Objektorientiertes Programmieren – Teil 2

Objektorientierte Analyse – Teil II

Spiele noch einmal das Spiel *PingPong* mit deinem Nachbarn und beobachte genau das Verhalten der Objekte.

Zuerst die Logik …

Welche Attribute braucht der Schläger mindestens?

Welche Daten-Typen sind dafür geeignet?

Über welche Methoden muss der Schläger mindestens verfügen?

Geben diese Methoden etwas zurück?

Von welchem Daten-Typ ist die Rückgabe?

Brauchen die Methoden Übergabe-Parameter?

Von welchem Daten-Typ sind diese Parameter?

Wie sehen ähnliche Überlegungen bei der Klasse *BALL* aus?

Fertige erneut erweiterte Klassen-Karten in denen Attribute und Methoden beschrieben werden.

Arbeite mit Bleistift oder am PC, damit du Änderungen unkompliziert vornehmen kannst.

Lasse sowohl bei den Attributen als auch bei den Methoden noch etwas Platz, falls du spätere Ideen noch einfügen musst.

… dann die Grafik

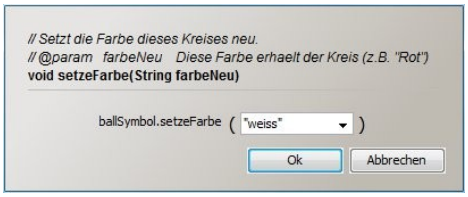
Bisher haben wir nur den logischen Aufbau und das Verhalten der Objekte in Form von Attributen und Methoden realisiert. Ohne Grafik gibt das Spiel jedoch keinen Sinn! Im Folgenden sollst du nun die grafische Darstellung deiner Spiel-Objekte realisieren.

Öffne in BlueJ noch einmal das Projekt *alpha_Formen*. Erstelle einen Kreis und zwei Rechtecke. Experimentiere solange mit den Methoden-Aufrufen herum, bis du das Bild auf der ersten Seite des ersten Teils dieses Skrips hergestellt hast.

Verwende die Direkteingabe und führe über alle Methoden-Aufrufe genau Buch!
Du wirst am Ende dieses zweiten Kapitels jeden Klick in JAVA-Befehle umsetzen müssen.

```
KREIS ballSymbol = new KREIS();
```


```
ballSymbol.setzeFarbe("weiss");
```



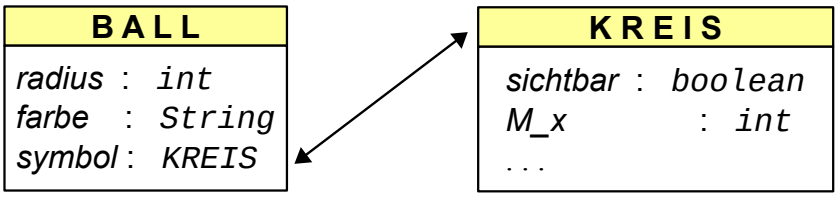
TIPP: In der Objekt-orientierten Programmierung verfolgt man die Devise, dass jede Klasse genau eine Aufgabe hat.

- Die Klassen *BALL* und *SCHLAEGER* z.B. haben die Aufgabe, die Logik dieser Spielfiguren zu beschreiben.
- Die Klassen *KREIS* und *RECHTECK* hingegen haben die Aufgabe, eine Grafik am Bildschirm darzustellen.

Anschließend müssen die beiden zusammen gehörigen Klassen „aneinander gekoppelt“ werden.

Aggregation

Um beide Aufgaben aneinander zu koppeln, führt man nun ***Referenz-Attribute*** ein. Darunter versteht man Attribute, deren Daten-Typ eine andere (vielleicht sogar selbst geschriebene) Klasse ist. So braucht jeder Ball nun neben den Attributen wie `radius` (vom Daten-Typ `int`) und `farbe` (vom Daten-Typ `String`) ein neues Attribut `symbol` vom Daten-Typ `KREIS`.



In diesem Fall spricht man von einem ***Aggregat***. Das bedeutet, dass ein Objekt (neben den „gewöhnlichen“ Attributen) auch noch aus anderen Objekten besteht.

Die **kleine Raute** am linken Ende der Verbindungslinie soll verdeutlichen, dass die Klasse *BALL* ein Aggregat ist, welche ein Attribut der Klasse *KREIS* besitzt *(ein Ball enthält genau einen Kreis)*.

Zeichne eine Klassen-Karte für die Klasse *SCHLAEGER* mit dem Referenz-Attribut `symbol`.

Zeichne außerdem ein Klassen-Diagramm, welches die Aggregation von *SCHLAEGER* und *RECHTECK* veranschaulicht.

Die Programmier-Sprache JAVA – Referenz-Objekte

Zur Umsetzung dieser neuen Inhalte in JAVA sind in unserer Klasse *BALL* nur drei Schritte notwendig.

Referenz-Attribute deklarieren

Das deklarieren des Referenz-Attributs geht genauso wie man es auch bei den einfachen Attributen getan hat:

Bspl.:

```
/**
 * Klasse Ball im Spiel PingPong
 */
public class BALL {

      private int radius;
      private KREIS symbol;
      ...

}
```

BALL
radius : int
symbol : Kreis
...

