10.5 Konstruktoren

- Eigentlich haben wir bereits die wichtigsten Aspekte der Klassenerweiterung geklärt.
- Im Folgenden gehen wir noch auf die Erzeugung der Objekte einer Unterklasse ein.
  - Dies stellt sich als technisch relativ schwierig heraus.

- Wichtige Beobachtungen
  - Konstruktoren der Oberklasse reichen alleine nicht für die Initialisierung eines Objekts der Unterklasse aus.
    - Neue Datenfelder in der Unterklasse können nicht initialisiert werden.
  - Konstruktoren der Unterklasse reichen alleine nicht für die Initialisierung der Datenfelder der Oberklasse aus.
    - Privat-deklarierte Datenfelder der Oberklasse können nicht zugegriffen werden.

Konstruktoren in Java

- Zur Erinnerung
  - Klassen können eine beliebige Anzahl von Konstruktoren (mit unterschiedlichsten Parametern) besitzen.
    - Genau wie bei überladenen Methoden müssen diese Konstruktoren sich in Ihrer Signatur unterscheiden.
  - Wird zu einer Klasse kein Konstruktor explizit definiert, so wird automatisch der parameterlose Konstruktor erzeugt.
  - Wird ein Konstruktor aufgerufen, so wird in der zugehörigen Klasse nach einer entsprechenden Implementierung gesucht. Wenn eine passende Implementierung nicht vorhanden ist, gibt es einen Übersetzungsfehler.

Besonderheit bei der Klassenerweiterung in Java

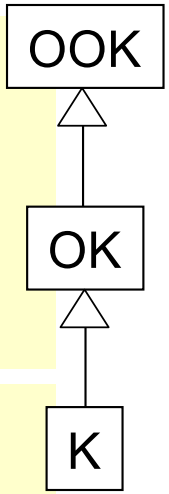
- Die Konstruktoren einer Klasse werden **nicht** vererbt.

Konstruktoren und Oberklassen

- Jeder Konstruktor einer Klasse **muss** (implizit oder explizit, direkt oder indirekt) einen Konstruktor der Oberklasse aufrufen!
  - *Expliziter und direkter Aufruf* erfolgt durch Angabe des Schlüsselworts ***super*** und den entsprechenden Parametern für den Konstruktor der Oberklasse.
    - Dieser Aufruf muss erster Befehl im Konstruktor der Unterklasse sein.
  - *Indirekter Aufruf* eines Konstruktors der Oberklasse erfolgt dann, wenn zunächst ein anderer Konstruktor der gleichen Klasse gerufen wird.
    - Dazu verwenden wir das Schlüsselwort ***this*** und die entsprechenden Parameter. Dieser Aufruf muss erster Befehl im Konstruktor sein.
    - Ein gegenseitiges (bzw. zyklisches) Aufrufen der Konstruktoren innerhalb einer Klasse ist verboten!
  - In allen anderen Fällen (erster Befehl ist nicht ein Konstruktoraufruf mit this und super) erfolgt ein impliziter Aufruf des parameterlosen Konstruktors, der dann in der Oberklasse verfügbar sein muss.

Beispiel 1 (impliziter Konstruktoraufruf)

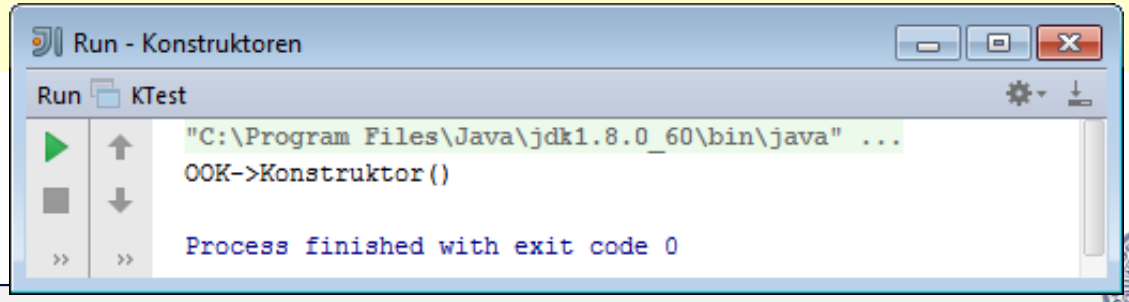
```
class OOK{
   OOK() {
        System.out.println("OOK->Konstruktor()");
   }
   OOK(int i1) { System.out.println("OOK->Konstruktor(int)"); }
}
```




```
class OK extends OOK{
}
```

