Technische Universität Braunschweig

Programmieren 1 – Vorlesung #5

Arne Schmidt

Wiederholung

Letzte Woche

Referenztypen:
Array, Strings und
Enum

Objektorientierte
Programmierung

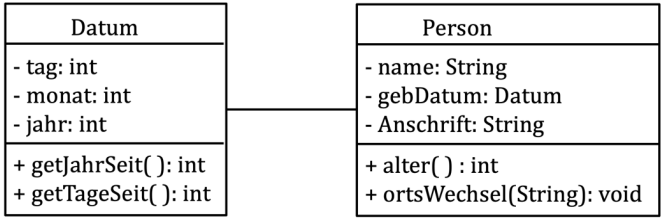
Kapselung

6	9	1	8	5	2	7	0	3

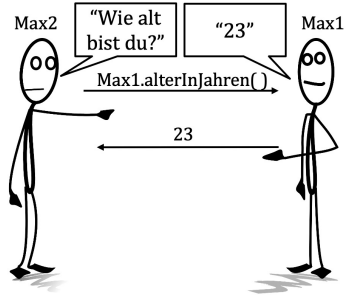
“Das ist ein String <(o.o<)”

{Jan, Feb, Mär, Apr, Mai, Jun, Jul, …}

== und equals()

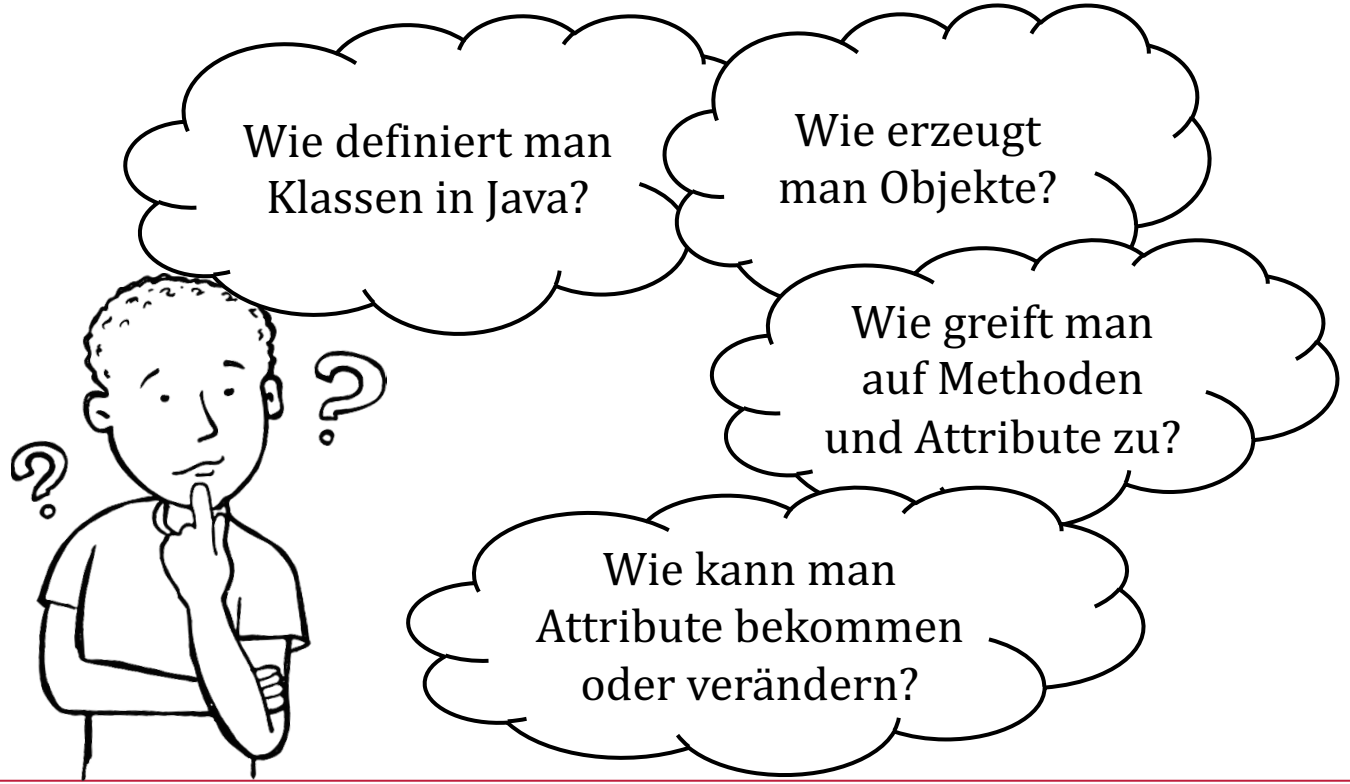


Beziehungen



Austausch von Nachrichten

Heute


Wie definiert man Klassen in Java?
Wie erzeugt man Objekte?
Wie greift man auf Methoden und Attribute zu?
Wie kann man Attribute bekommen oder verändern?

Kapitel 4.1 – Klassen in Java

Typischer Aufbau

Eine Klasse in Java wird mit dem Schlüsselwort class eingeleitet und wird von dem Klassenbezeichner gefolgt. In geschweiften Klammern folgt dann die Klassendefinition.

Typische Reihenfolge:

1. Attribute
2. Konstruktoren
3. Get- und Set-Methoden
4. Andere Methoden

```
class MyClass{
   attribut1;
   ....
   attributN;

   Konstruktor1( );
   ...
   KonstruktorM( );
   get- / setMethoden
   weitere Methoden
}
```

Erzeugen von Objekten

Objekte werden **immer** mit dem Schlüsselwort new erzeugt (Ausnahme: Strings und Arrays)

```
MyClass var;
var = new MyClass( );
```

Oder einfach

```
MyClass var = new MyClass( );
```

Bei Erzeugung ("new MyClass()") werden zunächst alle Attribute auf den Standartwert gesetzt:

- Referenztypen besitzen den Wert null
- Primitive Datentypen den jeweiligen Standartwert

