Script generated by TTT

Title: groh: profile1 (26.06.2015)

Date: Fri Jun 26 09:37:25 CEST 2015

Duration: 69:15 min

Pages: 52

Konstruktoren (Constructors)

- können vollständige und konsistente Initialisierung der Objekte sicherstellen
- (auch generell für Methoden): Mehrere Varianten anbieten --> verschiedene Grade an Details für verschiedene Nutzer der Klasse (<---> Abstraktion (API), Information Hiding)
- Unterklassen-Konstruktoren: Zugriff auf Oberklassen-Konstruktor mit **super** und Erweiterung nach Bedarf

```
class Bicycle {
  int cadence;
  int speed;
  int gear;

  Bicycle(int c, int s, int g) {
    cadence = c;
    speed = s;
    gear = g;
  }

  Bicycle(int g) {
    cadence = 0;
    speed = 0;
    gear = g;
  }
}
```

```
class Tandem extends Bicycle {
  int numberOfDrivers;

  Tandem(int c, int s, int g, int n) {
    super(c, s, g);
    numberOfDrivers = n;
  }
}
```


Konstruktoren - Beispiel

```
class Person {
  String firstName;
  String lastName;
  long   taxIdent;                              // Requirement: taxIdent must be unique!

  Person(String fName, String lName, long tIdent) { // first constructor
    firstName = fName;
    lastName = lName;

    // Use the given tax identifier `tIdent` only if we can make sure it is unique:
    if (isUniqueTaxIdentifier(tIdent)) {
      taxIdent = tIdent;
    } else {
      System.err.println("Not unique!");
    }
  }

  Person(String fName, String lName) {  // second constructor
    firstName = fName;
    lastName = lName;

    // A unique tax identifier is created as a side-effect of this constructor:
    taxIdent = createUniqueTaxIdentifier();
  }
}
```

(gilt auch für andere Methoden): Welcher Konstruktor aufgerufen wird, hängt von den Aufruf-Parametern ab (<-->"Signatur")

```
// Complete and consistent and convenient ☺
Person p1 = new Person("Max", "Mustermann", 12345);     // first constructor is called
Person p2 = new Person("Fabienne", "Fabelhaft");        // second constructor is called
```


Methodenaufruf und Parameter-Übergabe

Übergabe einer Liste von Parametern an Methoden / Konstruktoren

```
int doSomething(int primitiveParameter1,
                SomeClass referenceParameter)
{
       int someInt = 17 + 9;
       primitiveParameter1 = 0;
       referenceParameter = null;
       return someInt;
}
```

Methodenaufruf und Parameter-Übergabe

Übergabe einer Liste von Parametern an Methoden / Konstruktoren

```
int doSomething(int primitiveParameter1,
                SomeClass referenceParameter)
{
       int someInt = 17 + 9;
       primitiveParameter1 = 0;
       referenceParameter = null;
       return someInt;
}
```

Übergabe von Parametern primitiven Typs: durch Kopieren des Werts (call by value)

```
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, someObject);
// after this statement, x still has value 1.
```


Methodenaufruf und Parameter-Übergabe

Übergabe einer Liste von Parametern an Methoden / Konstruktoren

```
int doSomething(int primitiveParameter1,
                SomeClass referenceParameter)
{
       int someInt = 17 + 9;
       primitiveParameter1 = 0;
       referenceParameter = null;
       return someInt;
}
```

Übergabe von Parametern primitiven Typs: durch Kopieren des Werts (call by value)

```
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, someObject);
// after this statement, x still has value 1.
```


Methodenaufruf und Parameter-Übergabe

Übergabe einer Liste von Parametern an Methoden / Konstruktoren

```
int doSomething(int primitiveParameter1,
                SomeClass referenceParameter)
{
       int someInt = 17 + 9;
       primitiveParameter1 = 0;
       referenceParameter = null;
       return someInt;
}
```

Übergabe von Parametern primitiven Typs: durch Kopieren des Werts (call by value)

```
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, someObject);
// after this statement, x still has value 1.
```


Methodenaufruf und Parameter-Übergabe

Übergabe einer Liste von Parametern an Methoden / Konstruktoren

```
int doSomething(int primitiveParameter1,
                SomeClass referenceParameter)
{
       int someInt = 17 + 9;
       primitiveParameter1 = 0;
       referenceParameter = null;
       return someInt;
}
```

Übergabe von Parametern von Referenztyp: AUCH durch call by value:

```
int x = 1;
SomeClass someObject = new SomeClass();
int y = doSomething(x, someObject);
// someObject still references the same object
```

Methodenaufruf und Parameter-Übergabe

Warum ist das so?

