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*

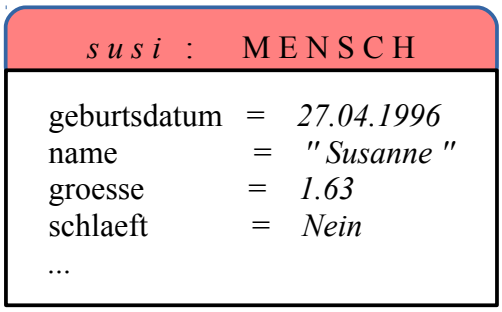
  Unten: *Methoden*

- 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:

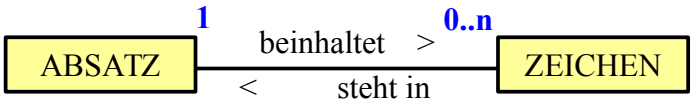
```
schlafen(7 Stunden)
```

- 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.

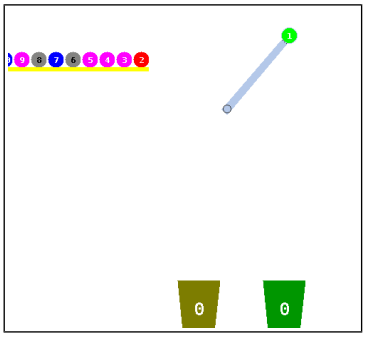


Dieses Klassen-Diagramm besagt:

„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.*

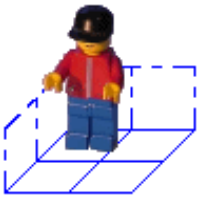
b) *Zähle alle Methoden ohne Übergabe-Parameter auf..*

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.*

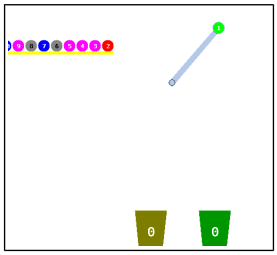
d) *Was geschieht wohl bei den beiden Methoden* `ROBOTER(...)`*?*
Welchen besonderen Namen hat sie?

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

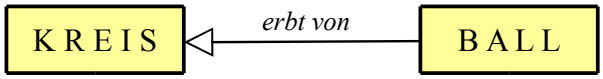
Sehr oft besteht der Wunsch, **eine bereits bestehende Klasse** zu **erweitern**. Man bekommt z.B. die Klasse *KREIS* von einem anderen Programmierer vorgegeben und hätte so gerne noch ein paar weitere Attribute (Eigenschaften) oder Methoden (Fähigkeiten) ergänzt.

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.

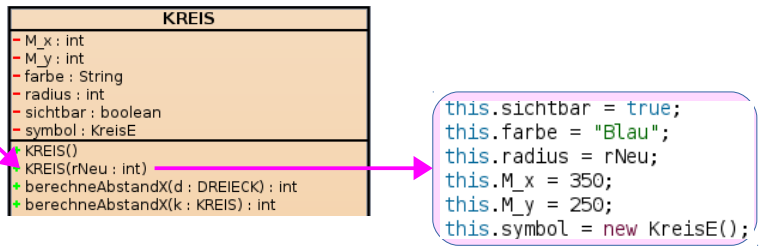
Beispiel 1:

```
class BALL extends KREIS {
    BALL(int rNeu) {
        super(rNeu);
    }
}
```

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

Konstruktor-Methode

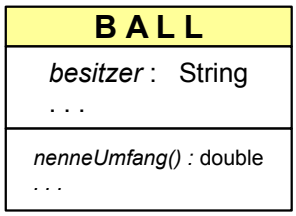
geerbten ***Superkonstruktor aufrufen***