Bei der Klassendefinition können Standardwerte für Attribute vordefiniert werden.

```
class Datum{
   int tag = 1;
   int monat = 1;
   int jahr = 1900;
   …
}
```

Kleines Beispiel

```
class Word{
   public String s;
}

public class main {
   public static void main(String[] args) {
      Word w = new Word();

      if (w.s == null) {
         System.out.println(“Oops, String is null!");
      }
   }
}
```

Gibt aus:
Oops, String is null!

Objekte – Zugriff auf Attribute

Der Zugriff auf Attribute (und auch Methoden) erfolgt mittels Punktnotation:

```
Datum date = new Datum( );
date.tag = 18;
date.monat = 8;
date.jahr = 2000;
```

Setzt das Datum auf den 18.8.2000.

```
class Datum{
   int tag = 1;
   int monat = 1;
   int jahr = 1900;
   …
}
```

Referenz auf Objekte

Variablen für Referenzobjekte enthalten nur eine Referenz (Zeiger), wo im Speicher das Objekt liegt.

Bei Zuweisung zu einer zweiten Variablen, z.B. Datum date2 = date; wird nur der Zeiger kopiert. Ändert date2 Attribute, sind diese auch über date sichtbar.

Damit kann es prinzipiell beliebig viele Variablen geben, die auf ein konkretes Objekt zeigen.

Von einer Klasse können beliebig viele Objekte erzeugt werden.

Kapitel 4.1.1 – Objektmethoden

Objektmethoden

Wie auch normale Methoden (siehe VL 3), werden Objektmethoden folgendermaßen definiert:
Modifikatoren Rückgabetyp Bezeichner(Liste mit Parametertypen inkl. Bezeichnern)

Beispiel für die Klasse Datum:

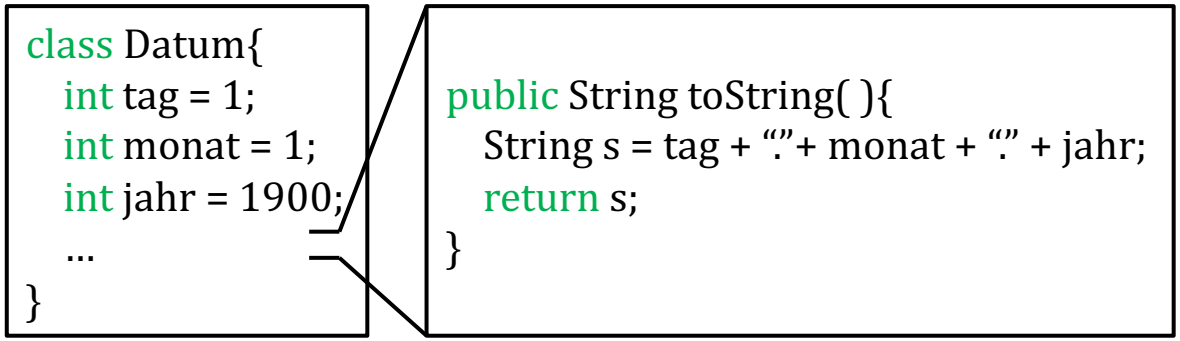
```
public String toString( ){
   String s = tag + "."+ monat + "." + jahr;
   return s;
}
```

```
class Datum{
   int tag = 1;
   int monat = 1;
   int jahr = 1900;
   …
}
```

Eine Objektmethode hat immer direkten Zugriff auf alle in der Klasse definierten Attribute.

Die **Modifikatoren** (zb public oder static) von Methoden (und auch Attributen) schauen wir uns im Detail nächste Woche an!

Objektmethoden – Rückgabetyp und Aufruf



Die Angabe des Rückgabetyps ist notwendig. Gibt eine Methode nichts zurück, ist der Rückgabetyp *void*.

Der Aufruf findet wieder mittels Punktnotation statt:

```
Datum date = new Datum( );
System.out.println(date.toString());
```

Was wird ausgegeben?

“1.1.1900”

Parameterübergabe (Reprise)

Zur Erinnerung:
Parameterübergabe findet mit Pass-Per-Value statt, d.h. der *Wert* der Variablen wird in die Parametervariable kopiert.

Vor myArray.arrayChange(array);

array in main → [1, 2, 3]

array in arrayChange

myArray.arr → [ ]

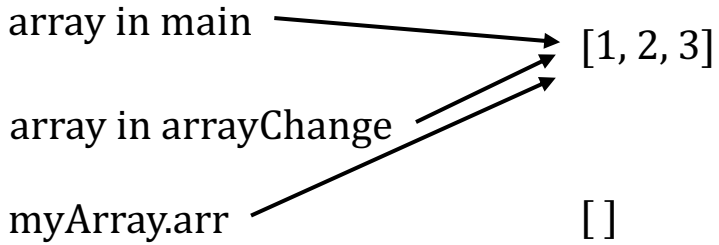
```
class MyArray{
   int[ ] arr = { };
   public void arrayChange(int[] array){
      arr = array;
      arr[array.length - 1] = 9;
   }
}

public class Main{
   public static void main(String[] args){
      MyArray myArray = new MyArray();
      int[] array = {1, 2, 3};
      myArray.arrayChange(array);
   }
}
```

Parameterübergabe (Reprise)

Zur Erinnerung:
Parameterübergabe findet mit Pass-Per-Value statt, d.h. der *Wert* der Variablen wird in die Parametervariable kopiert.

Vor arr[array.length - 1] = 9;



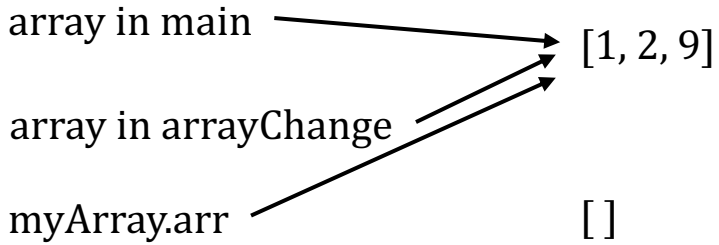

```
class MyArray{
   int[ ] arr = { };
   public void arrayChange(int[] array){
      arr = array;
      arr[array.length - 1] = 9;
   }
}

public class Main{
   public static void main(String[] args){
      MyArray myArray = new MyArray();
      int[] array = {1, 2, 3};
      myArray.arrayChange(array);
   }
}
```

Parameterübergabe (Reprise)

Zur Erinnerung:
Parameterübergabe findet mit Pass-Per-Value statt, d.h. der *Wert* der Variablen wird in die Parametervariable kopiert.