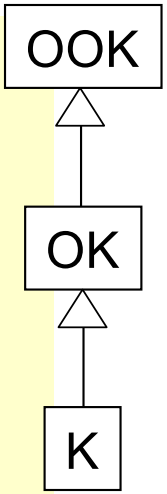
```
class K extends OK{
}
```

```
public class KTest {
   public static void main(String[] args) {
      K k = new K();
   }
}
```

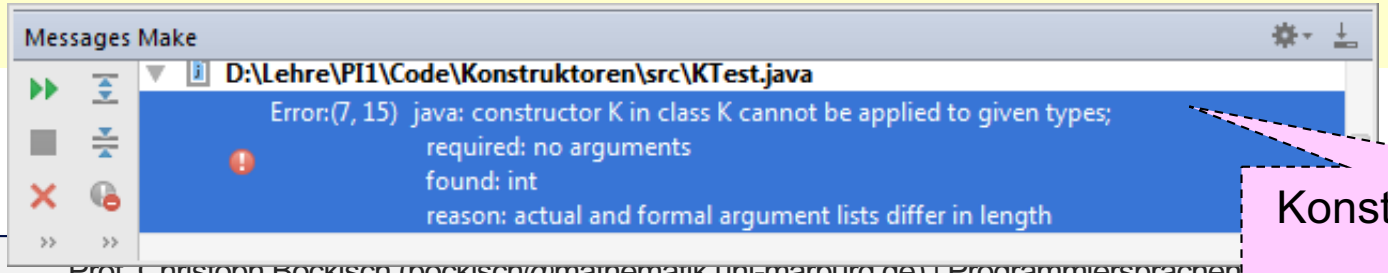


Beispiel 2 (keine Vererbung bei Konstruktoren)

```
class OOK{
   OOK() {
        System.out.println("OOK->Konstruktor()");
   }
   OOK(int i1) {
         System.out.println("OOK->Konstruktor(int)");
   }
}
class OK extends OOK{
}
class K extends OK{
}
```



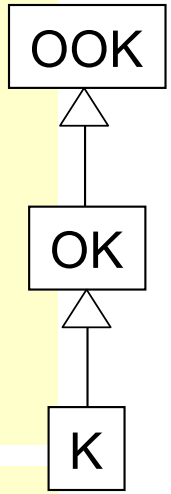

```
public class KTest {
   public static void main(String[] args) {
      K k = new K(5);
   }
}
```



Konstruktoren werden eben nicht vererbt!

Beispiel 3 (Reihenfolge der Konstruktoraufrufe)

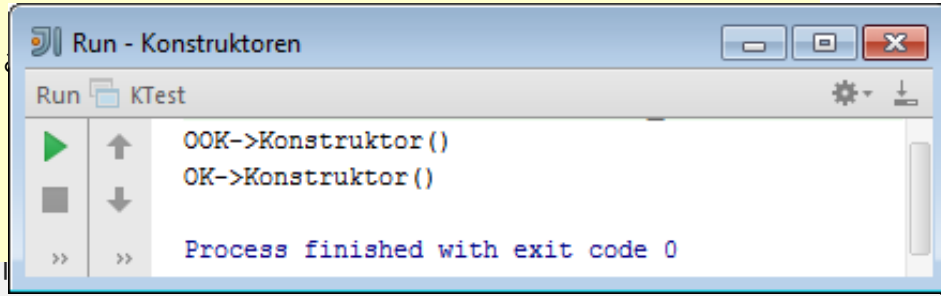
```
class OOK{
    OOK() {
        System.out.println("OOK->Konstruktor()");
    }
    OOK(int i1) {
        System.out.println("OOK->Konstruktor(int)");
    }
}
```




```
class OK extends OOK {
    OK() {
        System.out.println("OK->Konstruktor()");
    }
}
```