- *Erinnerung*: Variablen von Referenztyp zeigen auf ein Objekt dieses Typs (dieser Klasse) = Wert der Variablen ist diese Referenz
- Call by Value heisst: Kopien der Werte der Variablen werden übergeben. Diese Kopien können beliebig manipuliert werden. Sobald die Methode endet, werden die Kopien zerstört.
- Über die kopierten Referenzen auf Objekte, die ausserhalb der Methode weiterexistieren, können diese Objekte dennoch zugegriffen werden („.“-Operator) und bei Bedarf dauerhaft verändert werden. (Seiteneffekt)

Vereinfachtes Speicher-Modell

Zellnr (Adresse)	Zellname (Variablenname)	Zellinhalt
...		...
1149	**someObject**	<1150>
1150	someObject.xxx	23
1151	someObject.yyy	0
1152	someObject.zzz	7
...	...	...
5327	**referenceParameter**	<1150>
5328	...	...
5329	...	...

Methodenaufruf und Parameter-Übergabe

Warum ist das so?

- *Erinnerung*: Variablen von Referenztyp zeigen auf ein Objekt dieses Typs (dieser Klasse) = Wert der Variablen ist diese Referenz
- Call by Value heisst: Kopien der Werte der Variablen werden übergeben. Diese Kopien können beliebig manipuliert werden. Sobald die Methode endet, werden die Kopien zerstört.
- Über die kopierten Referenzen auf Objekte, die ausserhalb der Methode weiterexistieren, können diese Objekte dennoch zugegriffen werden („.“-Operator) und bei Bedarf dauerhaft verändert werden. (Seiteneffekt)

Vereinfachtes Speicher-Modell

Zellnr (Adresse)	Zellname (Variablenname)	Zellinhalt
...		...
1149	**someObject**	<1150>
1150	someObject.xxx	23
1151	someObject.yyy	0
1152	someObject.zzz	7
...	...	...
5327	**referenceParameter**	<1150>
5328	...	...
5329	...	...

Rückgabewerte von Methoden – `return`

- Methoden können Werte zurückgeben.
- Typ des Rückgabewertes muss in der Methodendefinition angegeben werden.
- In der Methode veranlasst `return` *expression*`;` die Rückgabe des Werts der *expression* und das Verlassen der Methode.
- Wenn Methode nichts zurück geben soll: Rückgabetyp ist `void`

```
long faculty(int n) {
  long result = 1;
  for (int i = 2; i <= n; i++) {
    result = result * i;
  }
  return result;
}
```

```
// Somewhere else...
long x = faculty(5);
System.out.println("Faculty of 5 is " + x + ".");
```


Rückgabewerte von Methoden – `return`

- Methoden können Werte zurückgeben.
- Typ des Rückgabewertes muss in der Methodendefinition angegeben werden.
- In der Methode veranlasst `return` *expression*`;` die Rückgabe des Werts der *expression* und das Verlassen der Methode.
- Wenn Methode nichts zurück geben soll: Rückgabetyp ist `void`

```
long faculty(int n) {
  long result = 1;
  for (int i = 2; i <= n; i++) {
    result = result * i;
  }
  return result;
}
```

```
// Somewhere else...
long x = faculty(5);
System.out.println("Faculty of 5 is " + x + ".");
```

Rückgabewerte von Methoden – `return`

- Wenn Rückgabe Referenz auf Objekt (oder Array), das in Methode erzeugt wurde --> Objekt/Array existiert natürlich auch nach Beendigung der Methode weiter.

```
Bicycle goGetABike() {
  Bicycle someBike;
  if (checkForSufficientFunds()) {
    someBike = new Bicycle();
    return someBike;
  } else {
    return null;
  }
}


// Call the method from somewhere else...
Bicycle bike = goGetABike();
...
```


