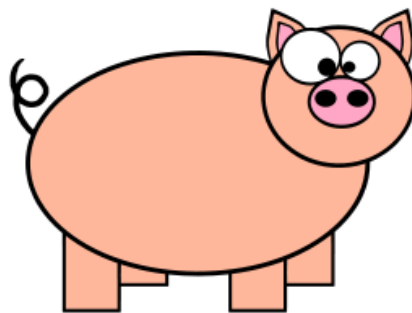


2.2

Schweine im Weltall



Benötigte IDEs:

Greenfoot, BlueJ

Verfasser:

Niko Diamadis

Erstellungs-/ Änderungsdatum

30. Juni 2020

Inhaltsverzeichnis

1	Implementierung des Grundgerüsts	1
1.1	Projekt vorbereiten	1
1.2	Erste Bewegungen	1
1.3	Zufällige Bewegungen	1
1.4	Nicht so zappelig	2
1.5	Spielende auf Mausklick	3
1.6	Altes Schwein	5
1.7	Spiel vorbereiten	6
2	Fachkonzept - Vererbung	7
3	Fachkonzept - Klassenmethoden	8
4	Übungen	9
4.1	Zufällig Springen	9
4.2	Aus der Welt laufen	10
4.3	Gewinnstufen	12
4.4	Eigene Ideen	15
4.5	Mathematische Funktionen	18
4.6	GeoCalc	19

1 Implementierung des Grundgerüsts

1.1 Projekt vorbereiten

Das Schwein kann bereits Methoden ausführen, auch wenn wir noch nichts programmiert haben, denn da das Schwein eine Unterklasse von `Actor` ist, übernimmt es automatisch alle Attribute und Methoden aus der Oberklasse.

1.2 Erste Bewegungen

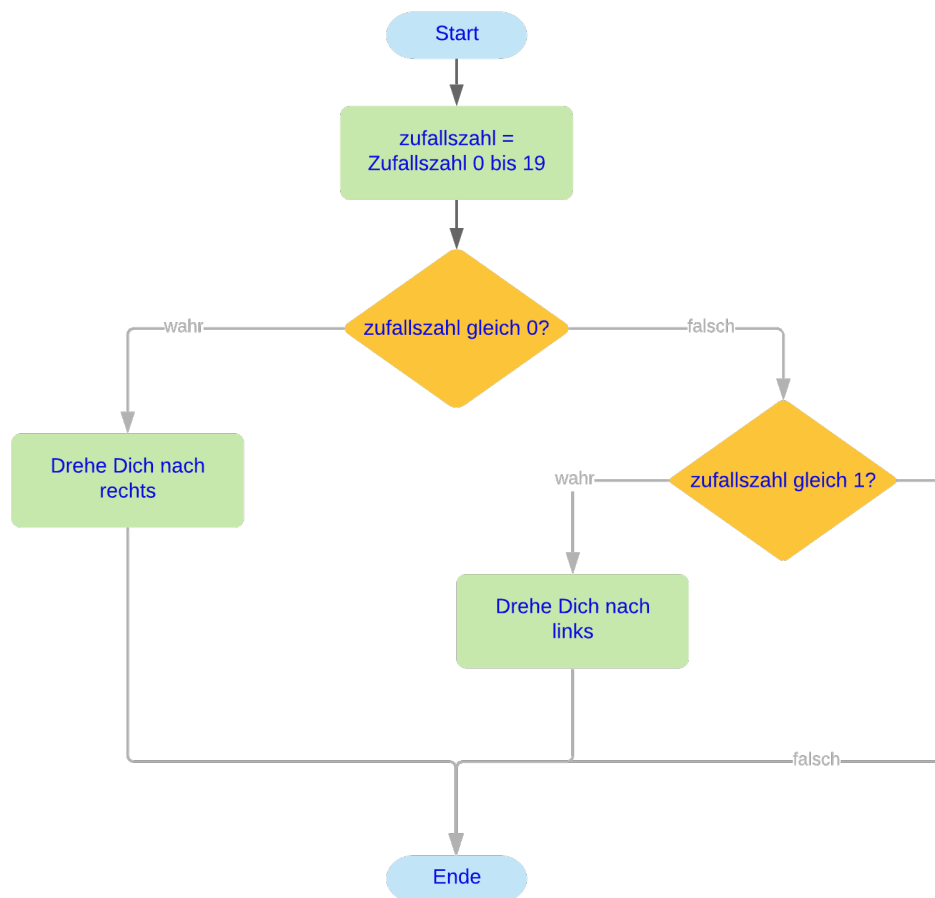
```
1 public void act() {  
2     move(2);  
3     turn(2);  
4 }
```

