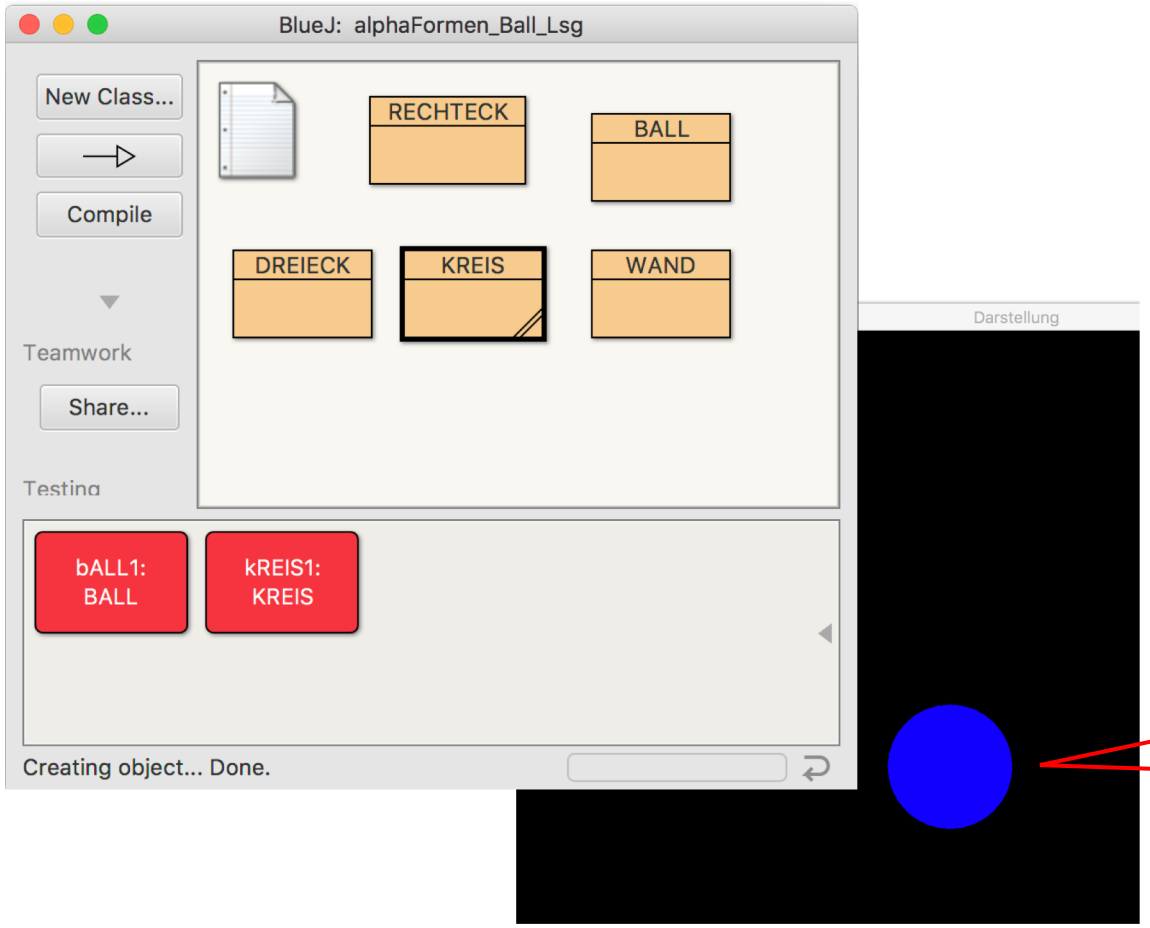
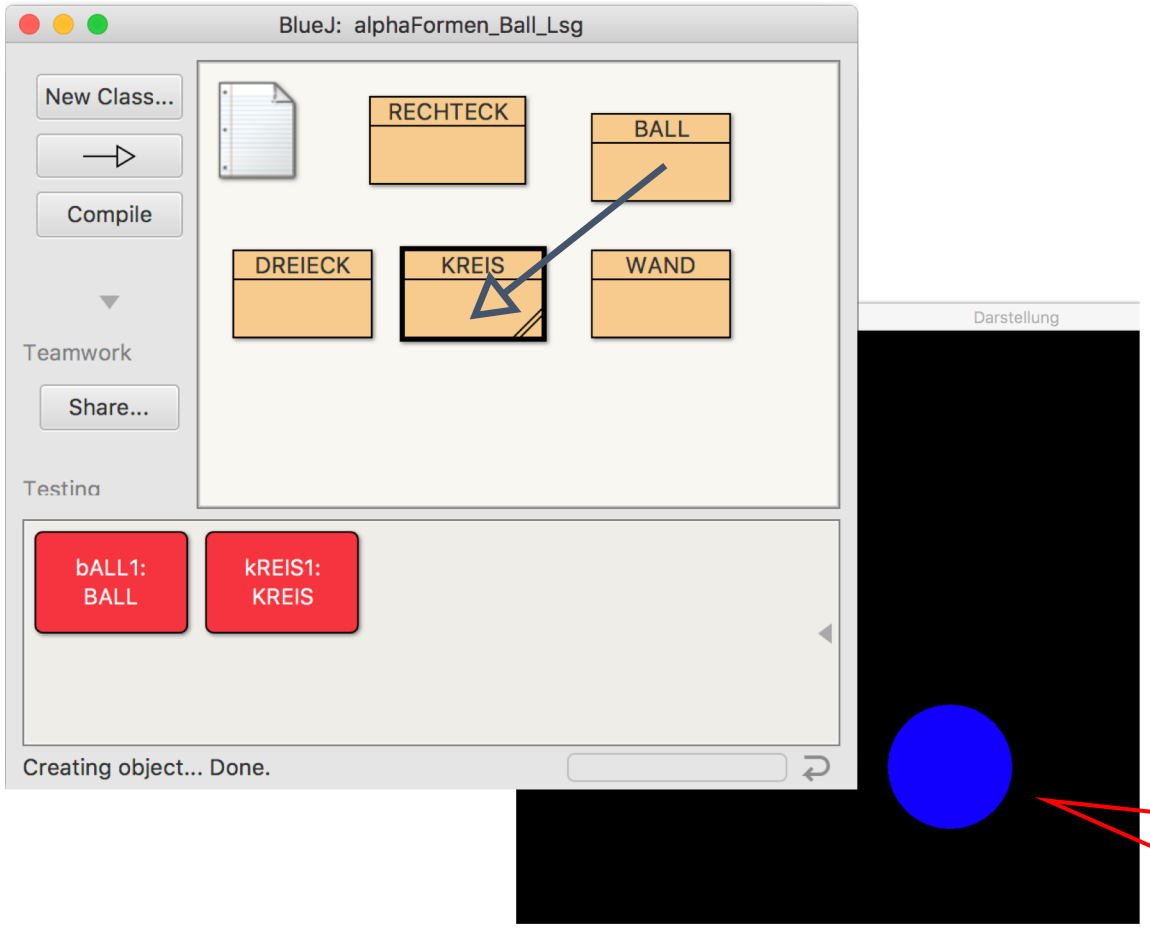
# 2. Vererbung und Kapselung