Aufrufen von Methoden; Referenz `this`

- Methoden können über den dot-"."-Operator mit Bezug auf die jeweiligen Objekte „auf diesen Objekten" (←→ Bezug auf Attribute des Objekts (←→ Sichtbarkeit)) aufgerufen werden.
- Schreibersparnis: Code der Methoden kann andere Methoden und Attribute des Objekts ohne Extra-Angabe des Bezugs aufrufen / referenzieren. Wenn doch nötig, können Objekte mit der Referenz `this` Bezug auf sich selbst nehmen.

```
public class Bicycle {
  public int cadence = 0;

  public void addToCadence(int amount) {
    cadence = cadence + amount  // also: this.cadence = this.cadence + amount;
  }

  public void someOtherMethod() {
    addToCadence(5);            // also: this.addToCadence(5)
  }
}
```

```
Bicycle bike1 = new Bicycle();
Bicycle bike2 = new Bicycle();

bike1.addToCadence(10);   // bike1.cadence == 10;
bike2.addToCadence(9);    // bike2.cadence == 9;
bike1.someOtherMethod();  // bike1.cadence == 15;
```


Aufrufen von Methoden; Referenz `this`

- Methoden können über den dot-"."-Operator mit Bezug auf die jeweiligen Objekte „auf diesen Objekten" (←→ Bezug auf Attribute des Objekts (←→ Sichtbarkeit)) aufgerufen werden.
- Schreibersparnis: Code der Methoden kann andere Methoden und Attribute des Objekts ohne Extra-Angabe des Bezugs aufrufen / referenzieren. Wenn doch nötig, können Objekte mit der Referenz `this` Bezug auf sich selbst nehmen.

```
public class Bicycle {
  public int cadence = 0;

  public void addToCadence(int amount) {
    cadence = cadence + amount  // also: this.cadence = this.cadence + amount;
  }

  public void someOtherMethod() {
    addToCadence(5);            // also: this.addToCadence(5)
  }
}
```

```
Bicycle bike1 = new Bicycle();
Bicycle bike2 = new Bicycle();

bike1.addToCadence(10);   // bike1.cadence == 10;
bike2.addToCadence(9);    // bike2.cadence == 9;
bike1.someOtherMethod();  // bike1.cadence == 15;
```

115

Access Modifiers, Packages

- Access Modifier für Methoden und Attribute:

	Class	Package	Subclasses	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
no modifier	✓	✓		
`private`	✓			

- Packages:
  - Kapseln (hierarchisch organisiert) Mengen von Klassen und Interfaces
  - Deklaration: `package` *nameOfPackage*;
  - Beispiele: `java.math, java.lang, java.net, de.tum.in`

Access Modifiers, Packages

- Access Modifier für Methoden und Attribute:

	Class	Package	Subclasses	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
no modifier	✓	✓		
`private`	✓			

- Packages:
  - Kapseln (hierarchisch organisiert) Mengen von Klassen und Interfaces
  - Deklaration: `package` *nameOfPackage*;
  - Beispiele: `java.math, java.lang, java.net, de.tum.in`

Access Modifiers, Packages

- Access Modifier für Methoden und Attribute:

	Class	Package	Subclasses	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
no modifier	✓	✓		
`private`	✓			

- Packages:
  - Kapseln (hierarchisch organisiert) Mengen von Klassen und Interfaces
  - Deklaration: `package` *nameOfPackage*;
  - Beispiele: `java.math, java.lang, java.net, de.tum.in`

Access Modifiers, Packages

- Access Modifier für Methoden und Attribute:

	Class	Package	Subclasses	World
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
no modifier	✓	✓		
`private`	✓			

- Packages:
  - Kapseln (hierarchisch organisiert) Mengen von Klassen und Interfaces
  - Deklaration: `package` *nameOfPackage*;
  - Beispiele: `java.math, java.lang, java.net, de.tum.in`

Access Modifiers: `static` und `final`

- **`static`:**
  - *Methode* oder *Attribut* gehört zur Klasse und nicht zum Objekt
    (*Attribut*: existiert nur einmal, ist für alle Objekte dasselbe)
    „Klassenmethode, Klassenattribut“ <---> „Instanzenmethode, Instanzenattribut“
- **`final`:**
  - für *Attribute*: können nicht mehr geändert werden (Konstanten)
  - für *Methoden*: können nicht overridden oder hidden werden (kommt gleich)
  - für *Klassen*: Es können keine Unterklassen davon abgeleitet werden.

```
final class MyClass {
        static int sameForAllInstances = 3;
        final int constantMayBeDifferentForEachInstance;
        static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;

        static void  methodOne() { /* ... */ }
        final void  methodTwo() { /* ... */ }
        static final void methodThree() { /* ... */ }
        void methodFour() { /* ... */ }
    …
}
```