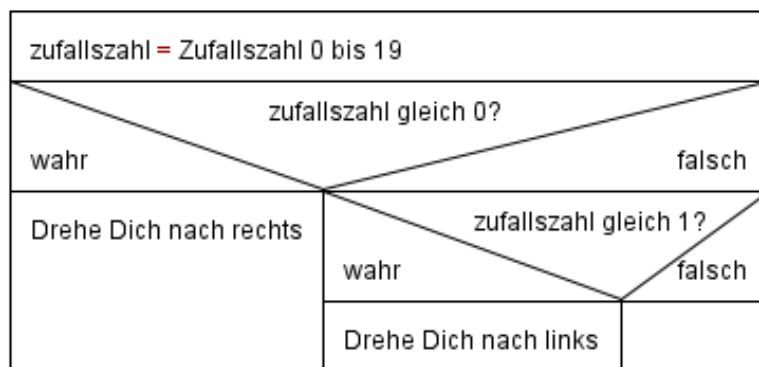
1.3 Zufällige Bewegungen

```
int zufallszahl = new java.util.Random().nextInt(2);  
  
if(zufallszahl == 0)  
    turn(2);  
else  
    turn(-2);
```

Nicht verwirren lassen, wenn man nur eine Zeile an Code in der if- bzw. else-Schleife stehen hat, können die geschweiften Klammern weggelassen werden. Ich benutze sie jedoch IMMER...

1.4 Nicht so zappelig





```

1 public void act() {
2     move(2);
3
4     int zufallszahl = new java.util.Random().nextInt(20);
5
6     if (zufallszahl == 0) {
7         turn(20);
8     } else if (zufallszahl == 1) {
9         turn(-20);
10    }
11 }

```

Wenn die Zufallszahl im Bereich von 2 bis 19 liegt passiert nichts. Innerhalb der `act`-Methode wird dann also nur `move(2)` aufgerufen.

Wir haben jetzt eine Wahrscheinlichkeit von 20%, dass sich das Schwein bewegt, es dreht sich dann jedoch mehr als vorher.

So können wir die zitterige Bewegung vermeiden.

1.5 Spielende auf Mausklick

Folgender Code muss einfach in der `act`-Methode zum restlichen Code hinzugefügt werden.

```

1 if (Greenfoot.mouseClicked(this)) {
2     Greenfoot.playSound("pig.wav");

```

```
3     Greenfoot.stop();  
4 }
```

1.6 Altes Schwein

Da nun auch ein Attribut hinzugefügt werden muss, hier nun der ganze Code der Klasse:

```
1 import greenfoot.*;
2
3 public class Schwein extends Actor {
4     int alter;
5
6     public void act() {
7         alter++;
8
9         if (alter == 1000) {
10             setImage("boom.png");
11             Greenfoot.stop();
12         }
13
14         if (Greenfoot.mouseClicked(this)) {
15             Greenfoot.playSound("pig.wav");
16             Greenfoot.stop();
17         }
18
19         move(2);
20
21         int zufallszahl = new java.util.Random().nextInt(20);
22
23         if (zufallszahl == 0) {
24             turn(20);
25         } else if (zufallszahl == 1) {
26             turn(-20);
27         }
28     }
29 }
```

1.7 Spiel vorbereiten

Wenn ihr die beschriebenen Schritte befolgt habt, findest du den Code zum automatischen Erzeugen des Schweins in der Welt-Unterklasse `PigWorld`.

```
1 public PigWorld() {  
2     super(600, 450, 1);  
3     prepare();  
4 }  
5  
6 private void prepare() {  
7     Schwein schwein = new Schwein();  
8     addObject(schwein, 303, 227);  
9 }
```

Wie hoffentlich zu erkennen ist, wird im Konstruktor die Methode `prepare` aufgerufen, welche eben für das Erzeugen des Schweins verantwortlich ist.

In der Methode wird ein neues Objekt der Klasse `Schwein` erzeugt mit dem Namen `schwein`.

Anschließend wird das Objekt in die Welt gesetzt, in meinem Falle auf die Koordinaten

$x = 303$; $y = 227$.

2 Fachkonzept - Vererbung

Zu dieser Seite sind meines Erachtens nach keine Anleitungen und/oder Erläuterungen nötig.

Wenn doch Fragen aufkommen, schreib' einfach an **nikodiamond3@gmail.com**.

3 Fachkonzept - Klassenmethoden

Zu dieser Seite sind meines Erachtens nach keine Anleitungen und/oder Erläuterungen nötig.

Wenn doch Fragen aufkommen, schreib' einfach an **nikodiamond3@gmail.com**.