Die Objekte der Klasse BALL werden im Gegensatz zu den Objekten von KREIS noch nicht graphisch dargestellt.

Um die BALL-Objekte auch graphisch darzustellen zu können, muss BALL die Eigenschaften von KREIS übernehmen.

*Darstellung von kREIS1 der Klasse KREIS*




Dies realisiert man in einer objektorientierten Programmiersprache durch das Prinzip der **Vererbung**.

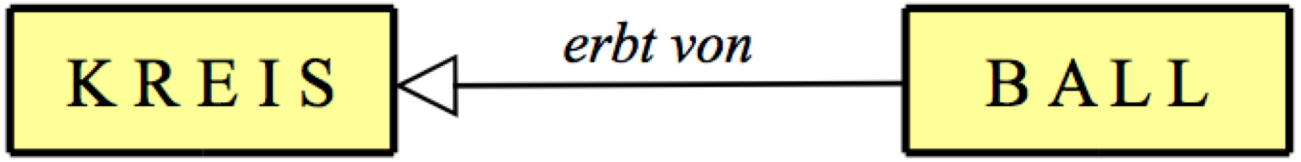
Die Attribute und Methoden von KREIS werden an BALL vererbt, sodass ein BALL-Objekt auch die Methoden von KREIS kennt.

*Darstellung von ball1 der Klasse BALL; kennt auch Attribute und Methoden von KREIS*

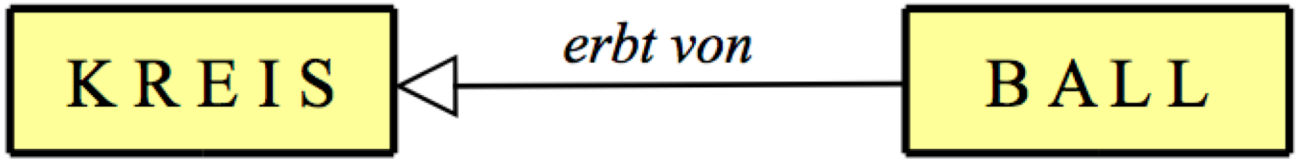
MERKE

**Vererbung** bedeutet, dass eine Klasse Attribute und Methoden an eine andere Klasse weitergeben kann.

Im Klassendiagramm stellt man dies durch einen durchgezogenen Pfeil mit nicht ausgefüllter Spitze dar:



MERKE




Die Klasse, von der geerbt wird (hier: KREIS),
heißt **Superklasse** (Oberklasse).

Die Klasse, die erbt (hier: BALL), nennt man **Subklasse** (Unterklasse).