Nach arr[array.length - 1] = 9;




```
class MyArray{
   int[ ] arr = { };
   public void arrayChange(int[] array){
      arr = array;
      arr[array.length - 1] = 9;
   }
}

public class Main{
   public static void main(String[] args){
      MyArray myArray = new MyArray();
      int[] array = {1, 2, 3};
      myArray.arrayChange(array);
   }
}
```

Parameterübergabe – Verhindern des Seiteneffekts

Zur Erinnerung:
Parameterübergabe findet mit Pass-Per-Value statt, d.h. der *Wert* der Variablen wird in die Parametervariable kopiert.

Pass-Per-Value mit Referenzen ist effizient, da nicht das komplette Objekt erst kopiert werden muss.

Möchte man explizit auf einer Kopie arbeiten, muss man obj.clone() übergeben.

```
class MyArray{
   int[ ] arr = { };
   public void arrayChange(int[] array){
      arr = array;
      arr[array.length - 1] = 9;
   }
}

public class Main{
   public static void main(String[] args){
      MyArray myArray = new MyArray();
      int[] array = {1, 2, 3};
      myArray.arrayChange(array.clone());
   }
}
```

Methoden Return-Typ

Besitzt die Methode einen Rückgabewert, wird ein solcher Wert mittels

return *Ausdruck*;

zurückgegeben werden. *Ausdruck* muss dabei von gleichem Typ sein, wie der Rückgabetyp der Methode.

Ist der Rückgabetyp void, muss keine return-Anweisung erfolgen. Ein return kann aber als vorzeitiger Abbruch der Methode verwendet werden.

void-Funktionen dürfen nicht in Ausdrücken verwendet werden und basieren hautpsächlich auf Seiteneffekten.

Überladen von Methoden

Zur Erinnerung:

Die **Signatur** eine Methode ist die Kombination von Methodennamen und die Ordnung der Parametertypen.

Der Compiler kann Methoden mit gleichem Namen also unterscheiden, wenn die Parameter verschieden sind.

Benutzt eine Klasse mehrere Methoden mit gleichem Namen, heißt das **überladen**.

Damit sind Methoden erlaubt wie:

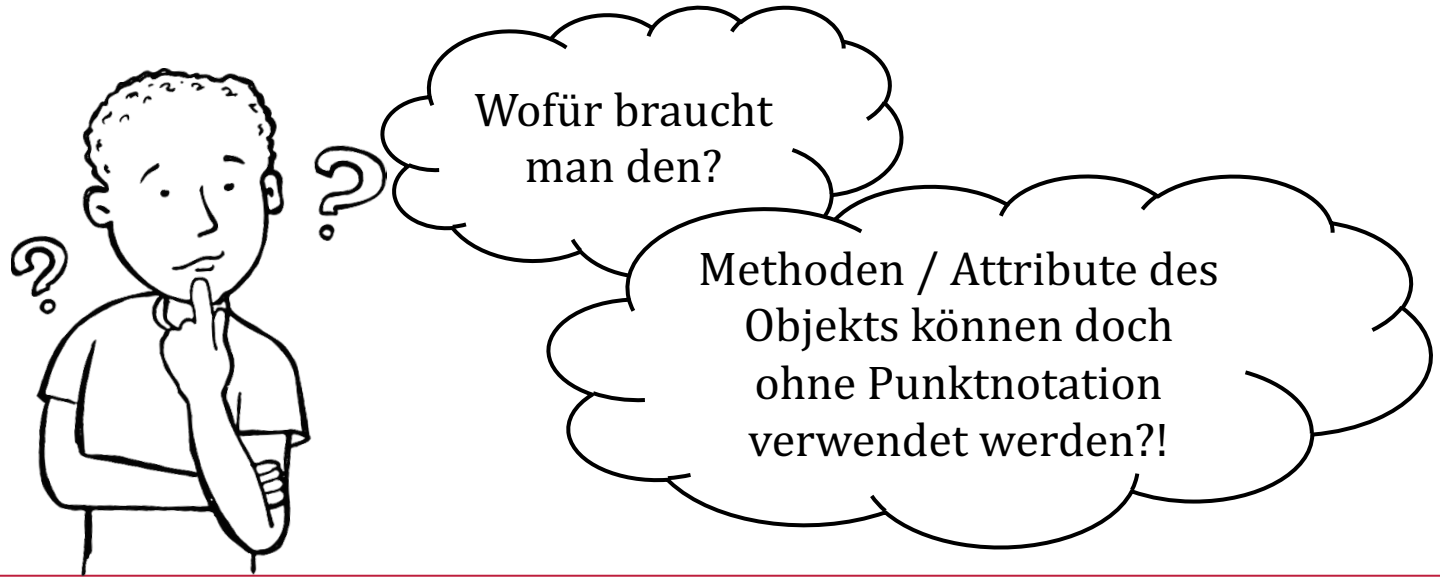
- double f(int a, int b){return 1.0;}
- double f(double a, double b) {return 2.0;}
- float f(float a, float b) {return 3.0f;}