Es wird einfach nur der Daten-Typ *KREIS* verwendet.

Konstruktor anpassen – new-Operator und Punktnotation

Das neu deklarierte Referenz-Attribut muss nun im Konstruktor noch initialisiert werden. Hierzu wird zunächst mit Hilfe des ***new-Operators*** ein neues Objekt erzeugt und in der Variablen *symbol* gespeichert. Anschließend werden mittels Punktnotation Methoden des Referenz-Objekts *symbol* aufgerufen, um die Grafik an die Logik anzupassen:

Bspl.:

```
/**
 * Klasse Ball im Spiel PingPong
 */
public class BALL {

      private int radius;
      private KREIS symbol;
      …


      /**
       * Konstruktor der Klasse Ball
       * @param  radius     Radius des Balls
       * @param  ...        ...
       * ...
       */
      public BALL(int radius, …) {
            this.radius = radius;
            this.symbol = new KREIS();
            this.symbol.setzeRadius(this.radius);
            …
      }

}
```

Restliche Methoden anpassen – noch mehr Punktnotation

Alle bisherigen Methoden der Klasse *BALL*, die Attribut-Werte verändern, müssen nun ebenfalls angepasst werden. Nach der Änderung der Logik muss nun auch die Grafik an die neuen Werte angepasst werden:

Bspl. 1: Den Radius des Balls neu setzen

```
/**
 * Methode zum Setzen des Radius
 * @param  radius     Radius des Balls
 */
public void setzeRadius(int radius) {
      this.radius = radius;                    erst die Logik ...
      this.symbol.setzeRadius(this.radius);    ... dann die Grafik
}
```

Bspl.: Die Farbe des Balls neu setzen

```
/**
 * Methode zum Setzen der Farbe
 * @param  farbe      Farbe des Balls als String (z.B. "rot")
 */
public void setzeFarbe(String farbe) {
      this.farbe = farbe;
      this.symbol.setzeFarbe(farbe);
}
```

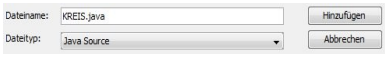
Programmier-Umgebung BlueJ – Teil III

Das war nun reichlich viel Theorie, die erst einmal in aller Ruhe und Ausführlichkeit in die Praxis umgesetzt werden will …

- Öffne in BlueJ dein Projekt mit der Klasse *BALL*.

- Importiere die Klasse KREIS aus dem Projekt *alpha_Formen*.
  Hierzu gehst du folgendermaßen vor:

  - *Klicke auf „Bearbeiten“*
  - *Klicke anschließend auf „Klasse aus Datei hinzufügen …“*
  - *Suche im erscheinenden Datei-Browser nach dem Projekt alpha_Formen und wähle daraus die Datei KREIS.java.*
  - *Klicke auf „Hinzufügen“*
  - *Klicke auf „Übersetzen“.*



Implementiere nun alle oben besprochenen Änderungen in deiner Klasse *BALL*.

Klicke nach jeder wesentlichen Änderung auf „Übersetzen“, damit du eventuelle Fehler gleich bemerkst.

Implementiere außerdem in allen anderen „setze-Methoden“ die Ergänzung für die grafische Darstellung.

Erzeuge ein Objekt deiner Klasse *BALL* und teste alle selbst geschriebenen Methoden ausgiebig bis du sicher sagen kannst, dass alles funktioniert.

- Jede Klasse soll nur genau eine Aufgabe haben.
  Deshalb trennen wir Logik und Grafik.
- Ein Objekt kann wiederum aus Objekten bestehen.
  Das nennt man ***Aggregation***.
- Ein Attribut, welches ein anderes Objekt darstellt,
  nennt man ***Referenz-Attribut***.

- Die Variablen ganz am Anfang einer Klasse nennt man auch ***globale Variablen***. Mit einem voran gesetzten `this` kann man im gesamten Quelltext der Klasse darauf zugreifen.
- Globale Variablen (Objekt-Variablen) werden immer und ausschließlich vor dem Konstruktor deklariert.
- Im Gegensatz dazu gibt es ***lokale Variablen***, die nur innerhalb einer Methode gelten, weil sie dort neu erzeugt werden. Will man mit lokalen Variablen arbeiten darf man auf keinen Fall ein `this` davor stellen!
- Man schreibt nur dann den ***Daten-Typ vor eine Variable***, wenn man sie gerade neu erzeugen möchte! Hat man die Variable bereits erzeugt und will sie nun verwenden, so darf auf keinen Fall den Daten-Typ davor setzen.