Access Modifiers: `static` und `final`

- **`static`:**
  - *Methode* oder *Attribut* gehört zur Klasse und nicht zum Objekt
    (*Attribut*: existiert nur einmal, ist für alle Objekte dasselbe)
    „Klassenmethode, Klassenattribut“ <---> „Instanzenmethode, Instanzenattribut“
- **`final`:**
  - für *Attribute*: können nicht mehr geändert werden (Konstanten)
  - für *Methoden*: können nicht overridden oder hidden werden (kommt gleich)
  - für *Klassen*: Es können keine Unterklassen davon abgeleitet werden.

```
final class MyClass {
        static int sameForAllInstances = 3;
        final int constantMayBeDifferentForEachInstance;
        static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;

        static void  methodOne() { /* ... */ }
        final void  methodTwo() { /* ... */ }
        static final void methodThree() { /* ... */ }
        void methodFour() { /* ... */ }
    …
}
```

117

Access Modifiers: `static` und `final`

- **`static`:**
  - *Methode* oder *Attribut* gehört zur Klasse und nicht zum Objekt
    (*Attribut*: existiert nur einmal, ist für alle Objekte dasselbe)
    „Klassenmethode, Klassenattribut“ <---> „Instanzenmethode, Instanzenattribut“
- **`final`:**
  - für *Attribute*: können nicht mehr geändert werden (Konstanten)
  - für *Methoden*: können nicht overridden oder hidden werden (kommt gleich)
  - für *Klassen*: Es können keine Unterklassen davon abgeleitet werden.

```
final class MyClass {
        static int sameForAllInstances = 3;
        final int constantMayBeDifferentForEachInstance;
        static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;

        static void  methodOne() { /* ... */ }
        final void  methodTwo() { /* ... */ }
        static final void methodThree() { /* ... */ }
        void methodFour() { /* ... */ }
    …
}
```

117

Access Modifiers: `static` und `final`

- **`static`:**
  - *Methode* oder *Attribut* gehört zur Klasse und nicht zum Objekt
    (*Attribut*: existiert nur einmal, ist für alle Objekte dasselbe)
    „Klassenmethode, Klassenattribut“ <---> „Instanzenmethode, Instanzenattribut“
- **`final`:**
  - für *Attribute*: können nicht mehr geändert werden (Konstanten)
  - für *Methoden*: können nicht overridden oder hidden werden (kommt gleich)
  - für *Klassen*: Es können keine Unterklassen davon abgeleitet werden.

```
final class MyClass {
        static int sameForAllInstances = 3;
        final int constantMayBeDifferentForEachInstance;
        static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;

        static void  methodOne() { /* ... */ }
        final void  methodTwo() { /* ... */ }
        static final void methodThree() { /* ... */ }
        void methodFour() { /* ... */ }
    …
}
```

117

Access Modifiers: static und final

```
public class MyClass {
        static int sameForAllInstances = 3;
        final int constantMayBeDifferentForEachInstance;
        static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;
        public int instVar;

        public MyClass(int iv, int const) {
                instVar = iv;
                constantMayBeDifferentForEachInstance = const;
        } // constructor

        public static void methodOne() { /* ... */ }
        public void methodTwo() { /* ... */ }
}
```

Muss im Constructor gesetzt werden (oder mit Initializer direkt bei Deklaration (bspw. `final int c = 0;`))

```
MyClass m1 = new MyClass(20, 11);
MyClass m2 = new MyClass(30, 9);
MyClass.CONSTANT_SAME_FOR_ALL_INSTANCES = 3333;   //ERROR
MyClass.sameForAllInstances = 99;
System.out.println(m1.sameForAllInstances);              // 99
System.out.println(m2.sameForAllInstances);              // 99
System.out.println(MyClass.sameForAllInstances);         // 99
System.out.println(m1.instVar);                          // 20
System.out.println(m2.instVar);                          // 30
m1.instVar = 77;
MyClass.methodOne();
m1.methodTwo();
MyClass.methodTwo();      //ERROR
```