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:




```
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

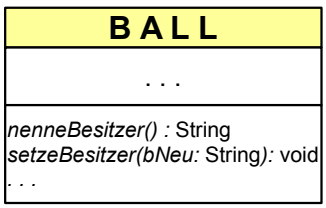
- *Öffne nun in BlueJ das Projekt alpha_Formen und speichere es unter dem Namen Vererbungs-Uebung neu ab.*
- *Erzeuge darin eine neue Klasse BALL mit dem JAVA-Code aus Beispiel 1.*
- *Übersetze die Klasse und beseitige alle eventuellen Fehler.*
- *Erzeuge ein Objekt der Klasse BALL und betrachte es im 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?*

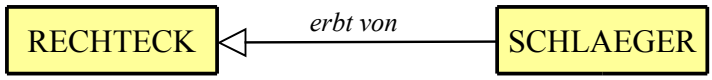
16. Noch mehr Erweiterung der Klasse BALL

- *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?*

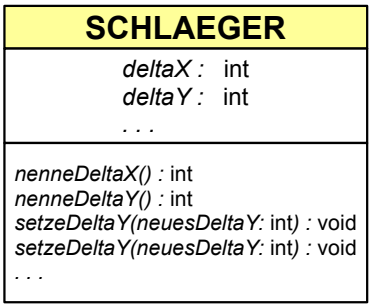


17. Übung – Klasse SCHLAEGER

- *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?

Kapselungs-Prinzip – Modifikatoren

Stelle dir vor, du gehst durch eine Menschenmenge und anschließend ist dein Geldbeutel verschwunden, jemand hat dir unbemerkt seinen Abfall in den Rucksack gepackt und dich mit Farbe beschmiert …

So in etwa geht es bisher den Objekten deiner selbst erstellten Klassen. Ihre **Attribute** sind **völlig ungeschützt**. Jeder kann damit tun, was er will.

Du wirst auf die Menschenmenge vermutlich im Vorfeld schon reagieren, indem du deinen Geldbeutel bei gut versteckst und darauf aufpasst … Einen Freund wirst du gerne mal auf Anfrage in deinen Rucksack greifen lassen um etwas dort abzulegen und wieder herauszuholen. Fremde sollen ohne dich vorher zu fragen nicht auf den Inhalt deines Rucksacks zugreifen können.

Beim Aufbau unserer eigenen Klassen können wir darauf reagieren, indem wir **Modifikatoren zur Sperrung der Attribute gegenüber anderen Objekten** einführen. Bei Bedarf können wir für andere Objekte Methoden zur Verfügung stellen, mit denen einige unserer Attribute in sinnvollen Grenzen und erst nach Anfrage verändert werden können.

Beispiel:

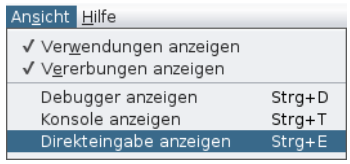
```
public class BALL {                                  Klasse BALL – jeder darf sie benutzen
    private String farbe;                            Attribut – für andere Objekte unzugänglich
    public BALL() {                                  Konstruktor – jeder darf Objekte erzeugen
        this.farbe = "rot";
    }
    public void setzeFarbe(String neueFarbe) {       verändernde Methode –
        if (neueFarbe == ''rot'' || neueFarbe == ''blau'') {   jeder darf kontrolliert
            this.farbe = neueFarbe;                  die Farbe verändern
        }
    }
}
```

Die Bedingung mit `if`*, der Vergleich mit* `==` *sowie das logische ODER* `||` *werden später genauer erklärt.*

Mit Hilfe von Modifikatoren kann man den Zugriff von anderen Objekten auf die eigenen Attribute und Methoden regeln. Man sagt auch, dass man die Attribute „verkapselt“ und nennt diese Technik deshalb das ***Kapselungs-Prinzip***.

- Attribute werden immer mit dem Modifikator `private` ausgezeichnet und sind somit für andere Objekte unzugänglich.
  Private Attribute können zwar vererbt werden, auf sie kann aber in der Subklasse nicht zugegriffen werden*, außer es gibt sondierende bzw. verändernde Methoden in der Superklasse!*
- Methoden werden (eigentlich) immer mit dem Modifikator `public` versehen und sind somit für andere Objekte zugänglich. Im Rumpf der Methode kann man dann den Zugriff sinnvoll beschränken.
- Es gibt noch den Modifikator `protected`.
  Seine Wirkung ist zu komplex und würde den Rahmen der zehnten Klasse sprengen.
  (Seine Wirkung unterscheidet sich erst dann von `public`*, wenn man sog. Packages bildet und von einem Package in ein anderes „greift“.)*

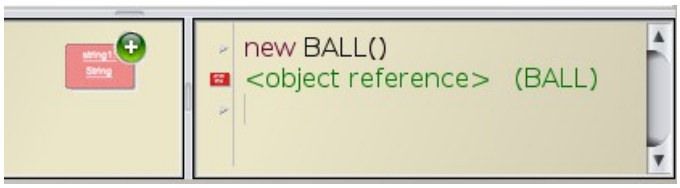
18. Kapselung der Klasse *BALL*



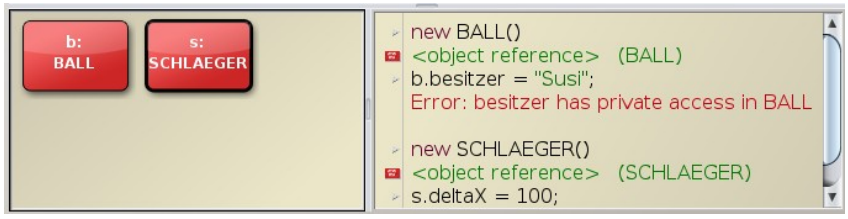
- *Öffne das Projekt Vererbungs-Uebung in BlueJ und speichere es unter dem Namen Kapselungs-Uebung.*
- *Ergänze nun in der Klasse BALL die Modifikatoren bei der Klassendeklaration, allen Attributen, dem Konstruktor und allen weiteren Methoden.*
  (Die Einschränkung `nur rot oder blau erlaubt!` Übernimmst du bitte vorerst von der Angabe. – Die ODER-Striche erhältst du links neben der y-Taste mit alt gr)

19. Demonstration der Kapselung

- *Öffne das Projekt Kapselungs-Uebung in BlueJ.*
  Klicke im Hauptmenü von BlueJ auf Ansicht und dann auf Direkteingabe.
- *Schreibe nun in der Direkteingabe `new BALL()` und drücke ENTER.*
  Klicke nun in der Antwort-Zeile auf das kleine rote Symbol, ziehe es nach links und lasse es dort los. Nenne das Objekt `b`.



- *Schreibe nun in der Direkteingabe `b.besitzer = "Susi";` und drücke ENTER.*
  Welche Reaktion erhältst du?
  Überprübe das Objekt `b` im Objekt-Inspektor.
- *Erzeuge nun in der Direkteingabe einen neues SCHLAEGER-Objekt, ziehe das Symbol wieder nach links und nenne es `s`.*
- *Schreibe nun in der Direkteingabe `s.deltaX = 100` und drücke ENTER.*
  Welche Reaktion erhältst du nun?
  Überprübe das Objekt `s` im Objekt-Inspektor.



20. Kapselung der Klasse *SCHLAEGER*

- *Kapsele nun auch die Klasse SCHLAEGER.*
- *Teste in der Direkteingabe, ob die Kapselung funktioniert.*
- *Erweitere die Klasse SCHLAEGER um sondierende und verändernde Methoden für die Attribute `deltaX` und `deltaY`.*
- *Teste in der Direkteingabe, ob die Methoden funktionieren.*

21. **Erweiterung der Klasse *BALL***

- *Erweitere auch die Klasse BALL um eine sondierende und eine verändernde Methode für das Attribut* `besitzer`*.*
- *Teste in der Direkteingabe, ob die Methoden funktionieren.*

Sicherung des bisher Gelernten

Beantworte folgende Fragen bevor du weiter machst:

1) Wie deklariert man eine neue Klasse? Gehe auf BlueJ und den JAVA-Code ein.

2) Wie und an welcher Stelle deklariert man die Attribute einer Klasse?

3) Wie deklariert man den Konstruktor einer Klasse? Was bewirkt er?

4) Wie und wo initialisiert man in einer Klasse die Attribute?

5) Wie lautet der prinzipielle Aufbau einer sondierenden Methode in JAVA?

6) Wie lautet der prinzipielle Aufbau einer verändernden Methode in JAVA?

7) Wie realisiert man Vererbung in JAVA? Was bewirkt Vererbung?

8) Was versteht man unter dem Kapselungs-Prinzip?

Zusatzmaterial: Test 2

Zwischenübungen zur Festigung

22. Klasse ZYLINDER als Übung

- *Lege in BlueJ ein neues Projekt Zwischenuebung an.*
- *Erzeuge darin eine Klasse ZYLINDER nach folgender Vorlage und denke dabei an das Kapselungs-Prinzip:*

ZYLINDER
radius : double *hoehe :* double *farbe :* String
nenneRadius() : double *nenneHoehe() :* double *nenneFarbe() :* String *setzeRadius(neuerRadius:* double*) :* void *setzeHoehe(neueHoehe:* double*) :* void *setzeFarbe(neueFarbe:* String*) :* void *berechneVolumen() :* double

23. *Übung zur Vererbung*

Setzen Sie zusätzlich folgende Situation im Projekt Zwischenübung um:

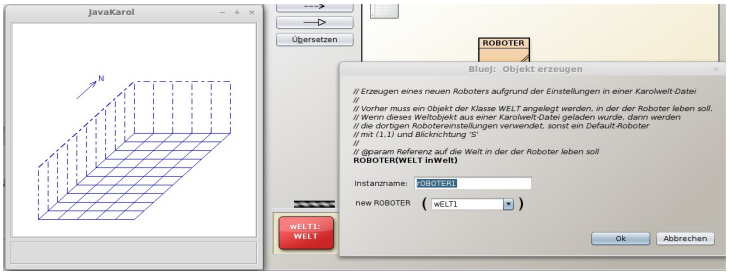
PERSON
name : String *alter :* int
nenneName() : String *nenneAlter() :* int *setzeName(neuerName:* String*) :* void *setzeAlter(neuesAlter:* int*) :* void

SCHUELER
Klasse : int *Abteilung :* char
nenneKlasse() : int *nenneAbteilung() :* char *setzeKlasse(neueKlasse:* int*) :* void *setzeAbteilung(neueAbteilung:* char*) :* void

24. JAVA-Karol – Vorbereitung

a) *Öffne das BlueJ-Projekt JavaKarol.*

- *Erzeuge ein neues WELT-Objekt mit dem Standard-Konstruktor (ohne Parameter). Suche im erscheinenden Datei-Browser den Ordner dieses BlueJ-Projekts und wähle darin die Datei welt-leer.kdw.*
- *Schließe nun das Welt-Fenster wieder.*
- *Wenn du weißt, dass sich die benötigte Welt-Datei im Projekt befindet, dann bietet sich der zweite Konstruktor (ein Parameter) an.*
  Rufe ihn nun auf und schreibe in die runden Klammern des sich öffnenden Fensters `"welt-leer.kdw"` (incl. Anführungszeichen).
  Warum braucht man hier Anführungszeichen?
- *Erzeuge nun ein ROBOTER-Objekt mit dem zweiten Konstruktor (nur ein Parameter) und übergib dabei das WELT-Objekt indem du seinen Namen in die runden Klammern des erscheinenden Fensters schreibst.*



Warum braucht man hier keine Anführungszeichen?

- *Rufe nun einige Methoden deines ROBOTER-Objekts durch Rechtsklick auf.*
- *Schließe das Welt-Fenster wieder.*

b) *Sieh dir nun den folgenden Film an.* *http://www.youtube.com/watch?v=W16l3bkZu5w&feature=relmfu*

Damit dein Roboter nun ohne deine Klicks etwas erledigen kann, braucht er eine von dir programmierte Robotersteuerung, die

- *den Roboter kennt und steuert*
- *die Welt kennt, in welcher der Roboter arbeiten soll*

Speichere Dein Projekt unter dem Namen JavaKarolTest und schreibe anschließend eine eigene Klasse ROBOTERSTEUERUNG.
Sie hat zwei Attribute:

- *ein Objekt vom Typ WELT mit dem Namen `welt`*