Aber nicht zusätzlich:

- int f(int a, int b) {return 1;}

Der this-Zeiger

Der **this-Zeiger** ist eine Referenzvariable innerhalb einer Objektmethode, die immer auf das aktuelle Objekt zeigt und kann wie eine Variable genutzt werden.



Kapitel 4.1.2 – Spezielle und typische Methoden: Konstruktoren und Destruktoren

Konstruktoren

Der **Konstruktur** ist eine Methode zur Erzeugung von Objekten. Weiter gilt:

- Der Bezeichner eines Konstruktors ist identisch mit dem Klassennamen.
- In Java besitzt der Konstruktor keinen Rückgabetyp.
- Konstruktoren können Parameter besitzen und überladen werden.
- Existiert kein Konstruktor, liefert der Compiler einen Default-Konstruktor
- Konstruktoren können **verkettet** werden.

Beispiel:

```
class Datum{
   int tag;
   int monat;
   int jahr;
```

```
Datum( ){
   tag = 1;
   monat = 8;
   jahr = 2000;
}
```

```
Datum(int tag, int monat, int jahr){
   this.tag = tag;
   this.monat = monat;
   this.jahr = jahr;
}
}
```

Verkettung von Konstruktoren

Innerhalb eines Konstruktors kann auch ein anderer Konstruktor zuerst ausgeführt werden. Dazu dient der Aufruf this(*parameter*).

Dadurch können Attribute, die nicht zu den Parametern passen, zunächst gewünschte Default-Werte beispielsweise über this( ) gesetzt werden. Das spart doppelten Code!

```
class Student{
   int matrikel;
   String name;
   String studienrichtung;

   Student( ){
      matrikel = 0;
      vorname = “John Doe”;
      studienrichtung = “Ohne Angabe”;
   }

   Student(String name ){
      this( );
      this.name = name;
   }
}
```

Destruktoren

Wenn Objekte nicht mehr benötigt werden, können diese zerstört werden.

Ein Destruktor ist eine Methode, die unmittelbar vor der Zerstörung aufgerufen wird. Dazu dient die parameterlose Methode finalize( ).

In Java ist ein Destruktur weniger von Interesse, da Java über den **Garbage Collector** automatisch den für das Objekt reservierten Speicher wieder freigibt.

Es ist allerdings nicht garantiert, dass der Garbage Collector den Destruktor aufruft.

In bspw. C++ spielen Destruktoren eine viel wichtigere Rolle!

Kapitel 4.1.3 – Spezielle und typische Methoden: Get- und Setmethoden

Public / Private Attribute

Um direkte Manipulation von Attributen zu vermeiden, können Attribute mit *Modifikatoren* versehen werden.

Besitzt ein Attribut den Modifikator **public**, kann dieses Attribut frei geändert werden.

Ist der Modifikator **private**, kann das Attribut nur durch Objektmethoden geändert werden.

In der Regel sollten alle Attribute auf private stehen. (Geheimnisprinzip!)