MERKE

Umsetzung in Java:

```
public class BALL extends KREIS {
```

*Klasse BALL erbt von Klasse KREIS*

```
  String besitzer;
```

*Deklaration eines neuen Attributs*

```
  public BALL() {
    super();
  }
```

*Konstruktor 1 von BALL: Mit super() ruft man den Konstruktor KREIS() der Oberklasse auf.*

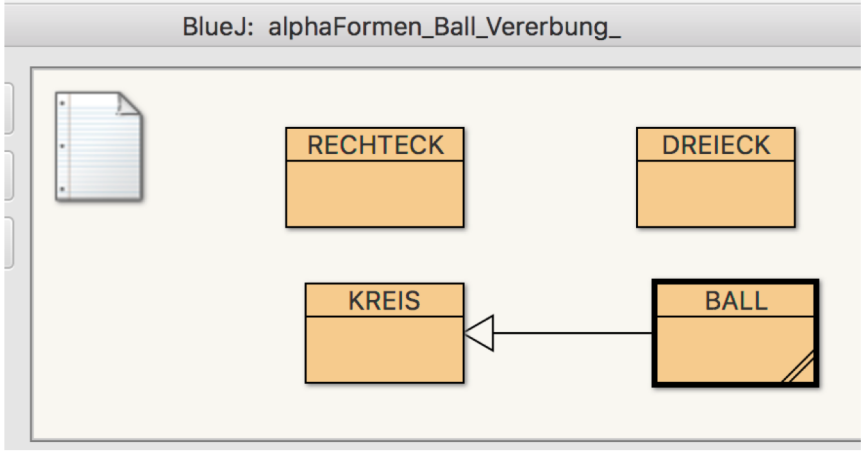
```
  public BALL(int rNeu) {
     super(rNeu);
     this.besitzer = "Hans";
  }
```

*Konstruktor 2 von BALL: Aufruf eines Konstruktors der Oberklasse; Initialisierung des neuen Attributs*

...

# Übung 1 – Klasse BALL

a)
Kopiere das BlueJ-Projekt „alphaFormen“ in deinen Ordner und ändere den Namen in „aphaFormen_Ball_Vererbung“.



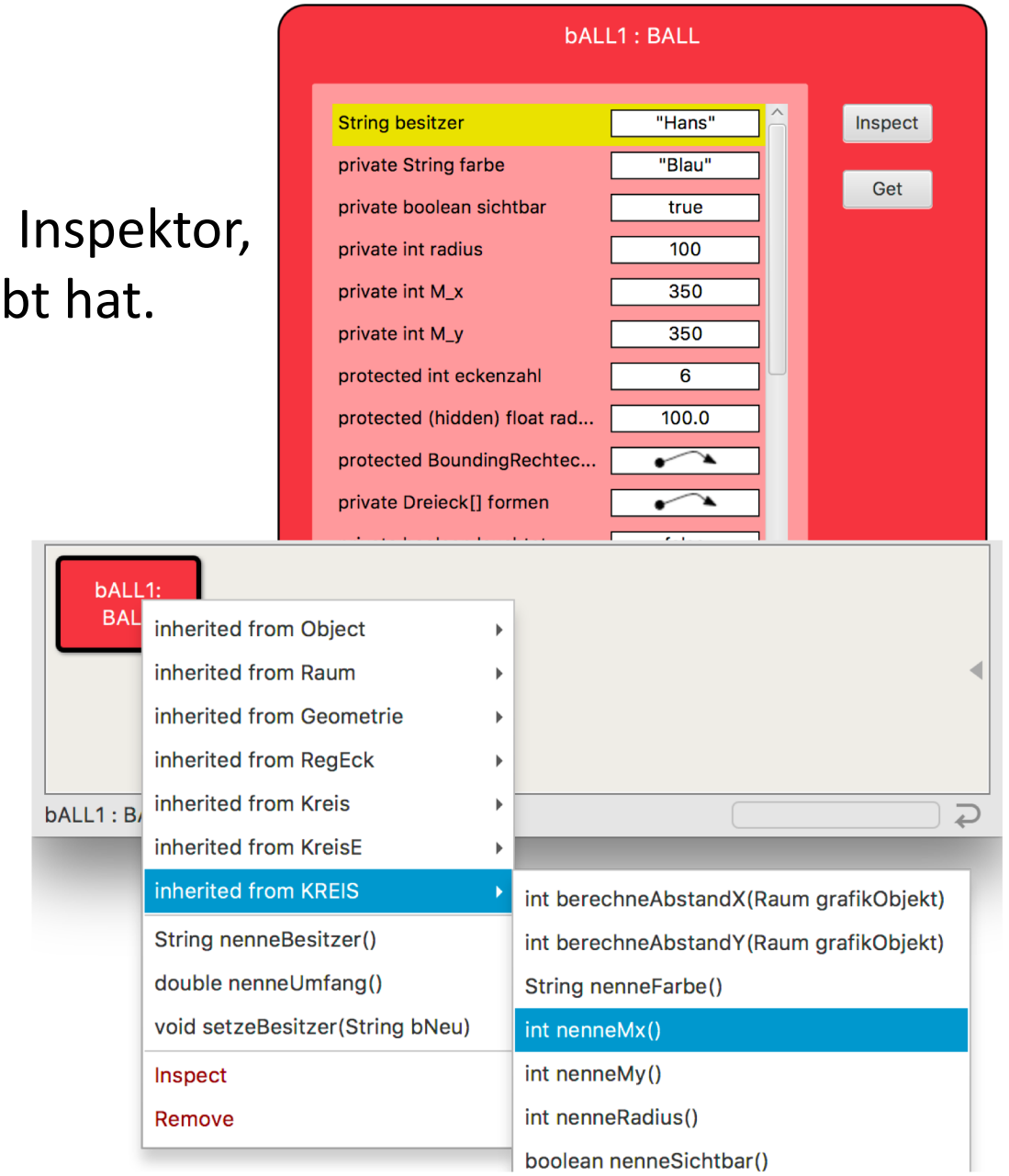

b)
Erzeuge darin eine neue Klasse BALL,
die von KREIS erbt.

c)
Ergänze die Klasse wie oben durch das neue Attribut *besitzer*
und die beiden Konstruktoren.