```
private WELT welt;
```

- *ein Objekt vom Typ ROBOTER mit dem Namen `karol`*

```
private ROBOTER karol;
```

Im Konstruktor wird erst das neue WELT-Objekt erzeugt:

```
this.welt = new WELT("welt-leer.kdw");
```

Anschließend wird im Konstruktor das ROBOTER-Objekt erzeugt:

```
this.karol = new ROBOTER(this.welt);
```

In einer Methode kannst du Karol nun Anweisungen geben:

```
public void tuWas() {
   karol.hinlegen();
   karol.schritt();
}
```

c) Zeichne eine Klassen-Karte und ein Klassen-Diagramm zu diesem Szenario.

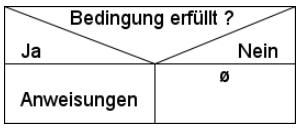
Bedingte Anweisungen

Fallunterscheidungen solltest du bereits aus der 7. Klasse NuT-Informatik kennen.
Hier eine Erinnerung:

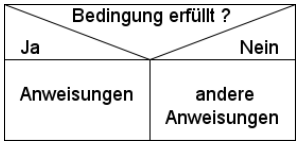
Es gibt einfache und mehrfache bedingte Anweisungen. Bei mehrfachen bedingten Anweisungen spricht man auch von Fallunterscheidungen.

Die hier dargestellten grafischen Darstellungen nennt man ***Struktogramm***.

- Die ***einfache bedingte Anweisung*** hat die Gestalt



- Die ***bedingte Anweisung mit Alternative*** hat die Gestalt



- Die ***mehrfache Fallunterscheidung*** hat die Gestalt

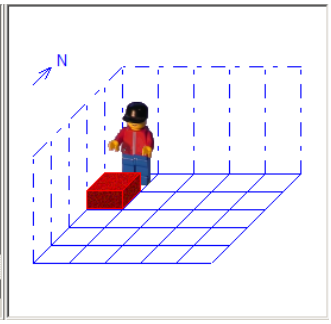
25. **Robot Karol – Erinnerungen 1**

a) *Öffne die Umgebung „Robot Karol“ und lade die Welt* welt-01.kdw .

b) *Schreibe folgendes Programm:*

```
wenn istZiegel
dann aufheben
*wenn
schritt

wenn istZiegel
dann aufheben
*wenn
schritt
```



c) *Führe das Programm durch Klick auf die Start-Taste aus.*

d) *Zeichne ein Struktogramm zu diesem Programm.*

e) *Welche Befehle sind sondierende Methoden?*

f) *Welche Aufrufe sind verändernde Methoden?*

g) *Wie müssten all diese Methodenaufrufe aus Objekt-orientierter Sicht in Punktnotation lauten? Schreibe die Antwort in deinem Hefter auf.*

- ***Bedingte Anweisung*** in JAVA:

```
if  ( Bedingung ) {
    Anweisung;
}
```

  - Die Bedingung muss in runden Klammern stehen.
  - Es gibt kein Schlüsselwort für „dann“.
  - Die Anweisung muss in geschweiften Klammern stehen.

- ***Bedingte Anweisung mit Alternative*** in JAVA:

```
if  ( Bedingung )  {
    Anweisung-1;
}
else {
    Anweisung-2;
}
```

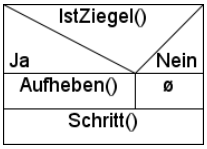
- ***Mehrfache Fallunterscheidung*** in JAVA:

```
if  ( Bedingung-1 )  {
    Anweisung-1;
}
else if  ( Bedingung-2 )  {
    Anweisung-2;
}
else if  ( Bedingung-3 )  {
    Anweisung-3;
}
```

- ***Mehrfache Fallunterscheidung mit Alternative*** in JAVA:

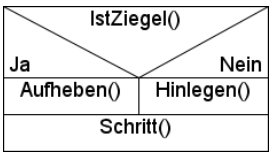
```
if  ( Bedingung-1 )  {
    Anweisung-1;
}
else if  ( Bedingung-2 )  {
    Anweisung-2;
}
else if  ( Bedingung-3 )  {
    Anweisung-3;
}
else {
    Anweisung-4;
}
```

Beispiel 1:




```
wenn istZiegel          if  ( this.Karol.IstZiegel() )  {
   dann aufheben           this.Karol.Aufheben();
*wenn                    }
schritt                  this.Karol.Schritt();
```

Beispiel 2:

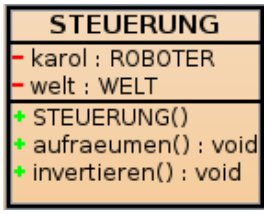



```
wenn istZiegel()        if  ( this.Karol.IstZiegel() )
                        {
   dann aufheben()         this.Karol.Aufheben();
                        }
                        else
                        {
   sonst hinlegen()        this.Karol.Hinlegen();
*wenn                   }
schritt()               this.Karol.Schritt();
```

26. JAVA-Karol – bedingte Anweisungen

Öffne das BlueJ-Projekt JavaKarol und speichere es sofort unter dem Namen JavaKarol_Bedingungen.

a) *Erstelle eine Klasse ROBOTERSTEUERUNG nach folgenden Vorgaben:*



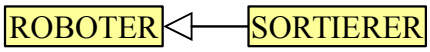
Als Welt wählst du die Datei welt-01.kdw .

- *Der Rumpf der Methode `aufraeumen()` entspricht dabei dem letzten Beispiel 1.*
- *Teste die Methode, indem du ein Objekt deiner ROBOTERSTEUERUNG erzeugst und die Methode durch mehrmaligen Rechtsklick aufrufst.*
- *Der Rumpf der Methode `invertieren()` entspricht dabei dem letzten Beispiel 2.*
- *Teste die Methode, indem du ein Objekt deiner ROBOTERSTEUERUNG erzeugst und die Methode durch mehrmaligen Rechtsklick aufrufst.*

b) *Schreibe eine weitere Methode `bedingterSchritt()`, welche Karol einen Schritt tun lässt, wenn keine Wand vor ihm ist.*
Teste auch diese Methode durch mehrmaligen Aufruf mittels Rechtsklick auf ein davor erzeugtes Objekt deiner ROBOTERSTEUERUNG.

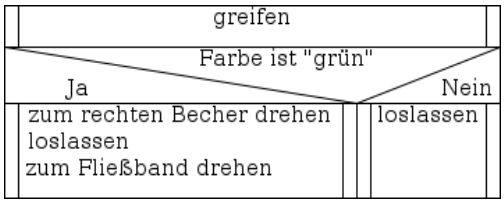
27. Greif-Roboter – bedingte Anweisungen 1

Öffne das BlueJ-Projekt GreifRoboter und speichere es unter dem Namen GreifRoboter_Bedingungen neu ab.



a) *Erstelle eine Klasse SORTIERER, welche von der Klasse ROBOTER erbt.*
Denke an einen Konstruktor, der den Superkonstruktor aufruft.

Schreibe darin eine Methode `sortiereGruen()`*, welche die nächste Kugel greift, und einfach fallen lässt, wenn sie nicht grün ist. Grüne Kugeln werden in den rechten Becher gelegt und der Arm wieder zum Fließband zurück bewegt.*



Tipp: *Hierzu brauchst du den Vergleichs-Operator ==.*