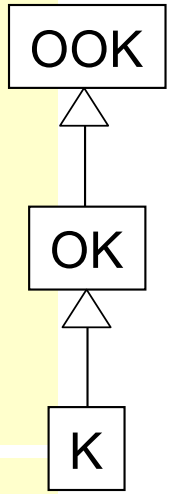
```
class K extends OK {
}
```

```
public class KTest {
    public static void main(String[] a
        K k = new K();
    }
}
```



Beispiel 4 (Kein parameterloser Konstruktor)

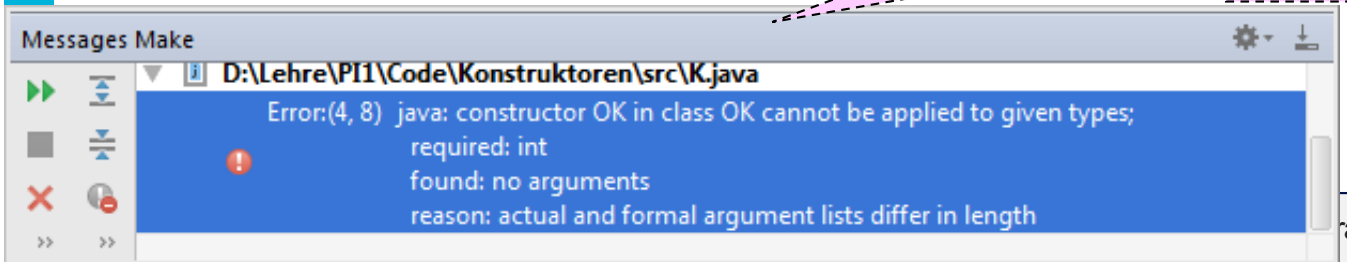
```
class OOK{
    OOK() {
        System.out.println("OOK->Konstruktor()");
    }
    OOK(int i1) {
        System.out.println("OOK->Konstruktor(int)");
    }
}
```



```
class OK extends OOK {
    OK(int x) {
        System.out.println("OK->Konstruktor(int)");
    }
}

class K extends OK {
}
```

Die Klasse OK hat nicht mehr automatisch einen parameterlosen Konstruktor!
Dieser wird aber von dem automatisch erzeugten Konstruktor für Klasse K aufgerufen!



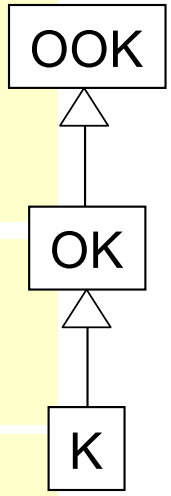
Beispiel 5 (Konstruktor mit Parameter)

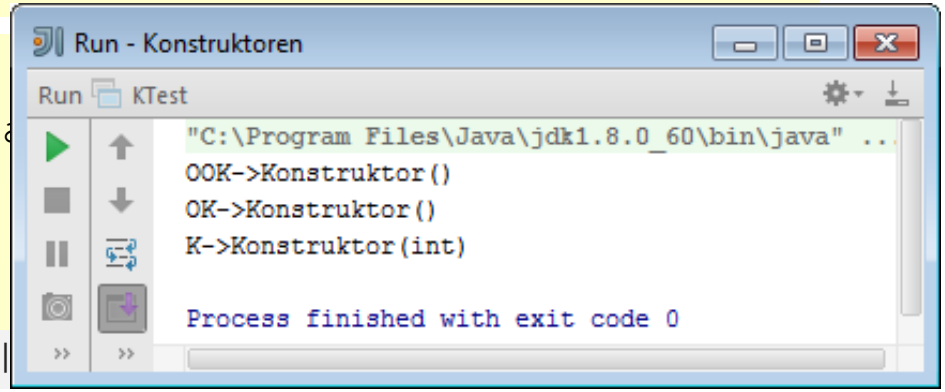
```
class OOK{
   OOK() { /* siehe oben */ }
   OOK(int i1) { /* siehe oben */ }
}
```

```
class OK extends OOK {
   OK() { System.out.println("OK->Konstruktor()"); }
}
```

```
class K extends OK {
   K() {
        System.out.println("K->Konstruktor()");
   }
   K(int x) {
        System.out.println("K->Konstruktor(int)");
   }
}
```

```
public class KTest {
   public static void main(String[] a
      K k = new K(5);
   }
}
```