# Übung 1 – Klasse BALL

d)
Erzeuge ein Objekt von BALL und prüfe im Inspektor, ob es die Attribute von KREIS korrekt geerbt hat.

e)
Klicke mit rechts auf die Objektkarte von BALL und prüfe, ob auch die Methoden von KREIS geerbt wurden.
Rufe einige Methoden auf und teste sie.



## Übung 1 – Klasse BALL

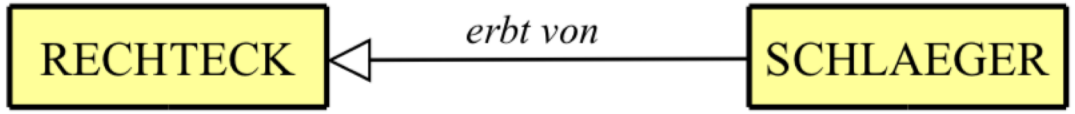
f)*
Ergänze die Klasse BALL um eine Methode *nenneUmfang(),* die den Umfang des Kreises als Zahl vom Typ d*ouble* ausgibt.
Auf den Wert von $\pi$ kann man durch den Befehl *Math.PI* zugreifen.

Erläuterung:

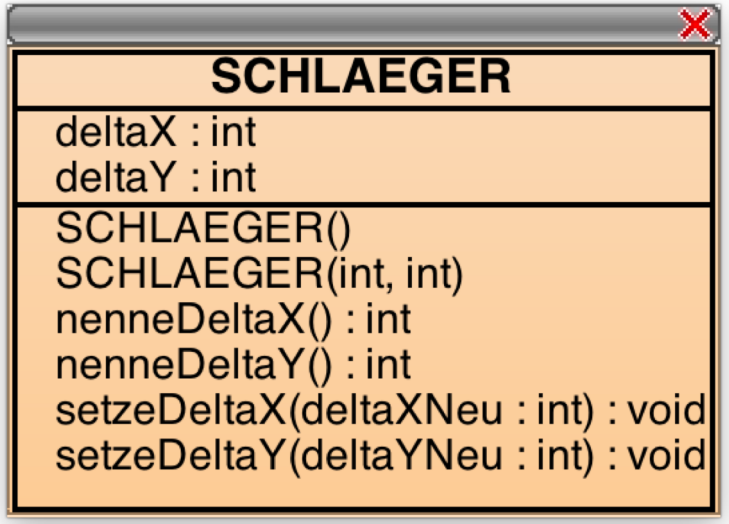
Die Klasse Math ist eine Javaklasse. Auf das Attribut *static double PI* kann man zugreifen, ohne ein Objekt der Klasse Math zu erzeugen. Dies wird durch die Kennzeichnung static gewährleistet.

Hier findest du die offizielle Java-Dokumentation. Suche dort die Klasse Math und https://docs.oracle.com/javase/7/docs/api/

# Übung 2 – Klasse SCHLAEGER




a)
Ergänze das Projekt aus Übung 1
um eine Klasse SCHLAEGER, die von RECHTECK erbt.
Im zweiten Konstruktor setzen die
Übergabeparameter die Werte für
`breite` und `hoehe` fest.




b)
Deklariere zwei weitere ganzzahlige Attribute
`deltaX` und `deltaY`.
(Sie werden später beim PingPong-Spiel benötigt.)
Initialisiere sie mit dem Werten 10 und 5.

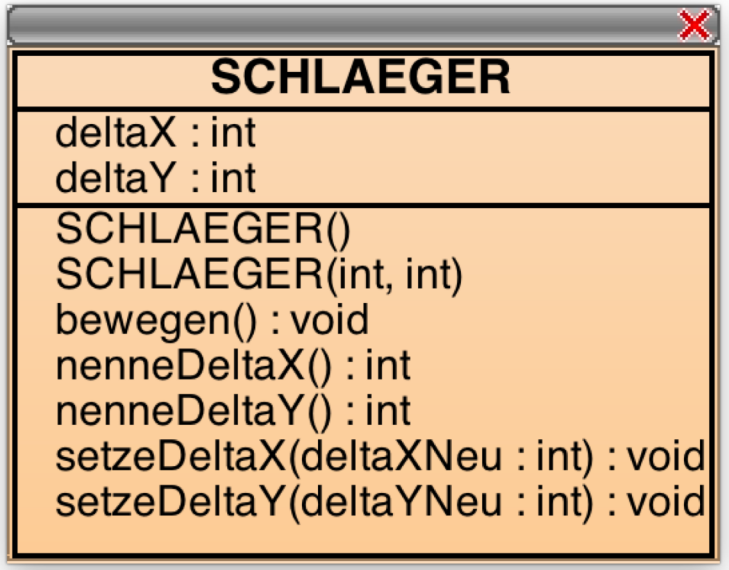
c)
Schreibe zu jedem der beiden Attribute eine sondierende und eine verändernde Methode.