Access Modifiers: static und final

public class MyClass {
static int sameForAllInstances = 3;
final int constantMayBeDifferentForEachInstance;
static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;
public int instVar;
public MyClass(int iv, int const) {
instVar = iv;
constantMayBeDifferentForEachInstance = const;
} // constructor
public static void methodOne() { /* ... */ }
public void methodTwo() { /* ... */ }
}
Muss im Constructor gesetzt werden (oder mit Initializer direkt bei Deklaration (bspw. final int c = 0;))
MyClass m1 = new MyClass(20, 11);
MyClass m2 = new MyClass(30, 9);
MyClass.CONSTANT_SAME_FOR_ALL_INSTANCES = 3333; //ERROR
MyClass.sameForAllInstances = 99;
System.out.println(m1.sameForAllInstances); // 99
System.out.println(m2.sameForAllInstances); // 99
System.out.println(MyClass.sameForAllInstances); // 99
System.out.println(m1.instVar); // 20
System.out.println(m2.instVar); // 30
m1.instVar = 77;
MyClass.methodOne();
m1.methodTwo();
MyClass.methodTwo(); //ERROR
118

Access Modifiers: static und final

public class MyClass {
static int sameForAllInstances = 3;
final int constantMayBeDifferentForEachInstance;
static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;
public int instVar;
public MyClass(int iv, int const) {
instVar = iv;
constantMayBeDifferentForEachInstance = const;
} // constructor
public static void methodOne() { /* ... */ }
public void methodTwo() { /* ... */ }
}
Muss im Constructor gesetzt werden (oder mit Initializer direkt bei Deklaration (bspw. final int c = 0;))
MyClass m1 = new MyClass(20, 11);
MyClass m2 = new MyClass(30, 9);
MyClass.CONSTANT_SAME_FOR_ALL_INSTANCES = 3333; //ERROR
MyClass.sameForAllInstances = 99;
System.out.println(m1.sameForAllInstances); // 99
System.out.println(m2.sameForAllInstances); // 99
System.out.println(MyClass.sameForAllInstances); // 99
System.out.println(m1.instVar); // 20
System.out.println(m2.instVar); // 30
m1.instVar = 77;
MyClass.methodOne();
m1.methodTwo();
MyClass.methodTwo(); //ERROR
118

Access Modifiers: static und final

public class MyClass {
static int sameForAllInstances = 3;
final int constantMayBeDifferentForEachInstance;
static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;
public int instVar;
public MyClass(int iv, int const) {
instVar = iv;
constantMayBeDifferentForEachInstance = const;
} // constructor
public static void methodOne() { /* ... */ }
public void methodTwo() { /* ... */ }
}
Muss im Constructor gesetzt werden (oder mit Initializer direkt bei Deklaration (bspw. final int c = 0;))
MyClass m1 = new MyClass(20, 11);
MyClass m2 = new MyClass(30, 9);
MyClass.CONSTANT_SAME_FOR_ALL_INSTANCES = 3333; //ERROR
MyClass.sameForAllInstances = 99;
System.out.println(m1.sameForAllInstances); // 99
System.out.println(m2.sameForAllInstances); // 99
System.out.println(MyClass.sameForAllInstances); // 99
System.out.println(m1.instVar); // 20
System.out.println(m2.instVar); // 30
m1.instVar = 77;
MyClass.methodOne();
m1.methodTwo();
MyClass.methodTwo(); //ERROR
118

Access Modifiers: static und final

```
public class MyClass {
        static int sameForAllInstances = 3;
        final int constantMayBeDifferentForEachInstance;
        static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;
        public int instVar;

        public MyClass(int iv, int const) {
                instVar = iv;
                constantMayBeDifferentForEachInstance = const;
        } // constructor

        public static void methodOne() { /* ... */ }
        public void methodTwo() { /* ... */ }
}
```

Muss im Constructor gesetzt werden (oder mit Initializer direkt bei Deklaration (bspw. `final int c = 0;`))

```
MyClass m1 = new MyClass(20, 11);
MyClass m2 = new MyClass(30, 9);
MyClass.CONSTANT_SAME_FOR_ALL_INSTANCES = 3333;   //ERROR
MyClass.sameForAllInstances = 99;
System.out.println(m1.sameForAllInstances);            // 99
System.out.println(m2.sameForAllInstances);            // 99
System.out.println(MyClass.sameForAllInstances);       // 99
System.out.println(m1.instVar);                    // 20
System.out.println(m2.instVar);                    // 30
m1.instVar = 77;
MyClass.methodOne();
m1.methodTwo();
MyClass.methodTwo();     //ERROR
```