Objektorientierte Analyse – Teil III

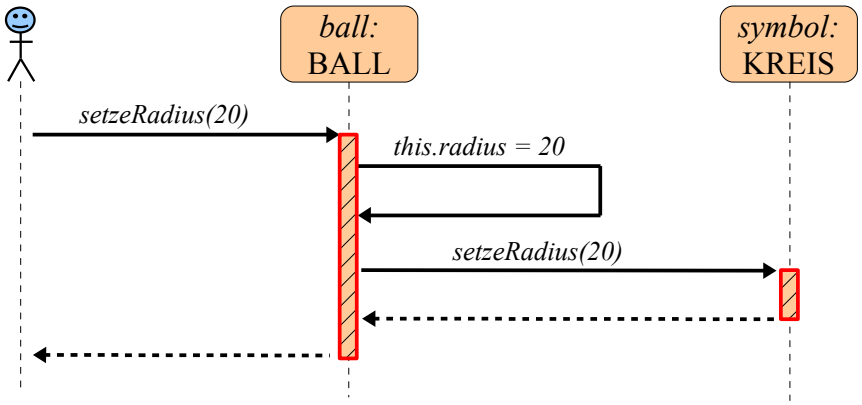
Sequenz-Diagramme

Wenn man es nun mit Aggregaten zu tun hat, dann verliert man schnell den Überblick über die in einer Methode zu erledigenden Aufgaben.

- So muss z.B. beim Ändern des Radius in der Klasse *BALL* zunächst das Attribut `radius` auf den neuen Wert gesetzt werden.
- Anschließend muss zusätzlich dem `symbol` mitgeteilt werden, dass es sich mit dem neuen Radius neu zeichnen soll.

Zum Veranschaulichen, welches Objekt wann etwas an sich selbst oder an anderen Objekten verändert, kommt ein sog. ***Sequenz-Diagramm*** zum Einsatz. Hier werden alle beteiligten Objekte nebeneinander dargestellt. Nach unten gerichtet denkt man sich die Zeit-Achse. Pfeile deuten an, wann ein Objekt etwas an sich selbst verändert oder eine Methode eines anderen Objekts aufruft.

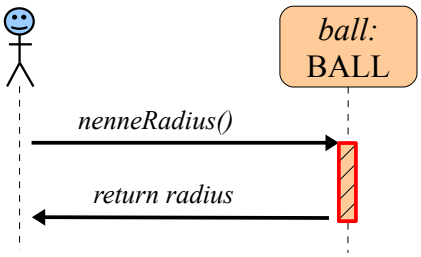
Beispiel:



Dabei geben die schraffierten Rechtecke an, wann ein Objekt aktiv ist.

Hat eine Methode einen Rückgabe-Wert, so wird der nach links weisende Pfeil nicht gestrichelt sondern durchgezogen gezeichnet. Außerdem beschriftet man ihn mit dem Rückgabe-Wert.

Beispiel:



Ein ***Sequenz-Diagramm*** stellt den zeitlichen Ablauf der Kommunikation zwischen Objekten dar.

Fertige ein *Sequenz-Diagramm* zu den Methoden `setzeFarbe(...)`, `setzePosition(...)` und `nenneFarbe()` der Klasse *BALL* an.

Übungsaufgaben zur Festigung

Nimm dir etwas Zeit und versuche alle neu gelernten Fachbegriffe in eigenen Worten zu erklären:

Aggregation, Referenz-Attribut, `new`-Operator, lokale und globale Variable, `this`, Sequenz-Diagramm

Ergänze hierzu dein „Vokabelheft“ … ergänze bei Bedarf eigene Beispiele …

Auftrag zum selbständigen Anwenden:

1. Importiere nun auch die Klasse *RECHTECK* und deine bisherige Version der Klasse *SCHLAEGER*.
2. Erweitere nun die Klasse *SCHLAEGER* um ein Referenz-Attribut und führe in allen entsprechenden Methoden die nötigen Änderungen durch..

Übersetze nach jedem wesentlichen Arbeitsschritt um eventuelle Fehler gleich zu erkennen.

Teste ausgiebig alle Methoden und kontrolliere die Attribut-Werte auch im Objekt-Inspektor.

Jetzt sollen sich die Objekte endlich bewegen:

1. Erweitere deine Klassen *BALL* und *SCHLAEGER* um zwei weitere Attribute `vX` und `vX`, welche die momentane Geschwindigkeit des Objekts darstellen sollen.
2. Implementiere zu jedem neuen Attribut „setze-Methoden“ und „nenne-Methoden“.
   Vergiss die Java-Doc-Kommentare nicht!
3. Implementiere eine Methode `bewegen()` in jeder Klasse, die zuerst zu den Koordinaten die Geschwindigkeit addiert
   Bspl.: `this.x = this.x + this.vX`
   und anschließend das Symbol an den neuen Mittelpunkt setzt. *(auch Java-Doc-Kommentar!)*

Übersetze nach jedem wesentlichen Arbeitsschritt um eventuelle Fehler gleich zu erkennen.

Teste ausgiebig deine neuen Methoden und kontrolliere die Attribut-Werte auch im Objekt-Inspektor.

Zeichne ein Sequenz-Diagramm zu dieser Methode.

Eine ***komplexe Wert-Zuweisung*** wird immer von rechts nach links ausgewertet:

Beispiel: `this.x = this.x + this.vX` *(z.B. x=100 und vX=5)*

- Zuerst wird das Ergebnis des Ausdrucks rechts vom „=“ ausgewertet:
  `this.x + this.vX` *(ergibt 105)*
- Anschließend wird dieses Ergebnis in der Variablen links des „=“ gespeichert:
  `this.x = 105`

Objektorientierte Analyse – Teil IV

Unsere Schläger und auch der Ball sollen das Spielfeld nicht verlassen. Damit dies möglich ist, muss die Bewegung der Schläger oben und unten am Spielfeld stoppen.