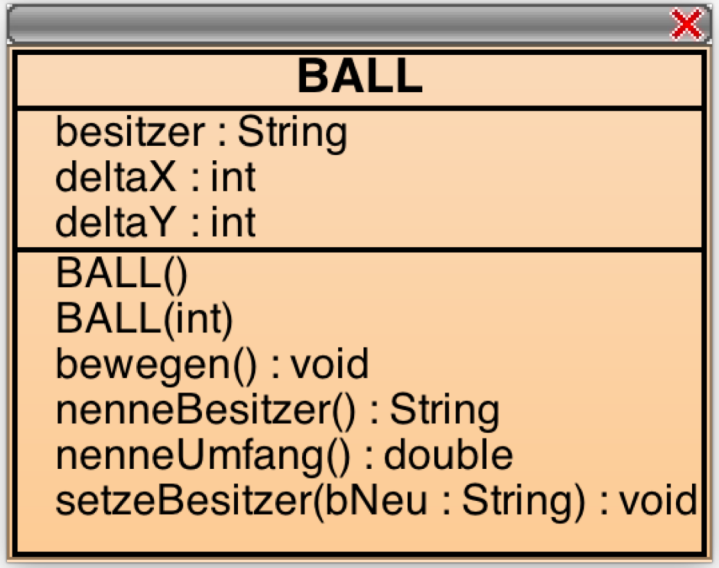
# Übung 2 – Klasse SCHLAEGER

d)*
Ergänze die Klasse SCHLAEGER um eine Methode `bewegen()`,
die bewirkt, dass sich der
Schläger horizontal um `deltaX`
und vertikal um `deltaY` bewegt.

Schreibe eine solche Methode auch für BALL.

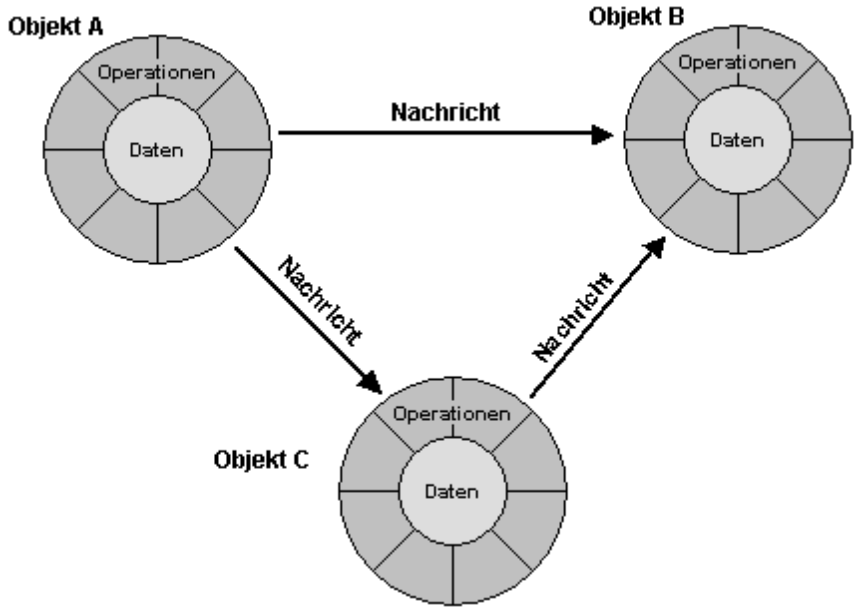
Experimentiere mit verschiedenen Werten für
`deltaX` und `deltaY`.



# Kapselung

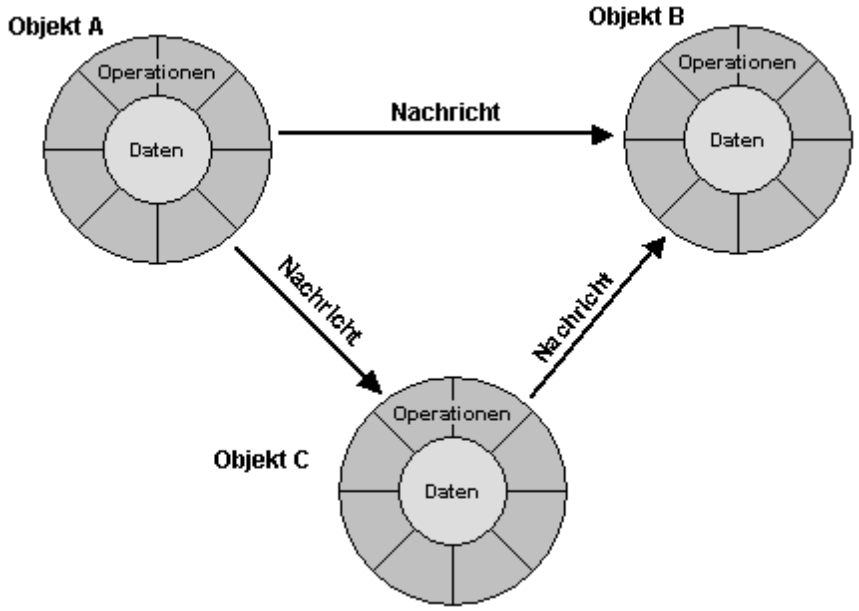
MERKE




Das Arbeiten mit Klassen und Objekten sowie die Vererbung zählen zu den Grundkonzepten einer objektorientierten Programmiersprache.