```
if ( this.kugelfarbeGeben() == "grün" ) …
```

b) *Schreibe eine weitere Methode* `sortiere(String farbe)`*, welche sich genauso verhält, aber die gewünschte Farbe als Übergabe-Parameter entgegen nimmt.*

28. Greif-Roboter – bedingte Anweisungen 2

Öffne das BlueJ-Projekt GreifRoboter_Bedingungen.

a) *Schreibe eine Methode* `nimmDrei()`*, welche eine Kugel vom Fließband nimmt und in den linken Becher wirft, wenn die Nummer nicht 3 ist. Wenn die Nummer 3 ist, dann wird die Kugel in den rechten Becher geworfen. Anschließend – egal, welcher Fall eingetreten ist – soll der Arm wieder zum Fließband bewegt werden.*

> ***TIPP:*** *Falls dir das Zurückbewegen des Arms auf diese Art nicht gelingt, dann bewege den Arm in jedem einzelnen Fall einfach gleich wieder zurück.*

Rufe die Methode mindestens vier Mal auf um sicherzustellen, dass sie richtig funktioniert.

b) *Zeichne ein Struktogramm zur Methode aus a).*

c) *Schreibe eine weitere Methode* `nimm(int nummer)`*, welche sich genauso verhält, aber die gewünschte Nummer als Übergabe-Paramter entgegen nimmt.*

Rufe auch diese Methode entsprechend oft auf.

d) *Schreibe eine weitere Methode* `nimmKleiner(int nummer)`*, welche alle Kugeln, deren Nummer kleiner als der Wert des Übergabe-Parameters sind, in den linken Becher wirft, bei größeren Nummern in den rechten Becher.*

Rufe auch diese Methode entsprechend oft auf.

e) *Zeichne eine Klassen-Karte der Klasse SORTIERER.*

29. Engine-Alpha – animierter Ball – Teil 1

Öffne das BlueJ-Projekt Ball-Animation_Vorlage und speichere es unter dem Namen Ball-Animation.

a) *Schreibe eine Klasse BALL die von KREIS erbt.*
*Welche **Attribute und Methoden** hat der BALL **geerbt**?*

b) *Die Klasse BALL soll zwei weitere ganzzahlige Attribute `deltaX` und `deltaY` haben.*
Programmiere außerdem einen Konstruktor, der den Ball gestaltet wie in der Objekt-Karte dargestellt.

ball : BALL	
deltaX	= *-10*
deltaY	= *-5*
farbe	= *"gelb"*
radius	= *15*
M_x	= *30*
M_y	= *20*

Tipp:
- ***Erstelle keine Attribute, die du schon geerbt hast.***
- ***Verwende die setze-Methoden** um Werte geerbter Attribute zu setzen.*
- ***Sprich die neuen Attribute mit `this` an** und verwende wie gewohnt die Wert-Zuweisung, um die Werte der neuen Attribute zu setzen.*
- ***Verwende bitte exakt die abgebildeten Werte!***

c) *Erstelle außerdem eine Methode `bewegen()`, welche in ihrem Rumpf die geerbte Methode `verschiebenUm(… , …)` aufruft.*
Als Parameter für diese Methode wählst du die Attribut-Werte von `deltaX` und `deltaY`. Hierdurch wird der Ball horizontal um den Wert von `deltaX` und vertikal um den Wert von `deltaY` verschoben.
Erzeuge ein Objekt und teste die Methode `bewegen()`.
Verschiebt sich dein Ball-Objekt am Bildschirm?
Wenn nicht, dann überprüfe die Attribut-Werte von `deltaX` und `deltaY` im Objekt-Inspektor. Sind sie verschieden von Null?

d) *Der Ball soll an den Fenster-Rändern reflektiert werden.*

Das Grafik-Fenster ist 800 Pixel breit und 600 Pixel hoch.
Suche nach einfach zu formulierenden Bedingungen, wann der Ball einen Bildschirmrand erreicht hat. Betrachte hierzu die Werte der geerbten Attribute `M_x` und `M_y`, das sind die Koordinaten des Mittelpunkts deines Balls (in Pixeln).

Reflexion am linken Rand, wenn ...
Reflexion am rechten Rand, wenn ...
Reflexion am oberen Rand, wenn ...
Reflexion am unteren Rand, wenn ...

Formuliere diese Bedingungen auf Papier in JAVA mithilfe einer vierfachen Fallunterscheidung. *(noch bleiben die geschweiften Klammern leer)*

Welche Anweisung muss in jedem der vier Fälle gegeben werden?
Betrachte hierzu die Schrittweiten `deltaX` und `deltaY`, welche du der Methode `verschiebenUm(… , …)` übergeben hast. Fertige je eine Zeichnung mit einem Ball kurz vor der Reflexion am rechten Rand bzw. kurz nach der Reflexion am rechten Rand und betrachte jeweils den Wert von `deltaX` und `deltaY` in beiden Situationen. Zeichne hierzu `deltaX` und `deltaY` jeweils als Pfeil ein. Welcher Wert bleibt vor und nach der Reflexion gleich, welcher verändert sich? Wie verändert sich der Wert genau?

Anweisung bei Reflexion links: …
Anweisung bei Reflexion rechts: …
Anweisung bei Reflexion oben: …
Anweisung bei Reflexion unten: …

Setze deine Ideen im Projekt um und teste, ob der Ball nun reflektiert!

Vererbung 2 – Überschreiben von Methoden

Erbt eine Klasse von einer anderen Klasse, so erbt sie auch alle Methoden, die mit dem Modifikator `public` deklariert wurden. Du kannst diese Methoden dann in deiner Klasse aufrufen und ihr Code wird unverändert ausgeführt. Manchmal möchte man aber das **Verhalten einer geerbten Methode verändern**.

Die Klasse *SPIEL* stellt dir z.B. die Methode `tick()` zur Verfügung. Diese Methode wird nach einem bestimmten Zeitintervall automatisch immer wieder ausgeführt. Standardmäßig schreibt sie *tick … tack …* in ein Fenster. Wenn du nun möchtest, dass irgend eine Aktion immer wieder in regelmäßigen Zeitabständen aufgerufen wird, dann musst du nur von SPIEL erben, denn `tick()` wird ja automatisch aufgerufen. Aber du willst kein *tick … tack …* in einem Fenster geschrieben haben. Du möchtest z.B. deinen Ball immer wieder automatisch bewegen.

Dann musst du die **geerbte Methode** nur **überschreiben**.

So kann man in einer Superklasse ein Feature anbieten (**z.B. automatisch aufgerufen zu werden** und dennoch dem Erzeuger der Unterklasse die Freiheit lassen, was nun automatisch ausgeführt werden soll. Ganz genauso verhält es sich auch bei **Tastatur-Ereignissen** oder **Mausklicks**. Auch hier wir die Technik allgemein vererbt, aber dem Erzeuger einer Subklasse die Freiheit gelassen, was genau beim Drücken einer Taste auf der Tastatur oder Maus geschehen soll.

Möchte man das Verhalten einer geerbten Methode nachträglich ändern, so muss man die ***geerbte Methode*** in der Subklasse ***überschreiben***.

Das Überschreiben einer geerbten Methode eröffnet man mit dem Statement `@Override`. In der nächsten Zeile schreibt man dann den Kopf der geerbten Methode ganz genau so, wie er schon in der Superklasse geschrieben steht. Alles, was man nun an Code innerhalb der geschweiften Klammern dieser Methode schreibt, wird in Zukunft anstatt des geerbten Codes ausgeführt.

Beispiel:

```
@Override
public void tick() {
    this.ball.bewegen();
}
```

Von nun an wird der Ball in regelmäßigen Abständen bewegt anstatt tick … tack … in einem Fenster zu schreiben.

30. Engine-Alpha – Die Klasse SPIEL – die Methode `tick()`

Öffne nun das BlueJ-Projekt Spiel-Superklasse.

Erzeuge durch Rechtsklick ein Objekt der Klasse SPIEL.
Beobachte insbesondere das weiße Fenster. Was du siehst ist die Ausgabe der Methode `tick()`. Sie wird automatisch in regelmäßigen Zeitabständen aufgerufen. Was sie tut, entscheidest du später selbst.
Rufe bei deinem SPIEL-Objekt die Methode `tickerIntervallSetzen(...)` auf. Der Übergebene Wert ist die Zeit in Millisekunden zwischen den erneuten Aufrufen der Methode `tick()`. Der Wert 500 bedeutet also 0,5 Sekunden.

Wenn du später von der Klasse SPIEL erbst, dann musst du nur die Methode `tick()` in deiner eigenen Klasse neu schreiben. In ihrem Rumpf entscheidest du dann, was bei einem Tick passiert, z.B. könnte sich der Ball automatisch weiter bewegen ...

Bevor du deinen Ball automatisch bewegen kannst, musst du noch eine Kleinigkeit über Referenz-Attribute lernen:

Referenz-Attribute

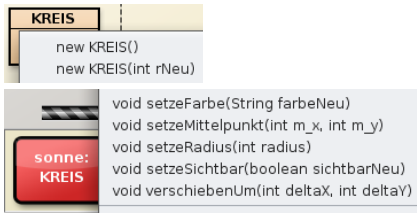
Bisher waren die **Attribute** unserer Klassen immer **von recht einfacher Natur**. Es handelte sich um Zahlen, Texte, … Man konnte ihnen **mit einem Gleichheitszeichen einen Wert zuweisen**, z.B.

```
this.alter = 12;
this.name = „Hans“;
```

Bei deinem ersten BlueJ-Projekt *alphaFormen_Zeichnung* hattest du aber die drei Attribute `wand`, `dach` und `sonne`. Dies waren eigenständige Objekte, nicht einfach nur eine Zahl oder ein Text. Bevor du diese Objekte durch JAVA-Code in der Klasse Zeichnung verändert hast, hast du mit der Maus auf den roten Objekt-Karten Methoden aufgerufen. Erinnerst du dich noch? Du musstest mit der Maus **erst ein neues** KREIS-**Objekt erzeugen** und es sonne nennen. Danach hast du Methoden dieses Objekts aufgerufen, um es zu gestalten.

