Wiederholung

In der heutigen Betrachtungsweise der Informatik besteht die Welt aus Objekten. Jedem Objekt liegt ein Bauplan zugrunde, nach dem es erstellt wurde.

- Diesen Bauplan nennt man ***Klasse***. Eine Klasse legt fest, welche ***Attribute*** (Farbe, Breite, Höhe, ...) und welche ***Methoden*** (`speichern(), setzeFarbe(...)` ...) solche Objekte haben sollen.

  Der logische Aufbau einer Klasse wird in der Regel durch eine sog. ***Klassen-Karte*** veranschaulicht. Eine Klassen-Karte ist ein Rechteck, das von oben nach unten aus drei Bereichen besteht:

  Oben: *Name der Klasse*

  Mitte: *Attribute und Daten-Typen*

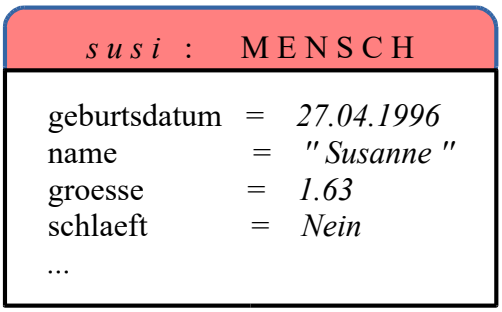
  Unten: *Methoden*

- Hat man nun mehrere ***Objekte*** einer Klasse erzeugt, so können sich diese durch ihre ***Attribut-Werte*** (`Farbe: ''rot''`, `Radius: 3cm`, ...) unterscheiden.

  Objekte werden durch eine ***Objekt-Karte*** veranschaulicht. Objekt-Karten haben im Vergleich zu Klassen-Karten abgerundete Ecken.
  Jede Objekt-Karte besteht aus zwei Bereichen:

  Oben: *Name und Klasse des Objekts*

  Unten: *Attribute und deren Werte*



  Objekte spricht man immer in ***Punktnotation*** an:

```
susi.laecheln()
susi.schlafen(7 Stdunden)
```

  allgemein: `Objektname.Methodenaufruf`

- ***Verändernde Methoden*** bringen das Objekt in einen anderen Zustand:

```
setzeSchriftgroesse(12)
```

- ***Sondierende Methoden*** geben eine Antwort auf eine Frage

```
nenneDeinenNamen()
```

- ***Übergabe-Parameter*** kommen in die runden Klammern:

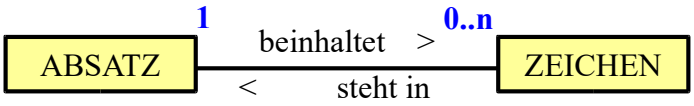
```
schlafen(7 Stunden)
```

- Methoden ohne Übergabe-Parameter erkennt man an den leeren Klammern:

```
laecheln()
```

Die Objekte stehen meist miteinander in irgendeiner Beziehung. Diese Beziehungen werden in einem sog. ***Klassen-Diagramm*** dargestellt.

- Dazu nimmt man **nur** den oberen Teil der Klassen-Karte (**Klassen-Name**).
- Man zeichnet **Verbindungslinien** zwischen zwei Klassen, wenn zwischen ihnen eine ***Beziehung*** besteht.
- Die Linien werden aussagekräftig beschriftet und die Beschriftung zur besseren Lesbarkeit mit einem **Richtungspfeil** versehen.
- Oft gibt man auch noch ***Kardinalitäten*** an. Darunter versteht man Zahlen, die angeben, wie viele Objekte der einen Klasse in Beziehung zur anderen Klasse stehen können.

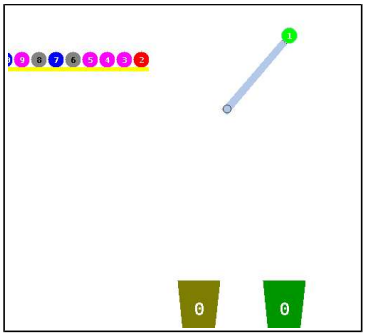


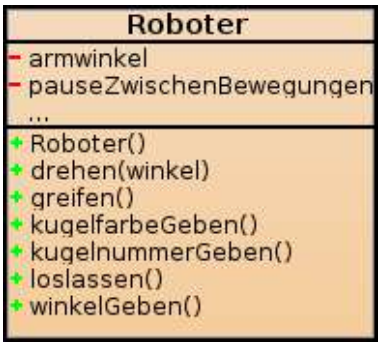
Dieses Klassen-Diagramm besagt:

„Ein Absatz kann kein, ein oder mehrere Zeichen enthalten.
Aber jedes Zeichen steht in genau einem Absatz.“

Aufgaben mit Beispiel-Objekten

1. Greif-Roboter





a) *Zähle alle Methoden mit Übergabe-Parameter auf.*

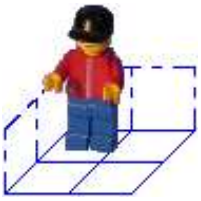
b) *Zähle alle Methoden ohne Übergabe-Parameter auf..*

c) *Zähle alle sondierenden Methoden auf.*

d) *Zähle alle verändernden Methoden auf.*

e) *Kannst du dir weitere Klassen vorstellen, zu denen der Roboter eine Beziehung hat?*
Zeichne dazu ein Klassen-Diagramm.

2. Roboter Karol



ROBOTER
+ ROBOTER(inWelt) + ROBOTER(startX, startY, startBlickrichtung, inWelt) + Aufheben() + Hinlegen() + IstMarke() + IstWand() + IstZiegel() + LinksDrehen() + MarkeLoeschen() + MarkeSetzen() + RechtsDrehen() + Schritt() + TonErzeugen()

a) *Nenne alle Methoden mit Übergabe-Parametern.*

b) *Nenne alle sondierenden Methoden.*

c) *Nenne alle verändernden Methoden.*

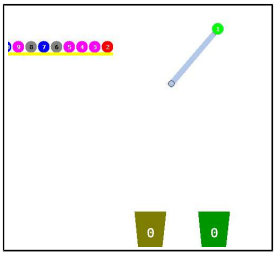
d) *Was geschieht wohl bei den beiden Methoden* `ROBOTER(...)`*?*
Welchen besonderen Namen hat sie?

e) *Die Klassenkarte von Karol zeigt keine Attribute.*
Welche Attribute könnte er haben?

f) *Zeichne eine Objekt-Karte von Robot-Karol wie er im Bild gezeigt ist.*

g) *Kannst du dir eine weitere Klasse vorstellen, zu der die Klasse ROBOTER eine Beziehung aufweist?*
Zeichne dazu ein Klassen-Diagramm.

3. Übe mit dem Greif-Roboter in BlueJ



a) *Erzeuge ein Roboter-Objekt und nenne es* `greifi`*.*

b) *Veranlasse* `greifi`*, die erste Kugel zu greifen.*

c) *Lasse* `greifi` *den Arm 20 Grad nach rechts drehen.*

d) *Frage* `greifi` *nach der Farbe der gegriffenen Kugel.*

e) *Frage* `greifi` *nach dem aktuellen Winkel.*

f) *Finde heraus, wo der Winkel 0° ist.*

g) *Schreibe die Methodenaufrufe von a) bis f) in Punktnotation auf.*

Daten-Typen

Daten-Typen legen fest, von welcher logischen Art ein Attribut ist.

Die gängigsten Daten-Typen und worauf man dabei achten muss siehst du in der Tabelle:

Daten-Typ	*Bedeutung*	*Beispiel*	*Anmerkung*
`int`	ganze Zahl	`123 ; -321`	
`double`	Kommazahl	`13.24`	*Dezimalpunkt*
`char`	einzelnes Zeichen	`'a' ; '5'`	*Hochkommata*
`String`	mehrere Zeichen	`"Hallo"`	*Anführungszeichen*
`boolean`	Wahrheitswert	`true ; false`	

4. **Übe mit Parametern und Daten-Typen**

a) *Erkundige dich über das Koordinaten-System im BlueJ-Projekt alpha_Formen.*

b) *Erzeuge ein Objekt der Klasse Kreis und nenne es* `sonne`*.*

Färbe es gelb und verschiebe es in die rechte obere Ecke.

c) *Erzeuge von der Klasse RECHTECK auch ein Objekt und nenne es* `wand`*.*
Färbe es weiß ("weiss").

d) *Erzeuge von der Klasse Dreieck ein drittes Objekt und nenne es* `dach`*.*
Färbe das Dach rot.

e) *Kannst du daraus ein Haus mit Sonne erstellen?*

f) *Ergänze dein Bild um ein beliebiges viertes Objekt.*
Schreibe alle dafür nötigen Methodenaufrufe in Punktnotation auf!