Ein weiteres wichtiges Konzept ist die
**Kapselung.**

MERKE

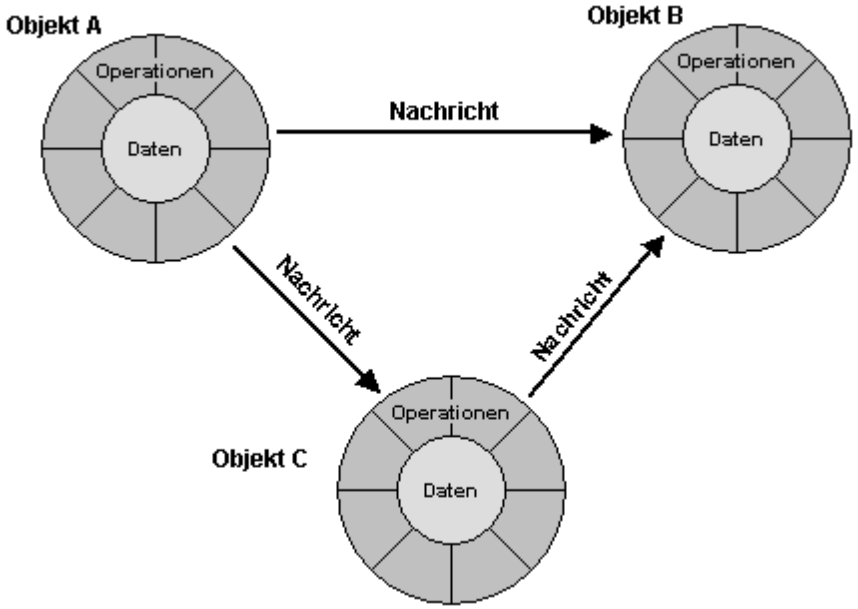



Die Objekte können miteinander kommunizieren, indem sie sich gegenseitig Nachrichten (Botschaften) zuschicken.

Eine Nachricht ist eine Aufforderung an das empfangende Objekt, eine seiner Operationen auszuführen.

Attribute und Methoden können mit **Sichtbarkeitsmodifikatoren** versehen werden, um eine **Zugriffskontrolle** zu realisieren.

MERKE

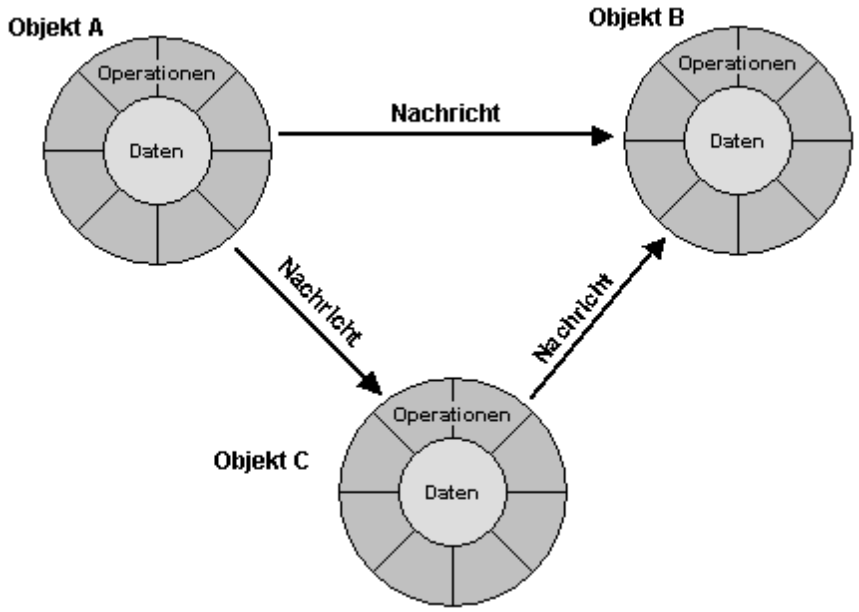



In Java bedeutet:

**public**:
generelle Sichtbarkeit, d.h. jedes Objekt kann auf das Attribut oder die Methode zugreifen.

**private**:
Sichtbarkeit nur innerhalb der Klasse, d.h. ein Objekt einer anderen Klasse kann auf das Attribut oder die Methode nicht zugreifen.

MERKE




Mit gezielten gesetzten **sondierenden und verändernden Methoden** kann man den Schutz kontrolliert wieder aufheben.

## Übung 3 – Beispiel zur Kapselung

a)
Kopiere das Projekt „aphaFormen_Ball_Vererbung“ aus Übung 1 in deinen Ordner und ändere den Namen in „alphaFormen_Ball_Kapselung“ ab.

b)
Ändere den Quelltext in BALL wie nebenstehend.
Lösche auch alle sondierenden und verändernden Methoden.