Der Ball muss – wenn er oben oder unten am Spielfeld angekommen ist – reflektiert werden. Hierzu dienen bedingte Anweisungen. Die Bewegung des Balls wird noch weitaus komplizierter. Er kann außerdem an den Schlägern reflektiert werden, im Aus landen …

Um all diese Möglichkeiten zu erfassen, definieren wir Zustände, in denen sich der Schläger (Ball) befinden kann. Anschließend denken wir über Ereignisse nach, die den Schläger (Ball) von einem Zustand in den anderen überführen.

Nachdem das Problem auf diese Weise voll erfasst wurde, fällt das Umsetzen in JAVA-Code nicht mehr schwer …

Zustands-Übergangs-Diagramme

Zur Erinnerung:

Ein ***Zustand*** eines Objekts ist durch einen vollständigen Satz von Attribut-Werten festgelegt. Jede Änderung eines oder auch mehrerer Attribut-Werte überführt damit das Objekt in einen anderen Zustand. Man spricht von einem ***Zustandsübergang***.

Zustand 1 *Zustand 2*

Jeder Zustandsübergang wird von einer ***auslösenden Aktion*** verursacht. Dies ist im obigen Beispiel der Aufruf der Methode `setzeY(70)`.

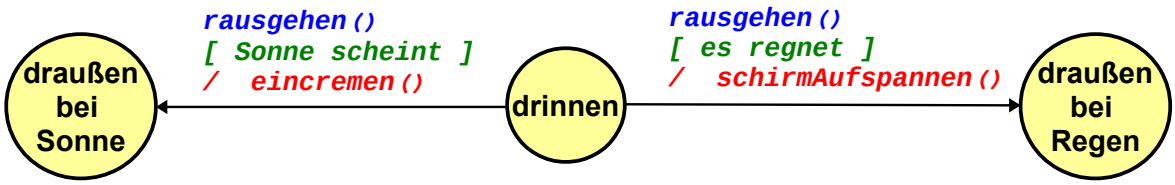
Nun kann es sein, dass es bei einer bestimmten auslösenden Aktion mehrere verschiedene Möglichkeiten gibt. Zum Beispiel soll sich unser Ball beim Aufruf der Methode *bewegen()* geradlinig bewegen, wenn er sich innerhalb des Spielfelds befindet. Er soll sich aber nicht mehr geradlinig weiter bewegen sondern reflektieren, wenn er z.B. den oberen Spielfeldrand erreicht hat. Es entscheidet also manchmal noch eine zusätzliche ***Bedingung***, in welchen Zustand ein Objekt nach einer auslösenden Aktion übergeht.

In so einem Fall spricht man auch von einem ***bedingten Übergang***.

Beim Reflektieren des Balls muss etwas anderes getan werden als bei der geradlinigen Bewegung. Deshalb geht der Ball danach ja auch je nach Bedingung in einen anderen Zustand über. Das was getan werden muss bei einem Zustandsübergang nennt man ***ausgelöste Aktion***. So muss z.B. bei einer geradlinigen Bewegung erst der neue Ort berechnet und anschließend die grafische Darstellung an diesen neuen Ort angepasst werden. Bei einer Reflexion hingegen muss zusätzlich vorher die neue Bewegungsrichtung festgelegt werden …

All diese Informationen werden an den Übergangspfeil geschrieben. Um dabei zu unterscheiden, was die auslösende Aktion, die Bedingung und die ausgelöste Aktion ist, verwendet man folgende Notation:

- **auslösende Aktion:** **1. Eintrag** am Übergangspfeil (ohne weitere Zusätze)
- **Bedingung:** **2. Eintrag** am Übergangspfeil (in eckigen Klammern [ ])
- **ausgelöste Aktion/en:** **3. Eintrag** am Übergangspfeil (nach einem Slash /)