Access Modifiers: static und final

```
public class MyClass {
        static int sameForAllInstances = 3;
        final int constantMayBeDifferentForEachInstance;
        static final int CONSTANT_SAME_FOR_ALL_INSTANCES = 7;
        public int instVar;

        public MyClass(int iv, int const) {
                instVar = iv;
                constantMayBeDifferentForEachInstance = const;
        } // constructor

        public static void methodOne() { /* ... */ }
        public void methodTwo() { /* ... */ }
}
```

Muss im Constructor gesetzt werden (oder mit Initializer direkt bei Deklaration (bspw. `final int c = 0;`))

```
MyClass m1 = new MyClass(20, 11);
MyClass m2 = new MyClass(30, 9);
MyClass.CONSTANT_SAME_FOR_ALL_INSTANCES = 3333;   //ERROR
MyClass.sameForAllInstances = 99;
System.out.println(m1.sameForAllInstances);            // 99
System.out.println(m2.sameForAllInstances);            // 99
System.out.println(MyClass.sameForAllInstances);       // 99
System.out.println(m1.instVar);                    // 20
System.out.println(m2.instVar);                    // 30
m1.instVar = 77;
MyClass.methodOne();
m1.methodTwo();
MyClass.methodTwo();     //ERROR
```


Overloading

Overloading: In einer Klasse mehrere Methoden mit gleichem Namen, aber verschiedener Parameterliste:

```
class OverloadingDemoClass {
      public int doSomething() {
            return 1 + 1;
      }

      public int doSomething(int param) {
            return param + 2;
      }
}
```

```
OverloadingDemoClass odc = new OverloadingDemoClass();
int result1 = odc.doSomething();
int result2 = odc.doSomething(33);
```

Sinn: Flexibilität (speziellere und weniger spezielle Varianten der Methode anbieten, Abstraktion (--> APIs):

Overriding

Overriding: In einer Unterklasse Methode mit gleichem Namen, und gleicher Parameterliste wie in Oberklasse:

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                          // mb.speed == 20
```

⇨ Ausgabe: **`subclass instance-method`**

Sinn: Unterklasse bietet speziellere Version der Methode an (Aspekt von Polymorphie)

Overriding

Overriding: In einer Unterklasse Methode mit gleichem Namen, und gleicher Parameterliste wie in Oberklasse:

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                            // mb.speed == 20
```

⇨ Ausgabe: **subclass instance-method**

Sinn: Unterklasse bietet speziellere Version der Methode an (Aspekt von Polymorphie)

Overriding

Overriding: In einer Unterklasse Methode mit gleichem Namen, und gleicher Parameterliste wie in Oberklasse:

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                            // mb.speed == 20
```

⇨ Ausgabe: **subclass instance-method**

Sinn: Unterklasse bietet speziellere Version der Methode an (Aspekt von Polymorphie)

121

Overriding

Overriding: In einer Unterklasse Methode mit gleichem Namen, und gleicher Parameterliste wie in Oberklasse:

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                            // mb.speed == 20
```

⇨ Ausgabe: **subclass instance-method**

Sinn: Unterklasse bietet speziellere Version der Methode an (Aspekt von Polymorphie)

121

Overriding

Overriding: In einer Unterklasse Methode mit gleichem Namen, und gleicher Parameterliste wie in Oberklasse:

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                            // mb.speed == 20
```

⇨ Ausgabe: **subclass instance-method**

Sinn: Unterklasse bietet speziellere Version der Methode an (Aspekt von Polymorphie)

121

Overriding

Overriding: In einer Unterklasse Methode mit gleichem Namen, und gleicher Parameterliste wie in Oberklasse:

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
```

Variante mit `super`

```
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        super(2 * increment); // call overridden method of superclass
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                          // mb.speed == 20
```

⇨ Ausgabe:

```
superclass instance-method
subclass instance-method
```

Sinn: Unterklasse bietet speziellere Version der Methode an (Aspekt von Polymorphie)

Hiding

Hiding: Analog zu Overriding aber für Klassenmethoden (`static`)

```
class Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("superclass class-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("subclass class-method");
    }
}
```

```
Bicycle.myClassMethod(10);          // "superclass class-method"
MountainBike.myClassMethod(10);     // "subclass class-method"
```


Hiding

Hiding: Analog zu Overriding aber für Klassenmethoden (`static`)

```
class Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("superclass class-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("subclass class-method");
    }
}
```

```
Bicycle.myClassMethod(10);          // "superclass class-method"
MountainBike.myClassMethod(10);     // "subclass class-method"
```