Die Attribute *besitzer* und *farbe* sind nun **private** gesetzt und können von einer anderen Klasse aus nicht verändert werden.

```
public class BALL extends KREIS
{
   private String besitzer;
   private String farbe;
   public BALL() {
       super();
       this.besitzer = "Hans";
       this.farbe = "rot";
   }
   public BALL(int rNeu) {
      super(rNeu);
      this.besitzer = "Hans";
      this.farbe = "rot";
}
```

# Übung 3 – Beispiel zur Kapselung

c)
Teste dies, indem du eine neue Klasse TEST mit nebenstehenden Quelltext erstellst.
Was passiert beim Compilieren der Klasse?

```
public class TEST {
    public void teste() {
        BALL b = new BALL();
        String neugierig = b.besitzer;
        b.besitzer = "Susi";
        b.farbe = "gelb";
    }
}
```

## Übung 3 – Beispiel zur Kapselung

d)
Auf das Attribut *besitzer* soll lesend zugegriffen werden können.

Schreibe dazu in der Klasse BALL eine sondierende Methode für *besitzer* und ändere den Quelltext von TEST wie untenstehend.

```
public class TEST {
    public void teste() {
        BALL b = new BALL();
        System.out.println("Besitzer: "    + b.nenneBesitzer() );
    }
}
```

## Übung 3 – Beispiel zur Kapselung

e)
Auf das Attribut *farbe* soll lesend und schreibend zugegriffen werden können. Die Änderung von *farbe* soll aber nur die Farben “rot“ und “blau“zulassen. Schreibe dazu in der Klasse BALL eine sondierende Methode für *farbe* sowie eine verändernde Methode wie folgt.
*(Die if-Anweisung wird im nächsten Kapitel genauer erklärt.)*

```
public class BALL{
...
    public void setzeFarbe(String farbeNeu) {
        if ( farbeNeu=="rot" || farbeNeu=="blau" ) {
            this.farbe = farbeNeu;
    }
...
}
```

# Übung 3 – Beispiel zur Kapselung

f)

Teste deine Änderungen mit der Klasse TEST:

```
public class TEST {
    public void teste(){
    BALL b = new BALL();
    System.out.println("Besitzer: " + b.nenneBesitzer());
    b.setzeFarbe("blau");
    System.out.println("Farbe: " + b.nenneFarbe());
    }
}
```

# Übung 4 – Kapselung in der Klasse SCHLAEGER

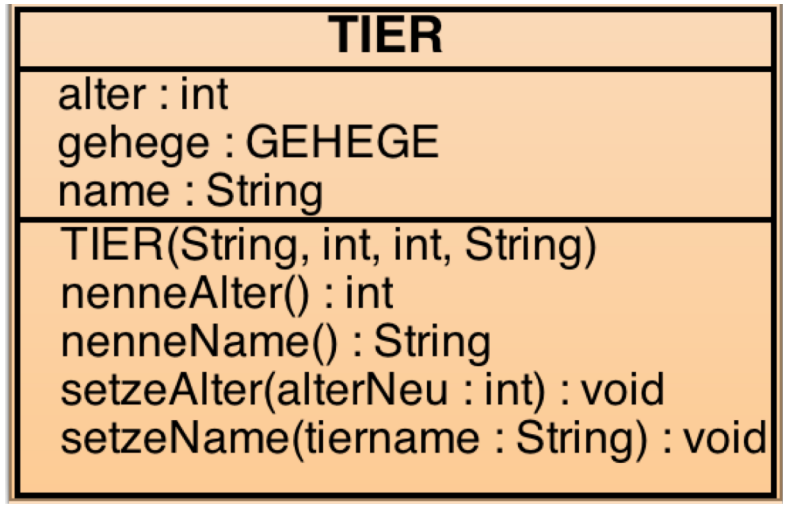
a)
Setze in der Klasse SCHLAEGER die Attribute *deltaX* und *deltaY* private.

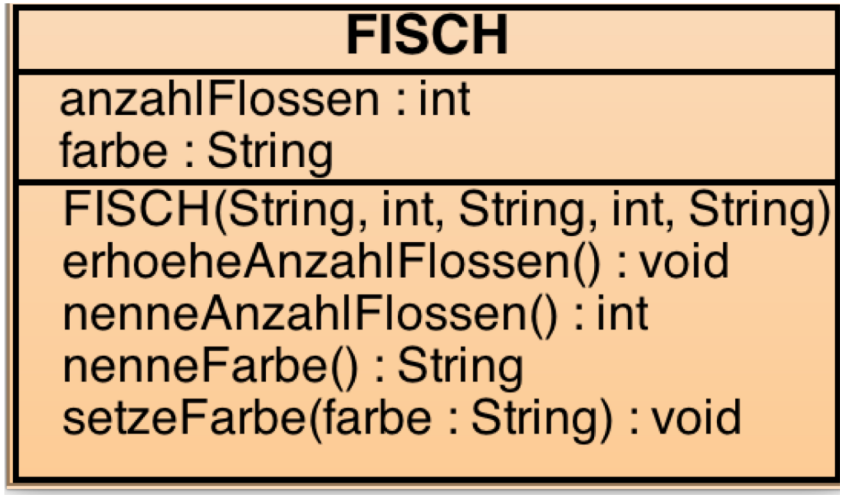
Schreibe sondierende und verändernde Methoden und teste mithilfe einer Testklasse, ob die Kapselung wie gewünscht funktioniert.