Get- und Set-Methoden

Um Informationen zu einem Objekt zu bekommen, müssen Methoden bereitgestellt werden. Nur die darüber bereitgestellten Informationen sind abrufbar.

Für Attribute, die abrufbar und / oder schreibbar sein sollen, werden Get- und Set-Methoden definiert.

Get-Methoden definieren wir mit

public *Attributtyp* get*Attributbezeichner*( ){…}

Set-Methoden definieren wir mit

public void set*Attributbezeichner(Attributtyp Bezeichner){…}*

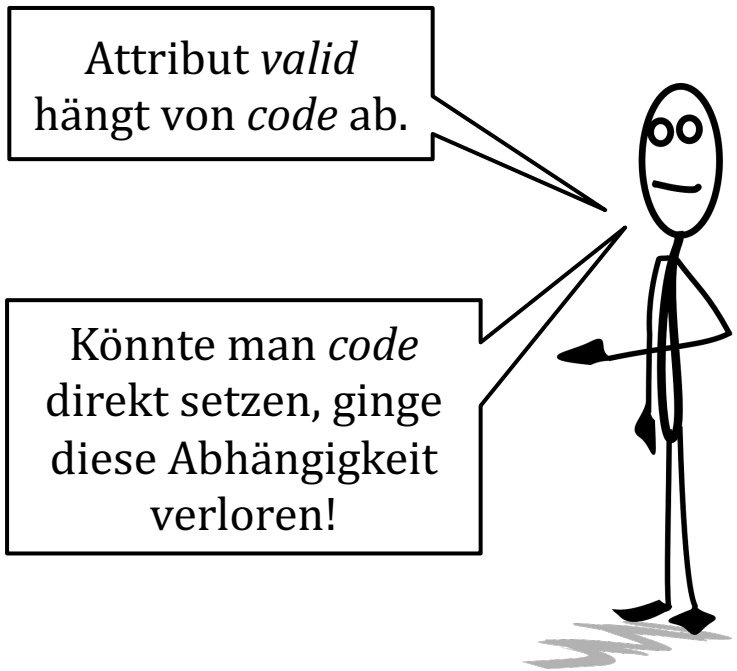
Public Attribute?

Public Attribute? Vermeiden!

```
class Secure{
   private boolean valid;
   private int code;

   public int getCode( ){
      return code;
   }

   public void setCode(int code){
      this.code = code;
      valid = magicFunction(code);
   }
}
```



Kapitel 4.1.4 – Classfiles, Packages und Imports

Wohin mit den Klassen?

Eine Klasse pro Quelldatei!

Generell können beliebig viele Klassen in einer Datei definiert werden. Darunter aber nur eine mit dem Modifizierer public.

Das erzeugt allerdings einige Probleme in Bezug auf Zugriff auf Klassen und / oder kompilieren der Dateien.

Daher folgende Richtlinie beachten:

Jede Klasse soll in einer eigenen Datei stehen.

Eine Klasse mit dem Bezeichner *Name* ist dann in der Datei *Name*.java definiert.

Zugriff auf Klassen

Damit Klassen genutzt werden können, müssen diese **importiert** werden.

Dazu nutzt man das Schlüsselwort import vor der Klassendefinition in der Form
import *Package.Classname*;

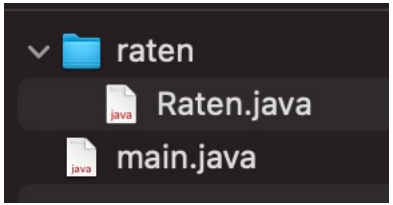
Dabei ist *Package* das **Paket**, in welchem die Klasse liegt, und *Classname* der Name der einzubindenden Klasse. Beispiele

```
import java.util.Arrays;
import java.util.Scanner;
```

Möchte man alle Klassen eines Pakets importieren, kann das Symbol * benutzt werden.

Erstellen von Packages

Jede Klasse kann einem Paket mit dem Schlüsselword package zugewiesen werden. Der Name des Pakets folgt dabei der Ordnerstruktur.



In Raten.java

```
package raten;

public class Raten{
    ...
}
```

In main.java

```
import raten.Raten;

public class main{
    public static void main(String[] args){
        ...
    }
}
```

Nächste Woche