Hiding

Hiding: Analog zu Overriding aber für Klassenmethoden (`static`)

```
class Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("superclass class-method");
    }
}
```

```
class MountainBike extends Bicycle {
    public static void myClassMethod(int someInt) {
        System.out.println("subclass class-method");
    }
}
```

```
Bicycle.myClassMethod(10);          // "superclass class-method"
MountainBike.myClassMethod(10);     // "subclass class-method"
```

Polymorphie

- Essentielles Feature in objektorientierter Programmierung.
- Polymorphie: Subklassen-Objekte können Superklassen-Variablen zugewiesen werden:

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```


Polymorphie

- Essentielles Feature in objektorientierter Programmierung.
- Polymorphie: Subklassen-Objekte können Superklassen-Variablen zugewiesen werden:

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```


Polymorphie

- Essentielles Feature in objektorientierter Programmierung.
- Polymorphie: Subklassen-Objekte können Superklassen-Variablen zugewiesen werden:

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

- Nur die Methoden und Attribute, die auch in der SUPERklasse vorhanden sind, können auf der Instanz (dem Objekt) der SUBklasse über die Variable vom Typ der SUPERklasse zugegriffen / aufgerufen werden
- Overriding / Hiding: Es wird jeweils aber die speziellere, in der Subklasse definierte Methode auf dem Objekt aufgerufen

Polymorphie

- Essentielles Feature in objektorientierter Programmierung.
- Polymorphie: Subklassen-Objekte können Superklassen-Variablen zugewiesen werden:

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

- Nur die Methoden und Attribute, die auch in der SUPERklasse vorhanden sind, können auf der Instanz (dem Objekt) der SUBklasse über die Variable vom Typ der SUPERklasse zugegriffen / aufgerufen werden
- Overriding / Hiding: Es wird jeweils aber die speziellere, in der Subklasse definierte Methode auf dem Objekt aufgerufen

```
bicycle.gear = 3;            // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;     // ERROR! seatHeight is not a field in class Bicycle
mountainBike.seatHeight = 20; // Ok

mountainBike.speedUp(5);     // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);         // Overridden method in subclass MountainBike is used
```

Polymorphie

- Essentielles Feature in objektorientierter Programmierung.
- Polymorphie: Subklassen-Objekte können Superklassen-Variablen zugewiesen werden:

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

- Nur die Methoden und Attribute, die auch in der SUPERklasse vorhanden sind, können auf der Instanz (dem Objekt) der SUBklasse über die Variable vom Typ der SUPERklasse zugegriffen / aufgerufen werden
- Overriding / Hiding: Es wird jeweils aber die speziellere, in der Subklasse definierte Methode auf dem Objekt aufgerufen

```
bicycle.gear = 3;            // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;     // ERROR! seatHeight is not a field in class Bicycle
mountainBike.seatHeight = 20; // Ok

mountainBike.speedUp(5);     // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);         // Overridden method in subclass MountainBike is used
```


Polymorphie

- Essentielles Feature in objektorientierter Programmierung.
- Polymorphie: Subklassen-Objekte können Superklassen-Variablen zugewiesen werden:

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

- Nur die Methoden und Attribute, die auch in der SUPERklasse vorhanden sind, können auf der Instanz (dem Objekt) der SUBklasse über die Variable vom Typ der SUPERklasse zugegriffen / aufgerufen werden
- Overriding / Hiding: Es wird jeweils aber die speziellere, in der Subklasse definierte Methode auf dem Objekt aufgerufen

```
bicycle.gear = 3;            // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;     // ERROR! seatHeight is not a field in class Bicycle
mountainBike.seatHeight = 20; // Ok

mountainBike.speedUp(5);     // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);         // Overridden method in subclass MountainBike is used
```

126

Overriding

Overriding: In einer Unterklasse Methode mit gleichem Namen, und gleicher Parameterliste wie in Oberklasse:

```
class Bicycle {
    int speed;
    public void speedUp(int increment) {
        speed = speed + increment;
        System.out.println("superclass instance-method");
    }
}
```

Variante mit `super`

```
class MountainBike extends Bicycle {
    public void speedUp(int increment) {
        super(2 * increment); // call overridden method of superclass
        System.out.println("subclass instance-method");
    }
}
```

```
MountainBike mb = new MountainBike();
mb.speedUp(10);                          // mb.speed == 20
```