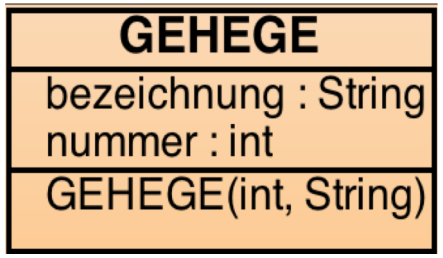
b)
Die Klasse SCHLAEGER erbt das Attribut *private farbe* von RECHTECK.
Prüfe in einem der Konstruktoren von SCHLAEGER, ob du es dort direkt durch die Anweisung *farbe = "gelb"* ändern kannst.

# Übung 5 – Tiere im Zoo, Quelltext ergänzen

Die Klassen TIER, FISCH und GEHEGE sind Teil einer einfachen Verwaltung von Tieren im Zoo:

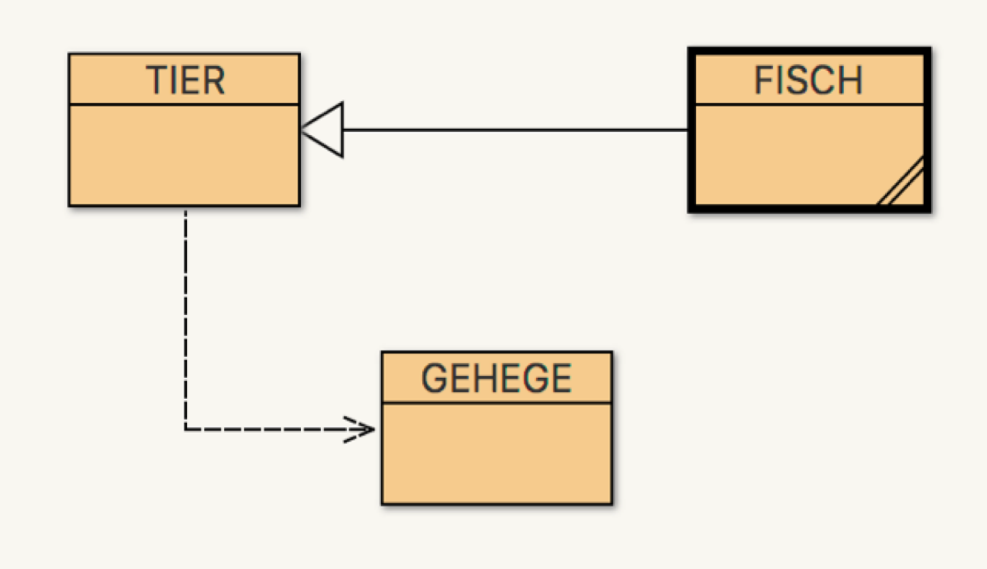


# Übung 5 – Tiere im Zoo, Quelltext ergänzen

Klassendiagramm:




**TIER**

alter : int
gehege : GEHEGE
name : String

TIER(String, int, int, String)
nenneAlter() : int
nenneName() : String
setzeAlter(alterNeu : int) : void
setzeName(tiername : String) : void

**FISCH**

anzahlFlossen : int
farbe : String

FISCH(String, int, String, int, String)
erhoeheAnzahlFlossen() : void
nenneAnzahlFlossen() : int
nenneFarbe() : String
setzeFarbe(farbe : String) : void

**GEHEGE**

bezeichnung : String
nummer : int

GEHEGE(int, String)

# Übung 5 – Tiere im Zoo, Quelltext ergänzen

1. Implementiere die Klasse TIER. Du kannst dazu den folgenden Lückentext als Gerüst verwenden:

```
public class TIER{
    String name;  int alter;   GEHEGE gehege;

    //Konstruktor
    [          ](String name, int alter, int nummer, String bezeichnung){
        [          ] = name;
                     = alter;
        gehege = [                                                    ]
    }

    //Methoden
    public [    ] setzeName [                    ]{
        name = tiername;
    }

    public [    ] nenneName() {
        [          ]
    }

}
```

# Übung 5 – Tiere im Zoo, Quelltext ergänzen

2. Implementiere die Klasse FISCH. Du kannst dazu den folgenden Lückentext als Gerüst verwenden:

```
public                                  {

    String farbe;   int anzahlFlossen;

    //Konstruktor
    public FISCH (String name, int alter, String farbe, int nummer,
                  String bezeichnung){

                                                                    
        this.farbe = farbe;
        this.anzahlFlossen = 2;
    }

    //Methoden

   //erhöht die Anzahl der Flossen um 1
   public void erhoeheAnzahlFlossen(){
                                                                    
    }

}
```

# Übung 5 – Tiere im Zoo, Quelltext ergänzen

3. Erstelle die Klasse GEHEGE mit folgendem Quelltext:

```
public class GEHEGE {
    private int nummer;
    String bezeichnung;

    public GEHEGE(int nummer, String bezeichnung){
        this.nummer = nummer;
        this.bezeichnung = bezeichnung;
    }

}
```

In der Klasse TIER soll mithilfe einer Methode *nenneGehegeNummer()* die Gehegenummer ausgegeben werden. Führe die notwendigen Änderungen in den Klassen GEHEGE und TIER durch. Das Attribut nummer in der Klasse GEHEGE soll private bleiben.

Schreibe dann in der Klasse FISCH eine Methode *datenAusgeben()* , in der alle Attributwerte in einem Textfenster ausgegeben werden.