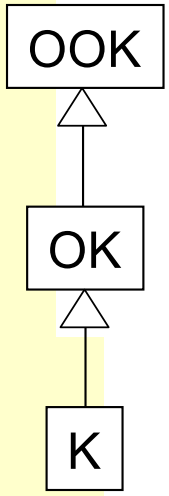


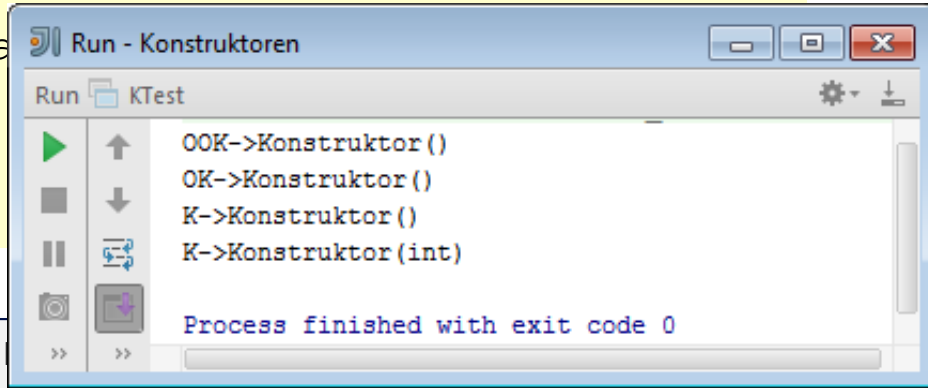


Beispiel 6 (Aufruf eines Konstruktors mit this)

```
class OOK{
    /* siehe oben */
}
class OK extends OOK {
   OK() { /* siehe oben */   }
}
class K extends OK {
   K() {
        System.out.println("K->Konstruktor()");
   }
   K(int x) {
        this();
        System.out.println("K->Konstruktor(int)");
   }
}
public class KTest {
   public static void main(String[] a
      K k = new K(5);
   }
}
```

Weil this() aufgerufen wird, wird für **diesen** Konstruktor nicht mehr der parameterlose Konstruktor der Oberklasse aufgerufen!



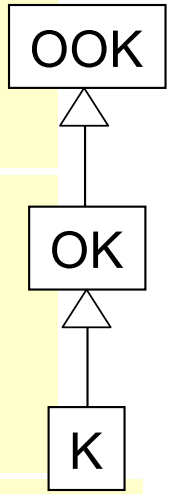


Beispiel 7 (Aufruf eines Konstruktors mit super)

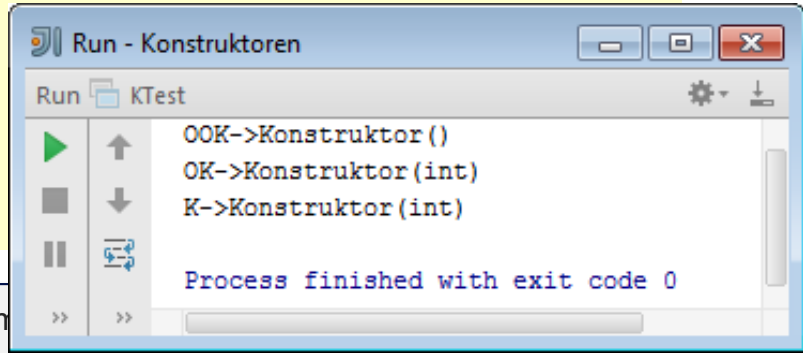
```
class OOK{
   /* siehe oben */
}
```

```
class OK extends OOK {
   OK(int x) {
        System.out.println("OK->Konstruktor(int)");
   }
}
```

```
class K extends OK {
   K() { /* siehe oben */   }
   K(int x) {
        super(x);
        System.out.println("K->Konstruktor(int)");
   }
}
public class KTest {
   public static void main(String[] args)
      K k = new K(5);
   }
}
```



Jetzt wird mit super der Konstruktor der Oberklasse gerufen.