5. **Vorübung zur Erstellung eigener Klassen**

a) *Öffne das BlueJ-Projekt alpha_Formen_Zeichnung.*

b) *Doppelklicke auf die Klasse ZEICHNUNG und sieh dir den JAVA-Code in Ruhe an. Hast du Fragen?*

c) *Schreibe nun in nach den Zeilen*

```
// … und legst ihr Aussehen fest
// (Fachsprache: initialisieren der Attribute)
```

die Methodenaufrufe, die du in Aufgabe 4 aufgeschrieben hast.

Beende jede Zeile mit einem Strichpunkt!

Klicke nach jeder Zeile auf „übersetzen" und beseitige deine Tippfehler.

d) *Wechsle nach dem Initialisieren jedes einzelnen Bausteins ins BlueJ-Hauptfenster und erzeuge mit Rechtsklick auf die Klasse ZEICHNUNG ein neues ZEICHNUNG-Objekt.*

Sieht es aus wie du wolltest? *Bessere bei Bedarf nach ...*

Klassen erstellen

Bevor du selbst programmierte Objekte erzeugen kannst musst du in einer Klasse den Bauplan für deine Objekte schreiben.

BALL
radius : int *farbe* : String

```
class BALL
{
    int radius;
    String farbe;
}
```

6. **Erste eigene Klasse**

Erstelle ein neues BlueJ-Projekt und nenne es Uebung-01.

Erstelle in diesem Projekt eine neue Klasse mit dem Namen BALL.

Schreibe den JAVA-Code von oben hinein.

Übersetze die Klasse fehlerfrei.

Erzeuge ein erstes Objekt deiner Klasse BALL. (Rechtsklick auf die Klassen-Karte)

Öffne den Objekt-Inspektor. (Doppelklick auf das rote Objekt-Symbol).

Was stört dich beim Anblick des Objekt-Inspektors?

Methoden erstellen

Dass der `radius` deines ersten Kreis-Objekts `0` war und seine `farbe` `"null"` liegt daran, dass wir bisher zwar Attribute ***deklariert*** (Variablen „bestellt“) haben.

Aber wir haben sie nicht ***initialisiert***, d.h. ihnen Werte zugewiesen.

Wertzuweisung geschieht mit dem **„=“ – Operator**: `radius = 20;`

Unterscheide die Begriffe

- Variablen ***deklarieren*** (beantragen, z.B. `int radius`)
- Variablen ***initialisieren*** (Anfangszustand festlegen, z.B. `radius = 20`)

Die Konstruktor-Methode

Mit welchen Attribut-Werten ein neu erzeugtes Objekt ausgestattet ist, bestimmt eine besondere Methode, der ***Konstruktor***. Ihn rufen wir auch auf, wenn wir mit ***new*** … ein neues Objekt erzeugen. Im Bild ist `Ball()` der Konstruktor der Klasse *BALL*. Durch seinen Aufruf wird ein neues BALL-Objekt erzeugt.

Randbemerkung:

Einen Standard-Konstruktor, der das Objekt erzeugt, ohne die Attribute zu initialisieren, erstellt JAVA unsichtbar im Hintergrund automatisch. Sonst hätten wir unser erstes Objekt der Klasse *BALL* nicht erzeugen können.

Beispiel:

```
class BALL {                          Klasse BALL deklarieren
   int radius;                        Attribute
   String farbe;                      deklarieren
   BALL() {                           Konstruktor-Methode:
      this.radius = 10;               Attribute
      this.farbe = "weiss";           initialisieren
   }
}
```

- Der ***Konstruktor*** muss immer **genau so geschrieben** werden **wie der Klassen-Name**.
- Er bekommt am Ende **runde Klammern** um ihn als Methode zu kennzeichnen.
- Den Konstruktor muss man zusammen mit dem ***new*** **– Operator** immer aufrufen, wenn man ein neues Objekt haben möchte.
  z.B. `new BALL()`

7. Konstruktor-Methode

Erstelle nun in deiner Klasse BALL einen Konstruktor wie im Beispiel oben.

Klicke nun mit rechts auf die Klassenkarte und erzeuge mit `new BALL()` *ein neues BALL-Objekt.*

Doppelklicke auf das rote Objekt-Symbol und betrachte dein BALL-Objekt im Objekt-Inspektor.