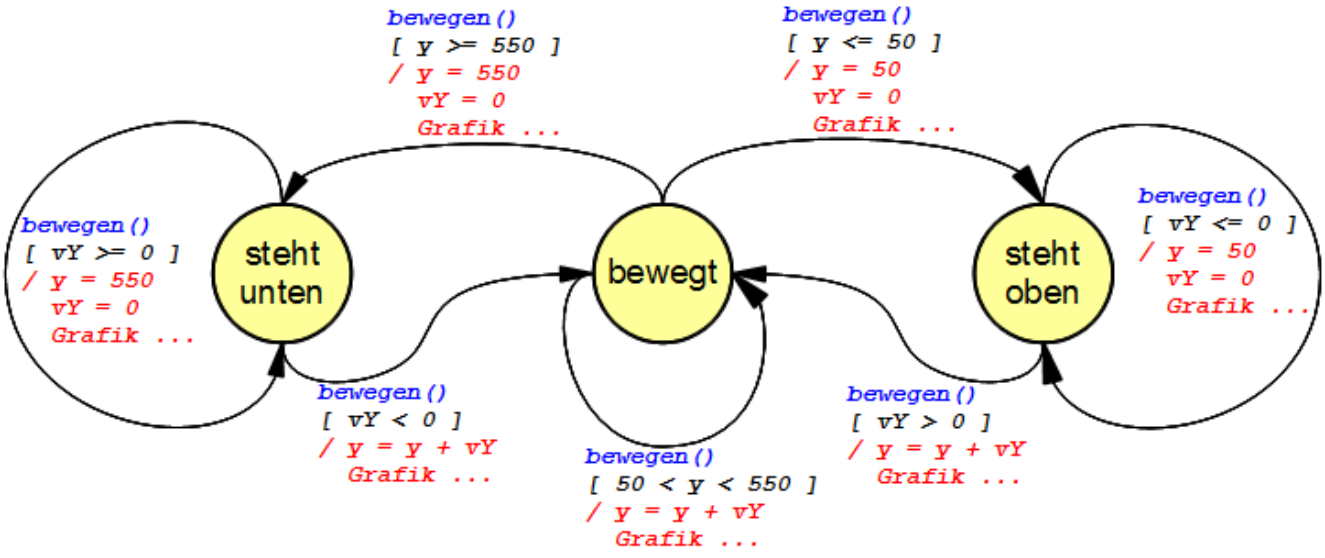


Zustands-Übergangs-Diagramm mit bedingtem Übergang

Der Schläger

Eigentlich ist jede Änderung eines Attribut-Werts eine Zustands-Änderung. Für uns sind aber nur drei Zustände von Bedeutung:

- Der Schläger befindet sich **oben** und steht still
- Der Schläger befindet sich **unten** und steht still
- Der Schläger befindet sich irgendwo dazwischen und **bewegt** sich hinauf oder hinunter



Zustands-Übergangs-Diagramm eines Schlägers im Spiel PingPong

Der Ball

Zeichne für den Ball selbst ein Zustands-Übergangs-Diagramm.

Welche Zustände sind wichtig?

Welche Übergänge gibt es? Unter welchen Bedingungen finden sie statt?

Was muss bei jedem Übergang getan werden?

Die Programmier-Sprache JAVA – Fallunterscheidungen

Die bedingte Anweisung

- Die einfachste Art der Fallunterscheidung in JAVA sieht folgendermaßen aus:

WENN …
DANN …

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

das „dann“ wird durch die geschweiften Klammern ersetzt !!!

- Oft benötigt man auch eine Alternativ-Anweisung:

WENN …
DANN …

```
if (Bedingung) {
    Anweisung(en)_1
}
```

SONST …

```
else {
    Anweisung(en)_2
}
```

- Benötigt man noch mehr Fälle, so ist das auch kein Problem:

WENN …
DANN …

```
if (Bedingung_1) {
    Anweisung(en)_1
}
```

SONST WENN …

```
else if (Bedingung_2) {
    Anweisung(en)_2
}
```

SONST WENN …

```
else if (Bedingung_3) {
    Anweisung(en)_3
}
```

SONST …

```
else {
    Anweisung(en)_4
}
```

den „sonst-Fall“ kann man auch weglassen!

Logische Operatoren

Bei den Bedingungen brauchst du Möglichkeiten um ***Vergleiche*** von zwei Werten durchzuführen oder mehrere Bedingungen mit ***UND*** oder ***ODER*** zu verbinden.

Operator	Bedeutung	Beispiel
==	Vergleich	`x == 10;` `farbe == "rot";` `sichtbar == false;`
<	kleiner	`x < 300;` `y <= 200;`
>	größer	`x > 300;` `y >= 200;`
!	nicht	`farbe != "rot";` `vX != 5;`
&&	und	`x > 100 && x < 500`
||	oder	`farbe == "rot" || farbe == "gelb"`

Bspl. 1: Stoppen des Schlägers am oberen oder unteren Bildschirmrand

```
// Wenn der Schläger oben oder unten anstößt,
// dann soll er am Spielfeld-Rand stehen bleiben
if (this.y <= 50) {
      this.vY = 0;
      this.y = 50;
}
else if (this.y >= 550) {
      this.vY = 0;
      this.y = 550;
}
```

Bspl. 2: Reflexion des Balls am oberen oder unteren Bildschirmrand

```
// Reflexion des Balls am oberen oder unteren Bildschirmrand
if ( this.y < 10  ||  this.y > 590 ) {
      this.vY = -1 * this.vY;                  neuer Wert ist -1 mal alter Wert
}
this.x = this.x + this.vX;                     neuer Wert ist alter Wert plus vX
this.y = this.Y + this.vY;                     neuer Wert ist alter Wert plus vY
this.symbol.setzeMittelpunkt(this.x, this.y);
```

Schreibe deine Methode `bewegen()` in der Klasse *BALL* um wie im Beispiel oben gezeigt:

- **WENN** die y-Koordinate des Mittelpunkts kleiner als `10` ist **ODER** die y-Koordinate des Mittelpunkts größer als `590` ist,
  - so soll `vY` den Wert `-vY` annehmen.
- Erst nach dieser bedingten Anweisung soll der Mittelpunkt neu gesetzt und das Symbol verschoben werden.

Dies bewirkt, dass der Ball oben und unten am Spielfeld reflektiert wird.