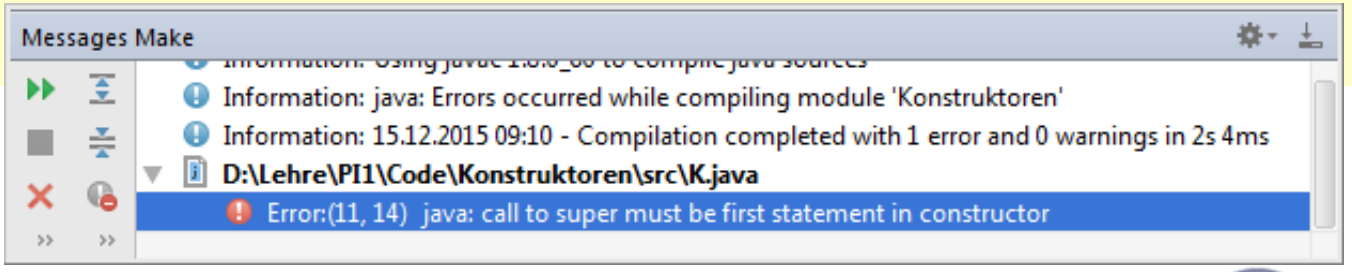


Beispiel 8 (Position des super-Aufrufs)

```
class OOK{
/* siehe oben */
}
```

```
class OK extends OOK {
   OK() {/* siehe oben */ }
   OK(int i) {/* siehe oben */}
}
```

```
class K extends OK {
   K() {
        System.out.println("K->Konstruktor()");
   }
   K(int x) {
        System.out.println("K->Konstruktor(int)");
        super(x);
   }
}
```



Beispiel für die Klassen Polygon und Trajectory

```
class Polygon extends Trajectory {
    int count; // Abspeicherung der Anzahl der Kanten
   /**  Erzeugt ein Polygon mit edges.length Kanten
     */
   Polygon(Edge[]  edges) {
       super(edges);
       count = edges.length();
       If (edges[count-1].endPoint != edges[count[0].startPoint)
         // Fehler: Programm abbrechen
       // Prüfe Überschneidungsfreiheit der Kanten
    }
   double area(){…}        // Berechnung des Flächeninhalts
…
}
```

Initialisierung von Objekten

- Algorithmus bei Aufruf eines Konstruktors der Klasse K
  1. Reservierung des Speichers und Basisinitialisierung der Datenfelder der Klasse K:
     - 0 für Zahlen,
     - false für boolean-Datentyp,
     - *null* für Referenzvariablen.
  2. Aufruf der entsprechenden Konstruktoren der Oberklassen
  3. (Zweite) Initialisierung der Datenfelder durch ihre Initialisierungsausdrücke
  4. Ausführung des Rumpfs des Konstruktors

Aber eigentlich ist alles ganz einfach!

- Einfach den Konstruktor so schreiben dass er die Datenfelder des Objekts korrekt initialisiert.
  - Initialisierung der Datenfelder, die in der Klasse definiert wurden.
- Gegebenenfalls kann man in der ersten Zeile des Konstruktors mit „super“ und mit „this“ gezielt andere Konstruktoren aufrufen.
- Da der Rumpf des „lokalen“ Konstruktors als letztes ausgeführt wird, kann man hier den Datenfeldern neue Werte geben.

Die Bedeutung von super

Schlüsselwort super tritt in zwei verschiedenen Bedeutungen auf

- Aufruf von Konstruktoren der Oberklasse
- Zugriffsmöglichkeit auf die Datenfelder und Methoden der Oberklasse:
  - **super.**<Datenfeldname>
  - **super.**<Methodenname>(…)

    Beim Methodenaufruf wird hier die Methode aus der Oberklasse benutzt und nicht die aus der erweiterten Klasse.

Bemerkung

- In Java ist der Ausdruck super.super.<Name> **nicht** erlaubt.
  - Seien OOK, OK und K drei Klassen, wobei OOK direkte Oberklasse von OK und OK direkte Oberklasse von K ist.
  - Sei m() eine Methode von OOK, die in OK und K jeweils überschrieben wird.
  - Dann kann in der Klasse K (über super) nur die Methode m() der Klasse OK, aber nicht die Methode m() der Klasse OOK aufgerufen werden.

10.6 Zugriffsrechte

- In Java gibt es vier verschiedene Zugriffsrechte
  - public
  - private
  - protected
  - package private

Zugriffsrecht public

- **Zugriffsschutz public steht vor dem Datenfeld bzw. vor der Methode**
  - Datenfelder und Methoden sind aus **jeder** anderen Klasse zugreifbar.