Haben die Attribute die von dir im Konstruktor programmierten Werte?

8. Übung – Klasse WAND

Schreibe analog zur Klasse BALL nun zur Übung eine weitere Klasse WAND.

- *Eine Wand soll die Attribute* `hoehe`, `dicke` *und* `farbe` *haben.*
- *Im Konstruktor soll die* `hoehe` *den Wert* `230`, *die* `breite` *den Wert* `40` *und die* `farbe` *den Wert* `„grau“` *bekommen.*
- *Erzeuge ein WAND-Objekt und erkunde es im Objekt-Inspektor.*
  Sind alle Attribut-Werte wie gewollt initialisiert worden?

Sondierende Methoden

Natürlich kannst du in deinen eigenen Klassen auch selbst sondierende Methoden schreiben. So soll dein Ball anderen Objekten z.B. seinen Radius oder seine Farbe verraten können.

Beispiel:

BALL
radius : int *farbe* : String
nenneRadius() : int

```
class BALL {                          // Klasse BALL deklarieren

   int radius;                        // Attribute
   String farbe;                      // deklarieren

   BALL() {                           // Konstruktor-Methode:
      this.radius = 10;               // Attribute
      this.farbe = "weiss";           // initialisieren
   }

   int nenneRadius() {                // neue Methode mit Antwort-Daten-Typ int
      return this.radius;             // gibt als Antwort den Radius
   }
}
```

- Eine ***sondierende Methode*** hat immer folgenden Aufbau:

```
Datentyp-der-Antwort    Name-der-Methode()    {
    return Attribut;
}
```

- **Vor dem Namen** der sondierenden Methode muss der **Datentyp der Antwort** stehen.
- Jede Methode bekommt **am Ende des Namens runde Klammern**, auch wenn sie keine Übergabe-Parameter hat.
- In geschweiften Klammern gibt man **mit dem Schlüsselwort `return` die Antwort**.

9. sondierende Methode

- *Erstelle nun in deiner Klasse BALL die sondierende Methode `nenneRadius()` wie oben beschrieben.*
- *Übersetze die Klasse neu und beseitige alle eventuellen Fehler.*
- *Erzeuge ein Objekt der Klasse BALL und betrachte es im Objekt-Inspektor. Rufe nun die neue Methode auf. Bekommst du den Wert als Antwort, der im Objekt-Inspektor zu sehen ist?*

10. weitere sondierende Methode

- *Schreibe nun eine weitere sondierende Methode, die dir die Farbe des Balls als Antwort gibt. Beachte dabei, den Daten-Typ von `farbe`.*
- *Übersetze die Klasse erneut und beseitige eventuelle Fehler.*
- *Erzeuge ein Objekt der Klasse BALL und betrachte es im Objekt-Inspektor. Rufe nun die neue Methode auf. Bekommst du den Wert als Antwort, der im Objekt-Inspektor zu sehen ist?*

Verändernde Methoden – Übergabe-Parameter

Auch Methoden zum Verändern des Zustands eines Objekts sind nicht schwer selbst zu erstellen. Diese Methoden brauchen allerdings einen Übergabe-Parameter, welcher den neuen Attribut-Wert aufnehmen kann. In der Regel geben verändernde Methoden keine Antwort.

Beispiel:

BALL
radius : int *farbe* : String
nenneRadius() : int *setzeRadius(rNeu:* int*) :* void

```
class BALL {                                  Klasse BALL deklarieren

   int radius;                                Attribute
   String farbe;                              deklarieren

   BALL() {                                   Konstruktor-Methode:
      this.radius = 10;                       Attribute
      this.farbe = "weiss";                   initialisieren
   }

  int nenneRadius() {                         Methode mit Antwort-Daten-Typ int
      return this.radius;                     gibt als Antwort den Radius
   }

   void setzeRadius(int rNeu) {               verändernde Methode mit Übergabe-
      this.radius = rNeu;                     Parameter rNeu vom Daten-Typ int
   }
}
```

- Eine ***verändernde Methode*** hat immer folgenden Aufbau:

```
void  Name-der-Methode(Daten-Typ  Name-des-Paramters)  {
   Attribut = Parameter;
}
```