Schreibe deine Methode `bewegen()` in der Klasse *SCHLAEGER* folgendermaßen um:

- **WENN** die y-Koordinate des Mittelpunkts kleiner als `50` ist, so soll
  - `vY` den Wert `0` annehmen
  - `y` den Wert `50` annehmen.
- **SONST WENN** die y-Koordinate des Mittelpunkts größer als `550` ist, so soll
  - `vY` den Wert `0` annehmen
  - `y` den Wert `550` annehmen.
- **SONST** soll wie gehabt *vX* zu *x* und *vY* zu *y* addiert werden.
- Erst nach allen Fallunterscheidungen soll das Symbol verschoben werden.

Dies bewirkt, dass der Schläger oben und unten nicht aus dem Spielfeld läuft.

Schreibe zur Übung eine Klasse *HANDY*.

Ein Handy hat eine `PIN` und eine Methode `pinEingeben(...)`, bei der überprüft wird, ob der PIN richtig ist. Außerdem gibt es eine Methode `pinAendern(...)` bei der drei Parameter nötig sind: alte PIN, neue PIN, neue PIN zur Kontrolle. Eine Methode `telefonieren()` soll nur funktionieren, wenn vorher der PIN richtig eingegeben wurde. Beim telefonieren werden 10ct von einer Prepaid-Karte abgezogen, solange diese noch nicht leer ist. Bei einer zweiten Methode `telefonieren(...)` soll man die Anzahl der Einheiten mitgeben können, die abgebucht werden sollen.

Vergiss die Java-Doc-Kommentare nicht!

Übersetze regelmäßig und teste ausgiebig unter Verwendung des Objekt-Inspektors.

Übungsaufgaben nach Bedarf

Nimm dir etwas Zeit und versuche alle neu gelernten Fachbegriffe in eigenen Worten zu erklären:

komplexe Wert-Zuweisung, bedingte Anweisung

Ergänze hierzu dein „Vokabelheft“ … ergänze bei Bedarf eigene Beispiele …

Betrachte nochmal dein Projekt mit der Klasse *AUTO*.

Verändere die Methoden `beschleunigen(...)` und `bremsen(...)` so, dass das Auto nicht schneller als 180 km/h und nicht langsamer als 0 km/h wird.

Setze auch Java-Doc-Kommentare!

Modelliere eine Klasse *AMPEL*.

Die Ampel verfügt über einen Wahrheitswert `istAn`. Sie hat außerdem einen `zustand` "rot" bzw. "gruen".

Sie verfügt über die Methoden `einschalten()`, `ausschalten()`, `schalteRot()` und `schalteGruen()`. Eine weitere Methode `schalten()` soll von rot auf grün schalten und umgekehrt. *(je nach vorhandenem Zustand)*

Eine frisch erzeugte oder gerade eingeschaltete Ampel soll immer rot sein.

Modelliere zuerst eine Klassen-Karte, ein Klassen-Diagramm und ein Zustands-Übergangs-Diagramm, bevor du programmierst!

Wie sieht das Sequenz-Diagramm zu `schalten()` aus?

Sollte dir das nicht anspruchsvoll genug sein, so erzeuge eine AutoAmpel mit drei Lampen.

Implementiere deine Klasse *AMPEL* in BlueJ.

Importiere die benötigten Klassen *KREIS* und *RECHTECK* aus dem Projekt *alpha_Formen*.

Setze auch Java-Doc-Kommentare!

Übersetze regelmäßig und teste ausgiebig unter Verwendung des Objekt-Inspektors.

Modelliere eine Klasse *PINGPONG* für dein PingPong-Spiel. Diese Klasse soll ein Ball-Objekt und zwei Schläger-Objekte als Referenz-Attribute enthalten. Beim Erzeugen eines PingPong-Objekts soll ein fertiges Spielfeld sichtbar sein.

1. Fertige eine Klassen-Karte mit allen nötigen Attributen und Daten-Typen an.
2. Fertige ein Klassen-Diagramm mit allen beteiligten Klassen an.
3. Überlege, was der Konstruktor alles initialisieren muss. Braucht er Übergabe-Parameter?

(Mache dir an dieser Stelle noch keine Gedanken über Methoden zum Bewegen der Objekte, das lösen wir gemeinsam mit einem netten Trick und etwas neuem Wissen im dritten Teil dieses Skripts ...)

Implementiere die Klasse *PINGPONG* in BlueJ.

Verwende dazu dein Projekt mit den Klassen *BALL* und *SCHLAEGER* sowie *KREIS* und *RECHTECK*.

Setze auch Java-Doc-Kommentare!

Übersetze regelmäßig und teste ausgiebig unter Verwendung des Objekt-Inspektors.

Selbstkontrolle: Hast du auch alles verstanden?

… dann kannst du bestimmt die folgenden Fragen ohne Probleme beantworten.

Wenn nicht, dann arbeite das entsprechende Teil-Kapitel noch einmal durch …

Erkläre die Begriffe *Aggregation* und *Referenz-Attribut*.

Was kannst du über den Daten-Typ eines Referenz-Attributs sagen?
Was muss man alles im Konstruktor tun, um ein Referenz-Attribut zu initialisieren?

Was wird in einem *Sequenz-Diagramm* dargestellt?
Beschreibe auch das generelle Aussehen in Worten.

Erkläre und unterscheide die Begriffe *globale Variable* und *lokale Variable*.
Wo werden sie jeweils deklariert?
Von wo aus und wie kann man jeweils darauf zugreifen?