Zugriffsrecht private

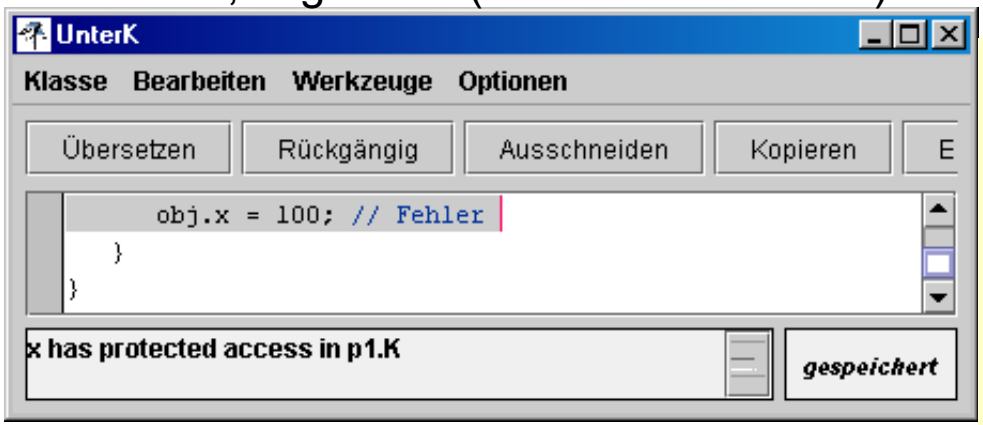
- **Zugriffsschutz private steht vor dem Datenfeld bzw. vor der Methode**
  - Datenfelder und Methoden sind aus **keiner** anderen Klasse zugreifbar.
- Methoden und Datenfelder sind nur in der Klasse zugreifbar, in der sie auch deklariert wurden.
  - Auch in Unterklassen kann nicht auf die Datenfelder und Methoden zugegriffen werden.

Diskussion

- Sei OK eine Klasse und m() eine Methode aus OK, die mit dem Attribut **public** gekennzeichnet ist und die auf **private** gekennzeichnete Datenfelder zugreift.
- Sei K eine direkte Unterklasse von OK.
  - Dann kann die Methode m() mit einem Objekt der Klasse K aufgerufen werden.
  - Privat deklarierte Datenfelder sind auch in den Objekten der Unterklasse vorhanden. Sie können jedoch nicht mehr direkt angesprochen werden.

Zugriffsrecht protected

- **Zugriffsschutz protected steht vor dem Datenfeld bzw. vor der Methode**
  - Datenfelder und Methoden sind über Objekte in den Klassen, die **zum gleichen Paket** gehören, zugreifbar.
  - Datenfelder und Methoden sind in **Unterklassen**, die in **anderen Paketen** definiert werden, zugreifbar (aber nur über "this")




```
package p1;
public class OK {
    protected int x;
    // ...
}
```

```
package p2;
import p1.OK;

class UK extends OK{
    void ok() {
        this.x = 100;
    }
    void nichtOk() {
        OK obj = new OK();
        obj.x = 100; // Fehler
    }
}
```

- im gleichen Paket ist der Zugriff aber erlaubt:

```
package p1;

public class OK {
   protected int x;
   // ...
}
```

```
package p1;

class UK extends OK{
   void ok() {
        this.x = 100;
   }

   void hierDochOk() {
      OK obj = new OK();
      obj.x = 100; // Korrekt!
   }
}
```

Warum wird überhaupt protected benötigt?

- Programme treten in mindestens zwei verschiedenen Rollen auf:
  - Programme werden durch einen Systementwickler erstellt und gewartet.
  - Programme werden durch Anwender benutzt.
- Entwickler sollten Zugriffsrechte für viele Datenfelder und Methoden eines Programms besitzen.
  - Java: Entwickler können die als „public“ und „protected“ gekennzeichnete Methoden und Datenfelder verwenden.
- Ein Anwender besitzt im allgemeinen weniger Zugriffsrechte als ein Entwickler.
  - Java: Anwender dürfen alle als „public“ gekennzeichnete Eigenschaften verwenden.