- **Vor dem Namen** der verändernden Methode muss das **Schlüsselwort *void*** stehen.
  Damit gibt man ausdrücklich an, dass **keine Antwort** erfolgen soll.
- **In den runden Klammern** wird ein **Übergabe-Parameter** deklariert.
  Hierfür schreibt man erst den Daten-Typ des Parameters und danach einen beliebigen Namen für den Parameter.
- **In den geschweiften Klammern** erfolgt eine **Wertzuweisung** ähnlich den Wertzuweisungen im Konstruktor. Nur wird hier der konkrete Wert (z.B. *20*) ersetzt durch den Namen des Übergabe-Parameters (z.B. *rNeu*).
  So kann der Benutzer Aufruf der Methode einen Wert für den Parameter mitgeben, der dann dem Attribut zugewiesen wird.

11. verändernde Methode

- *Erstelle nun in deiner Klasse BALL die verändernde Methode* `setzeRadius(int rNeu)` *wie oben beschreiben.*
- *Übersetze die Klasse neu und beseitige alle eventuellen Fehler.*
- *Erzeuge ein Objekt der Klasse BALL und betrachte es im Objekt-Inspektor. Rufe nun die neue Methode auf.*
  Verändert sich im Objekt-Inspektor der Attribut-Wert wie gewünscht?

12. weitere verändernde Methode

- *Schreibe nun eine weitere verändernde Methode, welche die Farbe des Balls verändert.*
  Beachte dabei, den Daten-Typ von `farbe`*.*
- *Übersetze die Klasse erneut und beseitige eventuelle Fehler.*
- *Erzeuge ein Objekt der Klasse BALL und betrachte es im Objekt-Inspektor. Rufe nun die neue Methode auf.*
  Ändert sich der Wert im Objekt-Inspektor wie gewünscht?

13. Übung – Methoden in der Klasse WAND

- *Schreibe nun auch in der Klasse WAND zu all deinen Attributen sondierende und verändernde Methoden.*
- *Übersetze nach jeder neuen Methode deine Klasse und teste die Methode auf ihre Funktion.*

Wäre es nicht schön, wenn die Objekte unserer selbst erstellten Klassen endlich auch grafisch dargestellt würden? Ist denn unsere Klasse *BALL* nicht sehr ähnlich der Klasse *KREIS*, deren Objekte man sofort sehen kann, sobald man den *new*-Operator aufgerufen hat … ?

Zusatzmaterial: Test 1

Vererbung 1 – Das Rad nicht jedes Mal neu erfinden

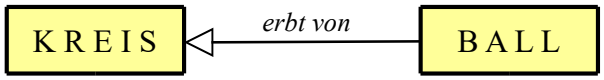
Sehr oft besteht der Wunsch, **eine bereits bestehende Klasse** zu **erweitern**. Man bekommt z.B. die Klasse *KREIS* von einem anderen Programmierer vorgegeben und hätte so gerne noch ein paar weitere Attribute (Eigenschaften) oder Methoden (Fähigkeiten) ergänzt.

Dazu muss man nicht den Programm-Code des anderen Programmierers durchlesen und vollständig verstehen. Es ist auch sehr ungeschickt, in der Klasse des anderen etwas zu verändern, denn das birgt die Gefahr, den vorgegebenen Code unwiderruflich zu zerstören! Man schreibt deshalb besser In einer anderen Datei) eine Erweiterung zu der vorgegebenen Klasse. In der Fachsprache des Programmierens sagt man, **man „erbt“** einfach **von dieser Klasse**.

Erben bedeutet, dass man (im Normalfall) **alle Attribute und Methoden** der vorgegebenen Klasse in einer eigenen Klasse **unverändert übernehmen** kann ohne dass man diesen Code noch einmal schreiben muss. Zusätzlich dazu kann man nun einerseits **weitere Attribute und Methoden ergänzen**. Man kann aber im Extremfall auch die bereits bestehenden Methoden verändern und den eigenen Bedürfnissen anpassen.

Für all das muss man **nichts in der vorgegebenen Klasse verändern**. Man muss **die vorgegebene Klasse nur aus Anwender-Sicht kennen**.

Vererbung stellt man ***im Klassen-Diagramm*** durch einen **durchgezogenen Pfeil mit einem Dreieck als Spitze** dar.



Hiermit wird dargestellt, dass die Klasse *BALL* die Klasse *KREIS* erweitert.
Man sagt auch, dass die Klasse *BALL* von der Klasse *KREIS* erbt.