In welchen Situationen muss man vor eine Variable den Daten-Typ schreiben, wann darf man es auf keinen Fall tun?

Erkläre, wie man in JAVA eine *bedingte Anweisung* realisiert.
Welche Schlüsselwörter brachst du hierzu?

Wieso haben wir die Logik und die Grafik eines Objekts in zwei Klassen aufgeteilt?

Beschreibe, wie man in ein BlueJ-Projekt eine Klasse aus einem anderen Projekt importieren kann.

Woran erkennt man im Code einen Java-Doc-Kommentar?
Wo überall sollten Java-Doc-Kommentare stehen?
Wofür sind Java-Doc-Kommentare gut?
Wie werden darin Übergabe-Parameter oder Rückgabe-Werte angegeben?

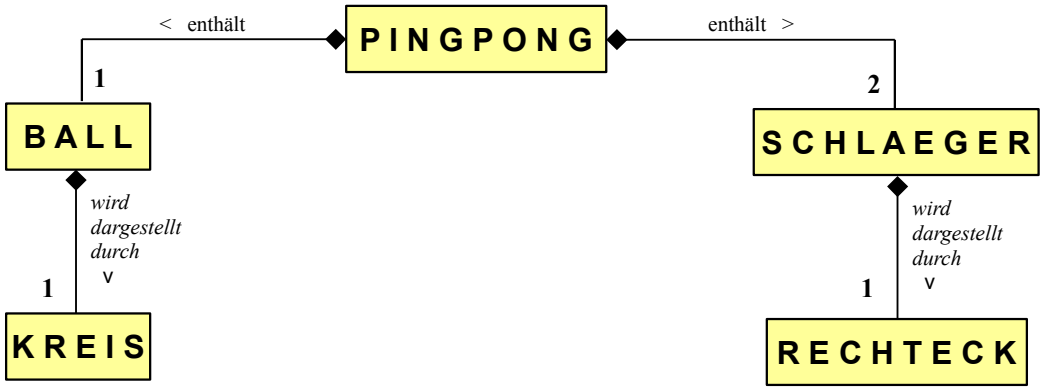
Welche der folgenden Fehler-Meldungen des Compilers sind dir schon einmal begegnet und auf welchen Fehler deuten sie hin?

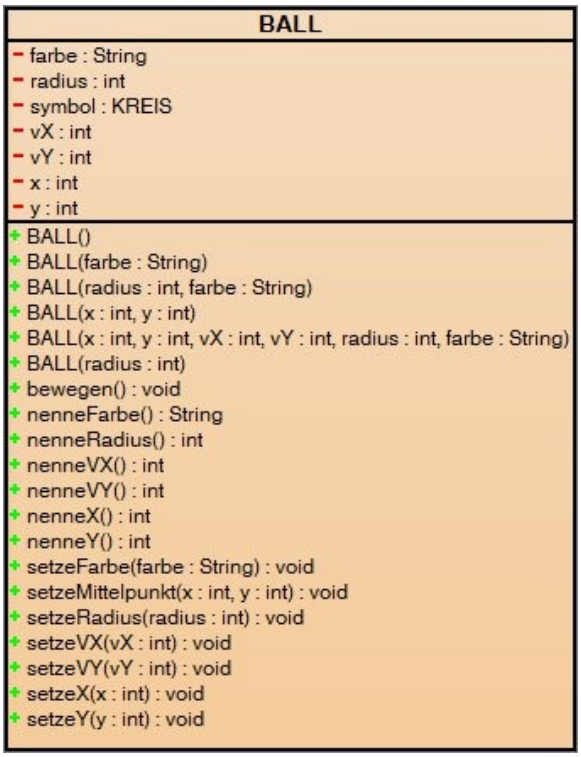
- *cannot find symbol – class string*
- *cannot find symbol – variable Radius*
- *incompatible types – found java.lang.String but expexted int*
- *reached end of file while parsing*
- *<identifier> expected*