Zugriffsrecht package private

- **Keine explizite Angabe eines Zugriffsschutzes vor dem Datenfeld bzw. vor der Methode**
  - Datenfelder und Methoden sind über Objekte in den Klassen, die **zum gleichen Paket** gehören, zugreifbar.

- Vergleich mit Zugriffsschutz protected
  - Das Zugriffsrecht package private ist restriktiver als protected.
  - In Unterklassen, die in einem anderen Paket liegen, ist der Zugriff nicht mehr erlaubt.

Zugriffsrechte und Vererbung

- Beim Überschreiben von Methoden ist es erlaubt die Zugriffsrechte zu verändern. Dabei gilt:
  - Die Zugriffsrechte in einer überschriebenen Methode können aufgeweicht werden, aber nicht restriktiver werden.
    - Z. B.: Eine protected-Methode aus der Oberklasse kann zu einer public-Methode in der Unterklasse werden, aber nicht umgekehrt.
      - Hierzu wird die Methode überschrieben und der public-Modifier benutzt
      - Die überschreibende Methode kann z.B. einfach über „super“ die Implementierung aus der Oberklasse aufrufen und diese so zugreifbar machen.

10.7 Abstrakte Klassen

- Abstrakte Klassen sind spezielle Klassen, für die **keine direkten Objekte** erzeugt werden können (ähnlich wir bei Interfaces).
  - Abstrakte Klassen können ***abstrakte Methoden*** besitzen, die keinen Rumpf haben und in abgeleiteten Klasse noch überschrieben und implementiert werden müssen.
  - Unterschied zu Interfaces
    - Abstrakte Klassen können Datenfelder, Konstruktoren und nicht-abstrakte Methoden besitzen.
    - Eine Unterklasse kann nur **eine** abstrakte Klasse erweitern.

- Abstrakte Klassen fassen typischerweise gemeinsame Eigenschaften der Unterklassen zusammen, ohne dass Objekte erzeugt werden können!
  - In einer Universität gibt es bereits die drei Klassen Professoren, Mitarbeiter und Nicht-wissenschaftliche Angestellte.
    - Diese drei Klassen haben folgende Felder gemeinsam: PersonalId, Abteilung.
  - Durch Anlegen einer abstrakten Klasse Personal können diese Felder (und entsprechende Methoden) in der abstrakten Klasse zur Verfügung gestellt werden.

Syntax

- Eine abstrakte Klasse wird mit dem Schlüsselwort **abstract** gekennzeichnet, das direkt vor **class** (bzw. den Modifikatoren der Klasse) steht.
- Eine abstrakte Methode hat ebenfalls das Schlüsselwort **abstract** vorangestellt.
  - Eine abstrakte Methode besitzt in Java keinen Methodenrumpf.

```
public abstract class GeoObjectWithExtent {
        private double x, y;

        GeoObjectWithExtent(double a, double b) {
                x = a; y = b;
        }
        public abstract Rectangle envelope();

        public abstract double area();

        public double coverage() {
                return area() / envelope().area();
        }
}
```

Konstruktoren können bereits in abstrakten Klassen vorhanden sein.

Die Implementierung von Methoden kann bereits abstrakte Methoden der Klasse verwenden.

Implementierung einer abstrakten Klasse

- Die konkreten Unterklassen einer abstrakten Klasse müssen die abstrakten Methoden implementieren.

```
public class Circle extends GeoObjectWithExtent {
        private double radius;

        public Circle(double a, double b, double r) {
                super(a,b);
                radius = r;
        }

        @Override
        public Rectangle envelope() {
                return new Rectangle (x, y, radius, radius);
        }

        @Override
        public double area() {
                return Math.PI*radius*radius;
        }
}
```

Aufruf des Konstruktors der Oberklasse.

Überschreiben der abstrakten Methode area.

Zusammenfassung

- Klassenerweiterung
  - Unterklasse erweitert eine Oberklasse
    - Überschreiben von Methoden
    - Konstruktoren
  - Zugriffsrechte
  - Polymorphie
    - Variablen vom Typ der Oberklasse können auf Objekte der Unterklasse verweisen.
    - Dynamisches Binden
      - Klasse des Objekts bestimmt, welche Implementierung bei überschriebenen Methoden verwendet wird.
- Abstrakte Klassen
  - Keine Objekterzeugung