```
this.sonne = new KREIS(50);

this.sonne.setzeFarbe(''gelb'');
```



- Ein Attribut, dessen Typ eine Klasse ist und das somit ein eigenständiges Objekt darstellt, nennt man ***Referenz-Attribut***.

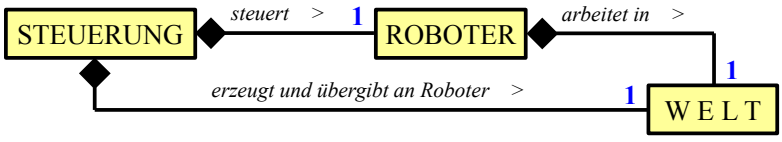
```
private KREIS sonne;
```

- Da Referenz-Attribute eigenständige Objekte sind, müssen sie nach dem Deklarieren erst einmal als neues Objekt erzeugt werden. Hierzu verwendet man den `new`-Operator um das Objekt zu erzeugen und gleichzeitig das Gleichheitszeichen, um eine Referenz auf das Objekt als Wert des Attributs zu speichern.

```
this.sonne = new KREIS(50);
this.sonne.setzeFarbe(''gelb'');
```

- Wenn ein großes Objekt aus kleineren Teil-Objekten zusammen gesetzt ist, so spricht man auch von ***Aggregation***. Aggregate sind also zusammengesetzte Objekte. In der Klasse des großen Objekts findet man dann die kleineren Bauteile als Referenz-Attribute wieder.

- **Beispiel** unserer bisherigen Aggregationen als Klassen-Diagramme:



- Das **Symbol** für ein Referenz-Attribut **im Klassen-Diagramm** ist eine Linie mit einer Raute an einem Ende.
  Die Raute liegt am der Klasse des Aggregats (des großen Objekts) an.

31. Engine-Alpha – animierter Ball – Teil 2

Öffne nun wieder das BlueJ-Projekt Ball-Animation.

- *Klicke im Hauptmenü auf Bearbeiten / Klasse aus Datei hinzufügen und wähle aus dem Projekt edu-Klassen-master (alle EDU-Klassen als Vorlage) die Klasse SPIEL.*

- *Schreibe nun eine Klasse BILLARD, welche von der Klasse SPIEL erbt. Vergiss nicht, einen Konstruktor zu schreiben in dem ein geerbter Super-Konstruktor aufgerufen wird …*

- *Deklariere (oberhalb des Konstruktors) und initialisiere (nach dem Aufruf des Super-Konstruktors) in der Klasse BILLARD nun ein Referenz-Attribut der Klasse BALL und nenne es* `ball`*.*
  Erzeuge ein BILLARD-Objekt und prüfe, ob sich der Ball im Fenster befindet.

  Stimmen Radius, Farbe und Lage mit dem Code der Klasse BALL überein?

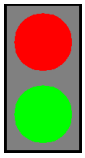
- *Lies dir nun noch einmal schnell den Abschnitt mit dem Überschreiben von geerbten Methoden durch und überschreibe anschließend in der Klasse BILLARD die geerbte Methode* `tick()`*. Rufe in ihrem Rumpf die Methode* `bewegen()` *des BALL-Objekts auf.*
  Erzeuge erneut ein Objekt der Klasse BILLARD. Der Ball sollte sich nun automatisch bewegen und dabei am Bildschirm-Rand reflektieren.

  Verhält sich dein Ball wie gewünscht?
  Behebe ggf. Fehler in der Klasse BALL, falls der Ball sich nicht korrekt verhält.

- ***Zusatz-Aufgabe:***
  Deklariere und initialisiere in deinem Projekt mehrere BALL-Objekte und nenne sie ball_1, ball_2, … und initialisiere sie an unterschiedlichen Mittelpunkten. Denke auch daran, diese neuen BALL-Objekte in der Methode `tick()` *zu berücksichtigen.*

Zwischenübungen zur Festigung

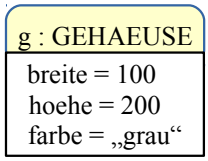
32. **Projekt AMPEL – Teil 1** *Zusatzmaterial:* ergänzende Präsentation



a) Öffne das Projekt *Projekt_AMPEL_Vorlage* und speichere es unter dem Namen *Projekt_AMPPEL_1*.

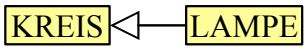
b) *Erstelle eine Klasse GEHAEUSE, welche von der Klasse RECHTECK erbt.*
Welche Attribute und Methoden erbst du?

Deine Klasse GEHAEUSE soll der rechts abgebildeten Klassenkarte entsprechen.
Die Übergabe-Parameter des Konstruktors stellen die Koordinaten des Mittelpunkts dar.
Die restlichen Attribut-Werte legst du bitte genau auf die Werte der abgebildeten Objekt-Karte fest.

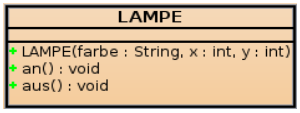


Erzeuge mehrere GEHAEUSE-Objekte mitunterschiedlichen Attribut-Werten.
Liegen sie dort, wo du sie dir vorgestellt hast?

c) *Erstelle eine weitere Klasse LAMPE, die von KREIS erbt.*
Welche Attribute und Methoden erbst du?



Die Klasse LAMPE soll der Klassen-Karte entsprechen.

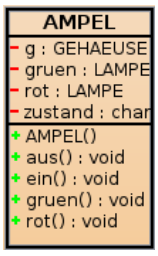


Die Übergabe-Parameter des Konstruktors legen die Farbe und die Koordinaten des Mittelpunkts fest.
Vergiss nicht, sie in geerbten Methoden zu verwenden.
Der Radius einer neuen Lampe soll den Wert 40 haben.

Die Methoden `an()` und `aus()` setzen die Sichtbarkeit der Lampe auf `true` bzw. `false`. *(siehe geerbte Methoden)*

Erzeuge mehrere LAMPE-Objekte mit unterschiedlichen Attribut-Werten.
Sehen sie aus, wie du es dir vorgestellt hast?

d) *Erstelle noch eine Klasse AMPEL.*
Die Klasse AMPEL soll der Klassen-Karte rechts entsprechen.
Im Konstruktor soll eine neue Ampel genau in der Mitte des Grafik-Fensters stehen und den Zustand 'e' haben.



Schreibe auch die vier Methoden:
`aus()` *: beide Lampen aus, Zustand 'a'*
`ein()` *: beide Lampen an, Zustand 'e'*
`gruen()` *: grüne Lampe an, rote Lampe aus, Zustand 'g'*
`rot()` *: rote Lampe an, grüne Lampe aus, Zustand 'r'*

Erzeuge ein AMPEL-Objekt und teste die vier neuen Methoden.
Funktioniert alles, wie es soll?

e) ***Zeichne ein Klassen-Diagramm des Ampel-Projekts.***

33. Engine-Alpha – Ampel Teil 2

Nun soll deine Ampel aus der letzten Aufgabe auf Tastatur-Ereignisse reagieren.

Bei Druck auf Taste ***g*** soll die Methode ***`gruen()`*** und bei Druck auf Taste ***r*** die Methode ***`rot()`*** aufgerufen werden. Zusätzlich soll mit Druck auf Taste ***e*** die Ampel eingeschaltet werden und automatisch weiter schalten. Die Taste ***a*** soll die Automatik und die Ampel wieder ausschalten.