⇒ Ausgabe:

```
superclass instance-method
subclass instance-method
```

Sinn: Unterklasse bietet speziellere Version der Methode an (Aspekt von Polymorphie)

Polymorphie

- Essentielles Feature in objektorientierter Programmierung.
- Polymorphie: Subklassen-Objekte können Superklassen-Variablen zugewiesen werden:

```
MountainBike mountainBike = new MountainBike();
Bicycle bicycle = mountainBike;
```

- Nur die Methoden und Attribute, die auch in der SUPERklasse vorhanden sind, können auf der Instanz (dem Objekt) der SUBklasse über die Variable vom Typ der SUPERklasse zugegriffen / aufgerufen werden
- Overriding / Hiding: Es wird jeweils aber die speziellere, in der Subklasse definierte Methode auf dem Objekt aufgerufen

```
bicycle.gear = 3;            // Ok, gear defined in class Bicycle
bicycle.seatHeight = 20;     // ERROR! seatHeight is not a field in class Bicycle
mountainBike.seatHeight = 20; // Ok

mountainBike.speedUp(5);     // Overridden method in subclass MountainBike is used
bicycle.speedUp(10);         // Overridden method in subclass MountainBike is used
```

126

Polymorphie, Interfaces <---> Polymorphie

- Sinn und Zweck von **Polymorphie**: Allgemeinere(s) Superklassen-Verhalten und -Zustände sind garantiert, d.h. können stets benutzt werden (--> gutes Software-Design).

  *„Ein MountainBike **ist** ein Bicycle“*

- Ähnlich: **Interfaces** „garantieren“ ebenfalls einen Satz von Methoden und Attributen --> diese garantierten Methoden und Attribute können von Objekten aller derjenigen Klassen, die das Interface implementieren, auch genutzt werden.

Polymorphie, Interfaces <---> Polymorphie

- Sinn und Zweck von **Polymorphie**: Allgemeinere(s) Superklassen-Verhalten und -Zustände sind garantiert, d.h. können stets benutzt werden (--> gutes Software-Design).

  *„Ein MountainBike **ist** ein Bicycle“*

- Ähnlich: **Interfaces** „garantieren“ ebenfalls einen Satz von Methoden und Attributen --> diese garantierten Methoden und Attribute können von Objekten aller derjenigen Klassen, die das Interface implementieren, auch genutzt werden.

Interfaces <---> Polymorphie : Beispiel

```
interface SubOrderApocrita {
    public void sting();
}
```

```
class LittleBee implements SubOrderApocrita {
      public void sting() {
            System.out.println("*pieks*");
      }
}
```

```
class AngryHornet implements SubOrderApocrita {
      public void sting() {
            System.out.println("*MEGAPIEKS*");
      }
}
```

```
LittleBee maja = new LittleBee();
AngryHornet horst = new AngryHornet();
SubOrderApocrita someStinger;
someStinger = maja;
someStinger.sting();          // *pieks*
someStinger = horst;
someStinger.sting();          // *MEGAPIEKS*
```


Interfaces <---> Polymorphie : Beispiel

```
interface SubOrderApocrita {
      public void sting();
}
```

```
class LittleBee implements SubOrderApocrita {
      public void sting() {
            System.out.println("*pieks*");
      }
}
```

```
class AngryHornet implements SubOrderApocrita {
      public void sting() {
            System.out.println("*MEGAPIEKS*");
      }
}
```

```
LittleBee maja = new LittleBee();
AngryHornet horst = new AngryHornet();
SubOrderApocrita someStinger;
someStinger = maja;
someStinger.sting();          // *pieks*
someStinger = horst;
someStinger.sting();          // *MEGAPIEKS*
```

Interfaces <---> Polymorphie : Beispiel

```
interface SubOrderApocrita {
      public void sting();
}
```

```
class LittleBee implements SubOrderApocrita {
      public void sting() {
            System.out.println("*pieks*");
      }
}
```

```
class AngryHornet implements SubOrderApocrita {
      public void sting() {
            System.out.println("*MEGAPIEKS*");
      }
}
```

```
LittleBee maja = new LittleBee();
AngryHornet horst = new AngryHornet();
SubOrderApocrita someStinger;
someStinger = maja;
someStinger.sting();          // *pieks*
someStinger = horst;
someStinger.sting();          // *MEGAPIEKS*
```