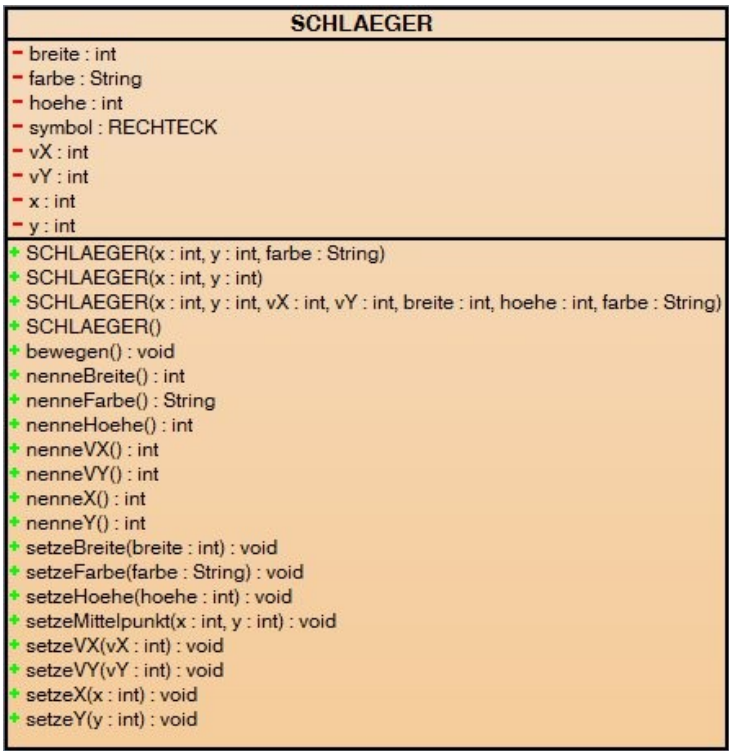
Zusammenfassung

- In diesem Kapitel hast du zuerst die die **grafische Darstellung** deiner Klassen *BALL* und *SCHLAEGER* realisiert und dabei Referenz-Attribute und das Prinzip der Aggregation kennen gelernt.
- Anschließend hast du deinen Objekten „Leben eingehaucht“ indem du die Methode bewegen() implementiert hast.
- Zu guter Letzt hast du deine erste einfache Version der Klasse *PINGPONG* erstellt, die aus einem Referenz-Attribut der Klasse *BALL* und zwei Referenz-Attributen der Klasse *SCHLAEGER* besteht.

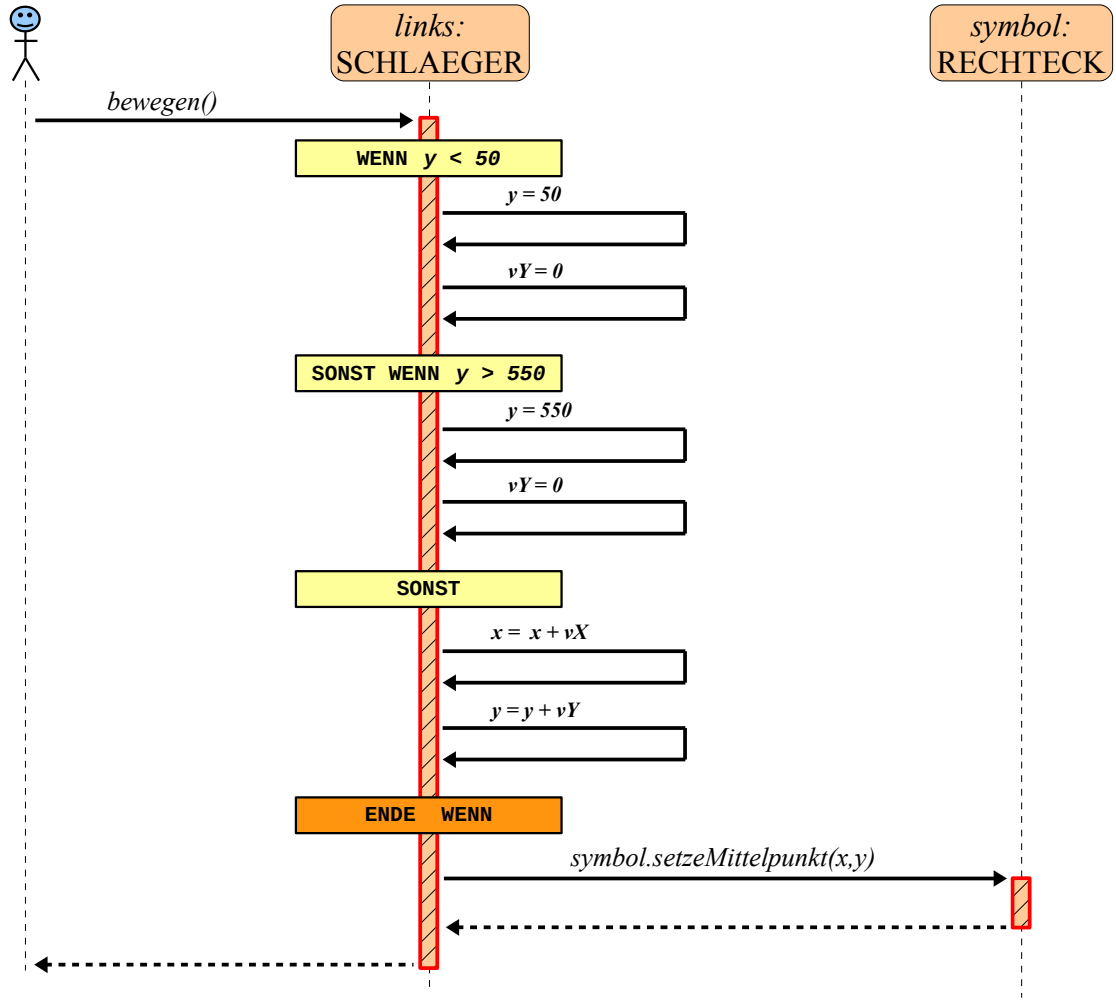
Du solltest nun folgende Situation geschaffen haben:







Die Methode `bewegen()` in der Klasse *SCHLAEGER* wird hier noch einmal durch ein Sequenz-Diagramm veranschaulicht:



Ausblick

Im nächsten Teil dieses Skripts wirst du lernen,

- **den Ball automatisch zu bewegen**
- **die Schläger auf Tastatur-Eingaben reagieren zu lassen**

Hierzu sind allerdings neue theoretische Grundlagen unabdingbar …