a) *Öffne dein bisheriges Ampel-Projekt und deklariere eine neue ganzzahlige Variable mit dem Namen `millisekunden`. Initialisiere diese Variable im Konstruktor auf den Wert 1000.*
Erzeuge ein Objekt der Klasse SPIEL und drücke auf die Tasten g, r, e, a. Beobachte dabei die Ausgabe im Konsolen-Fenster.
Merke dir die Nummern der Tasten, du wirst sie gleich brauchen.

b) *Lasse nun die Klasse AMPEL von der Klasse SPIEL erben.*
Denke auch daran, einen Super-Konstruktor aufzurufen.
Welche Attribute und Methoden hast du damit geerbt?

c) *Nun wollen wir die geerbte Methode **`tasteReagieren(int code)`** überschreiben.*
Was bedeutet Überschreiben einer Methode und wie macht man das in JAVA?
Überschreibe die Methode mit einem noch leeren Rumpf.

*Die Methode **`tasteReagieren(int code)`** funktioniert folgendermaßen:*

- *Sie wird immer automatisch aufgerufen, sobald eine Taste gedrückt wird.*
- *Jede gedrückte Taste übergibt ihren persönlichen Code automatisch an diese Methode:*
  - *a ruft also automatisch `tasteReagieren(0)` auf,*
  - *b ruft automatisch `tasteReagieren(1)` auf,*
  - *c ruft automatisch `tastereagieren(2)` auf, `...`*
- *Mithilfe einer (mehrfachen) Fallunterscheidung reagieren wir im Rumpf der Methode auf die unterschiedlichen Werte von `code`:*
  - *WENN Taste a gedrückt wurde: `if (code == 0)` …*
  - *SONST WENN Taste e Gedrückt wurde: `else if (code == 4)` …*

Wie sieht eine mehrfache Fallunterscheidung in JAVA aus? *(Skript)*

d) *Sorge nun dafür, dass bei Druck auf Taste **g** die Methode **`gruen()`** und bei Druck auf Taste **r** die Methode **`rot()`** aufgerufen wird.*
*Analog reagierst du nun auf die Taste **a** und **e** mit den Methoden **`aus()`** und **`ein()`**.*

Teste dein Ampel-Projekt, indem du ein Ampel-Objekt erzeugst und anschließend die Tasten a, e, r, g, drückst.
Schaltet die Ampel wie gewünscht?

Erinnere dich nochmal an die Funktionsweise der geerbten Methode **`tick()`**.
Wie ändert man die Abstände zwischen den Ticks?
Wie startet und stoppt man das Ticker-System? *(siehe Klassenkarte von SPIEL)*

e) *Ergänze nun die Methode* `aus()` *um den Befehl zum Stoppen des Tickers. Die Methode* `ein()` *soll den Ticker neu starten. Als Parameter übergibst du den Wert des Attributs* `millisekunden`.

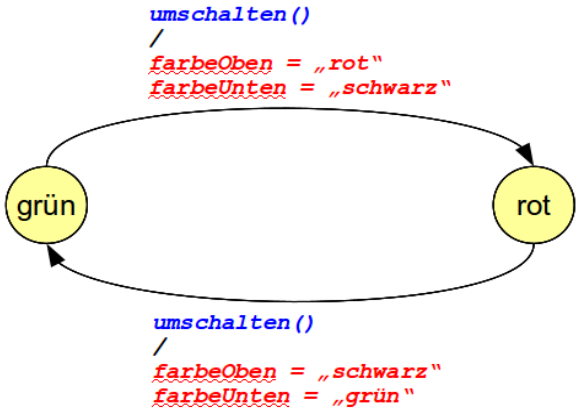
f) *Überschreibe nun auch die Methode* `tick()`. *In ihrem Rumpf soll eine zweifache Fallunterscheidung dafür sorgen, dass die Ampel auf rot geschaltet wird, wenn ihr Attribut zustand den Wert* `grün` *hat und umgekehrt. Schalte die Ampel um, indem du die entsprechende Methode aufrufst.*

Teste erneut dein Ampel-Projekt, indem du ein Ampel-Objekt erzeugst und anschließend die Tasten a, e, r, g, drückst.
Schaltet die Ampel nun auch automatisch?

verändernde Methoden als Zustands-Übergänge von Objekten

Durch die exakte Angabe der Werte von allen Attributen eines Objekts ist ein klar definierter ***Zustand*** dieses Objekts definiert. Ändert man einen oder mehrere Attribut-Werte, so befindet sich das Objekt in einem anderen Zustand. Man sagt auch, das Objekt ist von einem Zustand in einen anderen übergegangen. Solche ***Zustands-Übergänge*** werden in der Regel durch Aufrufe von Methoden des Objekts ausgelöst.

Wenn es keine Zweifel gibt, von welchem Objekt man spricht, so kann man die Darstellungsform vereinfachen. Ein Zustand wird dann nur durch einen Kreis dargestellt, der einen treffenden Namen trägt. Bei den Übergängen werden manchmal zu den auslösenden Ereignissen (z.B. Methodenaufrufen) auch noch die ausgelösten Aktionen (z.B. Änderung der Attribut-Werte) angegeben. Man spricht dann von einem ***Zustands-Übergangs-Diagramm***.



Zustands-Übergangs-Diagramm einer Fußgänger-Ampel

Zustands-Übergangs-Diagramme werden oft verwendet, um kompliziertes Verhalten eines Objekts zu modellieren, bevor man dies programmieren kann. Wir werden dies später bei der Bewegung der Schläger und des Balls sehr ausgiebig verwenden. An dieser Stelle sollst du nur das Prinzip verstehen.

- Ein ***Zustand*** beschreibt einen exakt festgelegten Satz von Attribut-Werten eines Objekts
- Bei einem ***Zustands-Übergang*** wird mindestens ein Attribut-Wert verändert.
- Zustände und Übergänge werden in einem ***Zustands-Übergangs-Diagramm*** veranschaulicht.
- Zustands-Übergangs-Diagramme dienen der Beschreibung von komplexen Vorgängen.

Dokumentation ist wichtig

Wenn mehrere Personen an einem gemeinsamen Projekt arbeiten, aber auch wenn du alleine über einen längeren Zeitraum an einem Projekt arbeitest, dann sollten alle Programmierer den Code auch ***dokumentieren***. JAVA stellt hierfür sog. ***JAVA-DOC-Kommentare*** zur Verfügung.

Aus diesen Kommentaren kann mit nur einem Knopfdruck eine ***HTML-Seite*** (Web-Seite) erstellt werden, in der du alles nachlesen kannst, was du über eine Klasse, ein bestimmtes Attribut oder eine Methode wissen möchtest.

Method Summary

void	setzeFarbe(java.lang.String farbeNeu) Setzt die Farbe dieses Kreises neu.
void	setzeMittelpunkt(int m_x, int m_y) Setzt den Mittelpunkt dieses Kreises neu.
void	setzeRadius(int radius) Setzt den Radius dieses Kreises neu.
void	setzeSichtbar(boolean sichtbarNeu) Setzt, ob dieser Kreis sichtbar sein soll.
void	verschiebenUm(int deltaX, int deltaY) Verschiebt diesen Kreis um eine Verschiebung - angegeben durch ein "Delta X" und "Delta Y".

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

KREIS