Die Klasse, von der geerbt wird (hier *KREIS*), nennt man ***Superklasse*** *(Oberklasse)*, die erbende Klasse (hier *BALL*) nennt man ***Subklasse*** *(Unterklasse)*.

Wie einfach das Erweitern einer Klasse in JAVA geht, siehst du am folgenden Beispiel 1:

Betrachte zunächst die Klassen-Karte der Klasse KREIS in BlueJ genau. Welche Attribute, Konstruktoren und weiteren Methoden besitzt die Klasse KREIS? All das erbst du gleich, ohne es selbst in Code fassen zu müssen.

Beispiel 1:

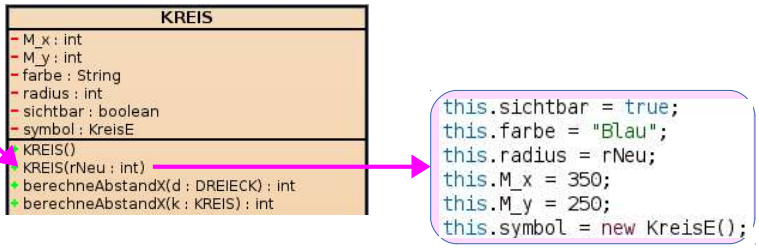
```
class BALL extends KREIS {

    BALL(int rNeu) {
        super(rNeu);
    }

}
```

Klasse BALL ***erbt von Klasse KREIS***

Konstruktor-Methode
geerbten ***Superkonstruktor aufrufen***



Im Rumpf des Konstruktors der Klasse BALL musst du ausdrücklich sagen, dass du den Code des Konstruktors der Klasse KREIS ausführen möchtest. Dies tut man nicht mit `new KREIS(rNeu)`*, weil man damit zusätzlich zum BALL-Objekt ein weiteres KREIS-Objekt erzeugen würde. Durch den Aufruf von* ***`super(rNeu)`*** *wendet man den Code des geerbten Konstruktors am eigenen Objekt an.*

- Eine Klasse erweitert mit dem Schlüsselwort ***extends*** eine andere Klasse.
- Im Konstruktor der erbenden Klasse wird als erster Befehl der geerbte **Superkonstruktor** der Superklasse aufgerufen.
  super(...)
- Die Subklasse verfügt nun (in der Regel) über alle Attribute und Methoden der Superklasse.
  Deklariere deshalb in der Subklasse NIE Attribute oder Methoden, die du geerbt hast *– damit würdest du das Geerbte „verdecken" und damit kaputt machen.*
- Benötigt man nun weitere Attribute, über welche die Superklasse noch nicht verfügt, so deklariert man sie einfach oberhalb des Konstruktors. Initialisiert werden sie im Konstruktor aber unterhalb des Superkonstruktors. *(s. Beispiel 2)*
- Benötigt man weitere Methoden, so schreibt man sie einfach unterhalb des Konstruktors.

Beispiel 2:

BALL
besitzer : String . . .
nenneUmfang() : double . . .

```
class BALL extends KREIS {                  Klasse BALL erbt von Klasse KREIS

    String besitzer;                        ein neues Attribut deklarieren

    BALL(int rNeu) {                        Konstruktor-Methode
        super(rNeu);                        geerbten Superkonstruktor aufrufen
        this.besitzer = "Hans";             das neue Attribut initialisieren
    }

    double nenneUmfang() {                  eine neue Methode
        return  2 * super.nenneRadius() * 3.14 ;
    }

}
```

Beachte, dass du geerbte Methoden oder Attribute mit ***super*** *aufrufst (geerbter Konstruktor:* ***super(rNeu)****, geerbte Methode:* ***super.nenneRadius()****), wohingegen neu hinzugekommene Attribute und Methoden der Subklasse mit* ***this*** *aufgerufen werden (neues Attribut initialisieren:* ***this.besitzer = ''Hans''****)!*

Welchen geerbten Konstruktor nehme ich?

Wenn du nur einen Konstruktor geerbt hast, dann nimmst du natürlich diesen einfach her.

Hast du mehrere Konstruktoren geerbt, so nimmst du den Konstruktor her, der am besten deine Zwecke erfüllt, so dass du nach dem Aufruf des Super-Konstruktors möglichst wenig weiteren Code schreiben musst.

