausnahmefälle: **default** und **protected**
- In der Praxis aber fast immer alles **private** => Getter- und Setter-Methoden(Konvention)



```
1 public class Student {  
2     private String name;  
3     int alter;  
4     private String legi;  
5     public boolean istEingeschrieben = true;  
6 }
```

# Getter- und Setter-Methoden

- wir nutzen die Sichtbarkeit zum „Kapseln“ Feldern
- **getter**-Methoden geben Werte zurück => verhindern, dass wir sie überschreiben können
- **setter**-Methoden sind eine Sicherheitsbarriere, die korrektes überschreiben garantieren können

```
1 public class Student {  
2     private int alter;  
3     public int alter(){  
4         return this.alter;  
5     }  
6     public void setAlter(int alter) {  
7         if(alter >= 0){  
8             this.alter = alter;  
9         }  
10    }  
11 }
```



# Objektorientierte Grundlagen - Zusammenfassung

- Klassen beschreiben den Bauplan für Objekte.
- Objekte besitzen Zustand (Felder) und Verhalten (Methoden).
  - Konstruktoren erzeugen Objekte.
- Sichtbarkeit + Getter/Setter = Encapsulation.
- **this** = Verweis auf das aktuelle Objekt.

# **Vorbesprechung Übung 5 – Bonus & Zusätzliches Material**

# **Kleine Programmiersimulation nächste Woche**

# Kahoot