```
public KREIS()
```

Konstruktor der Klasse KREIS. Erstellt einen neuen Kreis.

Method Detail

setzeFarbe

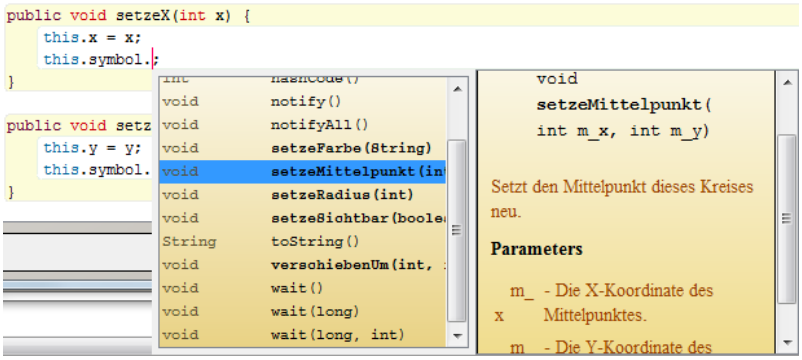
```
public void setzeFarbe(java.lang.String farbeNeu)
```

Setzt die Farbe dieses Kreises neu.

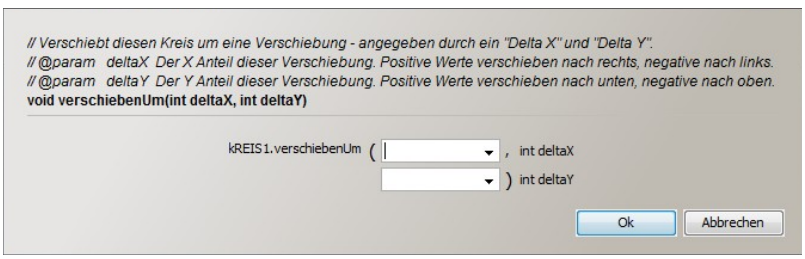
Parameters:

farbeNeu - Diese Farbe erhaelt der Kreis (z.B. "Rot")

Diese Kommentare erscheinen auch, wenn du im Editor von BlueJ mit der Tastenkombination *„Strg + Leertaste“* den ***Code vervollständigen*** möchtest …



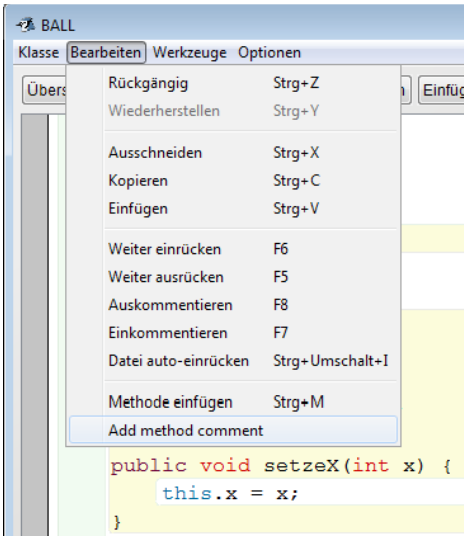
… oder wenn du grafisch die ***Methode eines Objekts aufrufen*** möchtest, mit einem Rechtsklick auf eine Objekt-Karte.



- Stelle im Editor den Cursor vor das `public` einer Methode mit Übergabe-Parameter.

  Wähle nun im Hauptmenü des Editors

  Bearbeiten / Methodenkommentar hinzufügen




```
/**
 * Method setzeX
 *
 * @param x A parameter
 */
public void setzeX(int x) {
    this.x = x;
}
```

- Ersetze den Kommentar-Text

  durch deinen eigenen.

```
/**
 * Methode zum Setzen der x-Koordinate des Balls
 *
 * @param   x   Die neue x-Koordinate des Balls
 */
public void setzeX(int x) {
    this.x = x;
}
```

- Verfahre ebenso mit einer

  sondierenden Methode.

```
/**
 * Methode zum Nennen der x-Koordinate des Balls
 *
 * @return  x   x-Koordinate des Balls
 */
public int nenneX() {
    return this.x;
}
```

- Ein ***JAVA-DOC-Kommentar*** beginnt mit den Zeichen ***/*****
- Jede weitere Zeile muss mit ******* beginnen.
- Die letzte Kommentar-Zeile beginnt mit ****/***
- Nach einem ***@param*** kann man Informationen zu **Übergabe-Parametern** geben:

  Struktur: ***@param parametername Erklärungstext***
- Nach einem ***@return*** kann man Informationen zu **Rückgabe-Werten** geben:

  Struktur: ***@return Erklärungstext***

34. Kommentiere nun alle Methoden in der Klasse AMPEL.

Betrachte die dein Ergebnis, indem du rechts oben im Editor-Fenster auf Dokumentations-Ansicht wechselst.
Benutze die Hyperlinks und lies deine eigene Dokumentation …

Erzeuge Objekte deiner Klassen und rufe durch Rechtsklick auf die Objekt-Karten alle Methoden auf. Beobachte deine eigene Dokumentation und beurteile, ob sie einem Dritten helfen würde, deine Klasse und ihre Methoden zu verstehen.