- Interessiert dich in deinem BALL-Konstuktor z.B. der Radius des Balls nicht *(`public BALL()`)*, so nimmst du ***super()*** und rufst damit den Code von ***KREIS()*** auf. Der Radius wird dabei auf einen dir unbekannten Wert gesetzt.
- Interessiert dich in deinem BALL-Konstuktor dagegen der Radius des neuen Balls *(`public BALL(int rNeu)`)*, so nimmst du ***super(rNeu)*** und rufst damit den Code von ***KREIS(rNeu)*** auf. So sparst du dir den Methodenaufruf `this.setztRadius(rNeu)`.

14. Klasse BALL erbt von Klasse KREIS

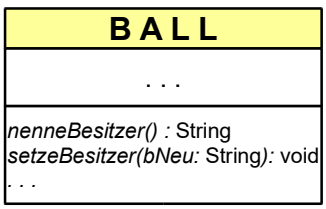
- *Öffne nun in BlueJ das Projekt alpha_Formen und speichere es unter dem Namen Vererbungs-Uebung neu ab.*
- *Erzeuge darin eine neue Klasse BALL mit dem JAVA-Code aus Beispiel 1.*
- *Übersetze die Klasse und beseitige alle eventuellen Fehler.*
- *Erzeuge ein Objekt der Klasse BALL und betrachte es im Objekt-Inspektor. Verfügt es über geerbte Attribute? Welche Werte haben diese?*
- *Klicke nun mit rechts auf die BALL-Objekt-Karte und sieh nach, ob der Ball auch Methoden geerbt hat. Betrachte dabei das Menü „geerbt von ...“ Sind es die bekannten Methoden der Klasse KREIS?*
- *Rufe auf diese Weise einige Methoden auf und prüfe, ob sie wie erwartet funktionieren.*

15. Erweiterung der Klasse BALL

- *Ergänze nun den JAVA-Code deiner Klasse BALL wie in Beispiel 2.*
- *Übersetze die Klasse und beseitige alle eventuellen Fehler.*
- *Erzeuge erneut ein BALL-Objekt und erkunde es im Objekt-Inspektor. Ist das neue Attribut vorhanden? Hat es den erwarteten Wert? Sind dennoch die geerbten Attribute vorhanden?*
- *Prüfe durch Rechtsklick auf das Objekt, ob auch die neue Methode vorhanden ist. Funktioniert sie? Sind auch die geerbten Methoden noch vorhanden?*

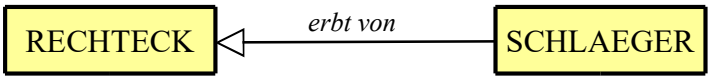
16. Noch mehr Erweiterung der Klasse BALL

- *Lasse dir die Klassenkarte der Klasse KREIS anzeigen und sieh nach, wie die Attribute für die Mittelpunkt-Koordinaten benannt wurden.*
- *Schreibe nun in der Klasse BALL die sondierende und eine verändernde Methode für die Variable* `besitzer` *deines Balls zurück gibt.*
- *Übersetze die Klasse, erzeuge ein BALL-Objekt und teste die neuen Methoden. Funktioniert alles wie erwartet?*

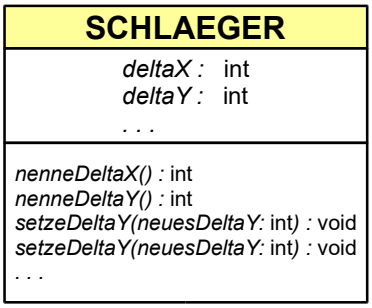


17. Übung – Klasse SCHLAEGER

- *Schreibe nun im Projekt Vererbungs-Uebung eine weitere Klasse SCHLAEGER, die von der Klasse RECHTECK erbt.*



- *Deklariere in der Klasse zwei weitere ganzzahlige Attribute* `deltaX` *und* `deltaY`*. (Wir brauchen sie später, damit der Ball sich bewegen kann.)*
- *Initialisiere die beiden neuen Attribute jeweils mit dem Wert 1.*
- *Schreibe zu jedem der beiden Attribute eine sondierende und eine verändernde Methode.*



- *Übersetze die Klasse, erzeuge ein SCHLAEGER-Objekt, erkunde es im Objekt-Inspektor und teste deine Methoden.*
  Klappt alles wie erwartet?