4 Übungen

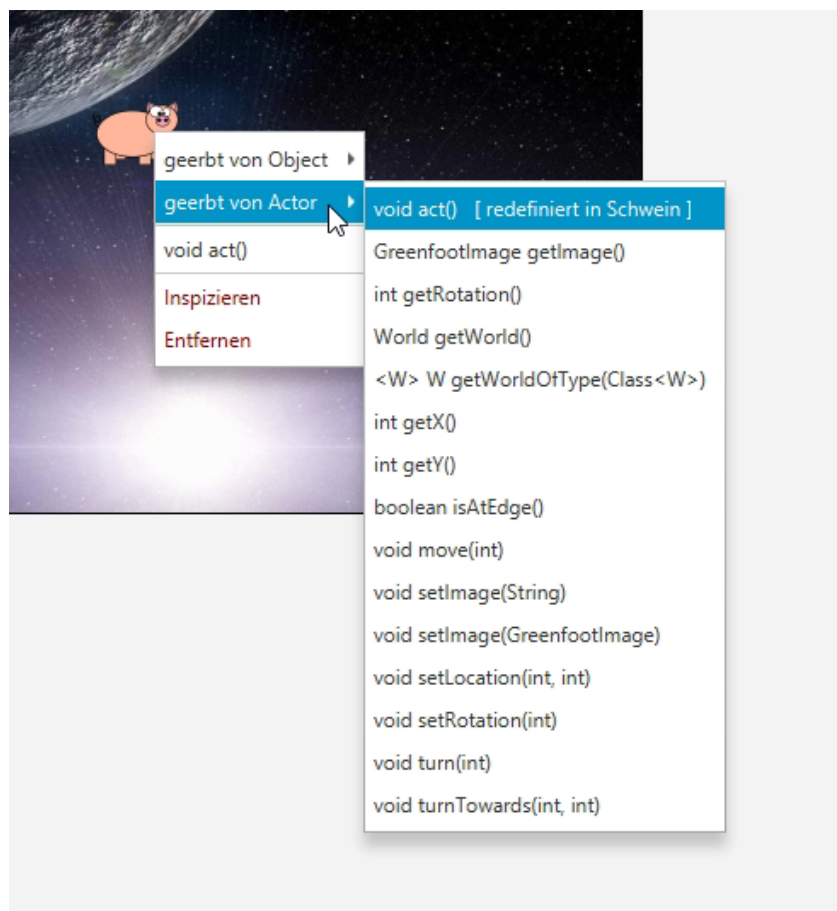
4.1 Zufällig Springen

Um herauszufinden, wie das Schwein ermitteln kann, ob es den Rand berührt, gucken wir uns an, welche Methoden es geerbt hat.

Dazu klickt man mit einem Rechtsklick auf das Schwein und geht über **geerbt von ...**.

Beim Durchsehen der Methoden stößt man auf die Methode **isAtEdge** mit boolean als Rückgabewert.

Das ist genau das Richtige für uns.



Wir wollen also, wenn das Schwein am Ende der Welt ist, es auf eine zufällige Position setzen.

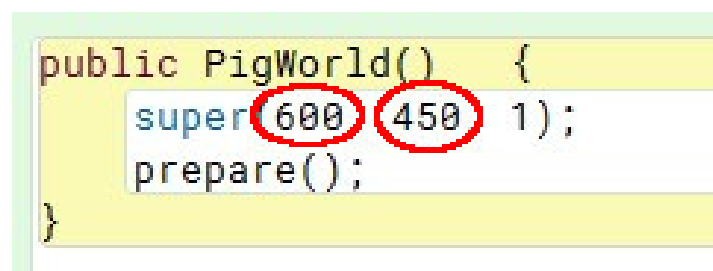
Als passende Methode findet man **setLocation** mit zwei Parametern, und zwar der x- und y-Koordinate.

Implementiert sieht das dann z.B. so aus:

```
1  if (isAtEdge()) {  
2      setLocation(Greenfoot.getRandomNumber(600), Greenfoot.getRandomNumber  
        (450));  
3  }
```

Bitte nicht wundern, die Zeile war zu lang für das Dokument, deshalb ist ein Zeilensprung eingebaut worden, es gehört aber alles in eine Zeile.

Die Breite und Höhe der Welt kann man aus dem Konstruktor der `PigWorld` ablesen.



```
public PigWorld() {  
    super(600, 450, 1);  
    prepare();  
}
```

4.2 Aus der Welt laufen

Der Programmierer wollte wohl erreichen, dass das Schwein bei Berühren des Endes der Welt auf die andere Seite gesetzt wird.

Jedoch gibt es da ein Problem:

Wenn das Schwein z.B. bei $x = 0$ ist, ist die erste Bedingung erfüllt und es wird auf `(599, getY())` gesetzt.

Anschließend wird jedoch überprüft, ob es bei $x = 599$ ist. Da dies auch erfüllt ist, wird es wieder zurück auf $x = 0$ gesetzt.

Um dieses Problem zu lösen, könnte man die vier if-Bedingungen in zwei if-/else if-Schleifen schreiben.

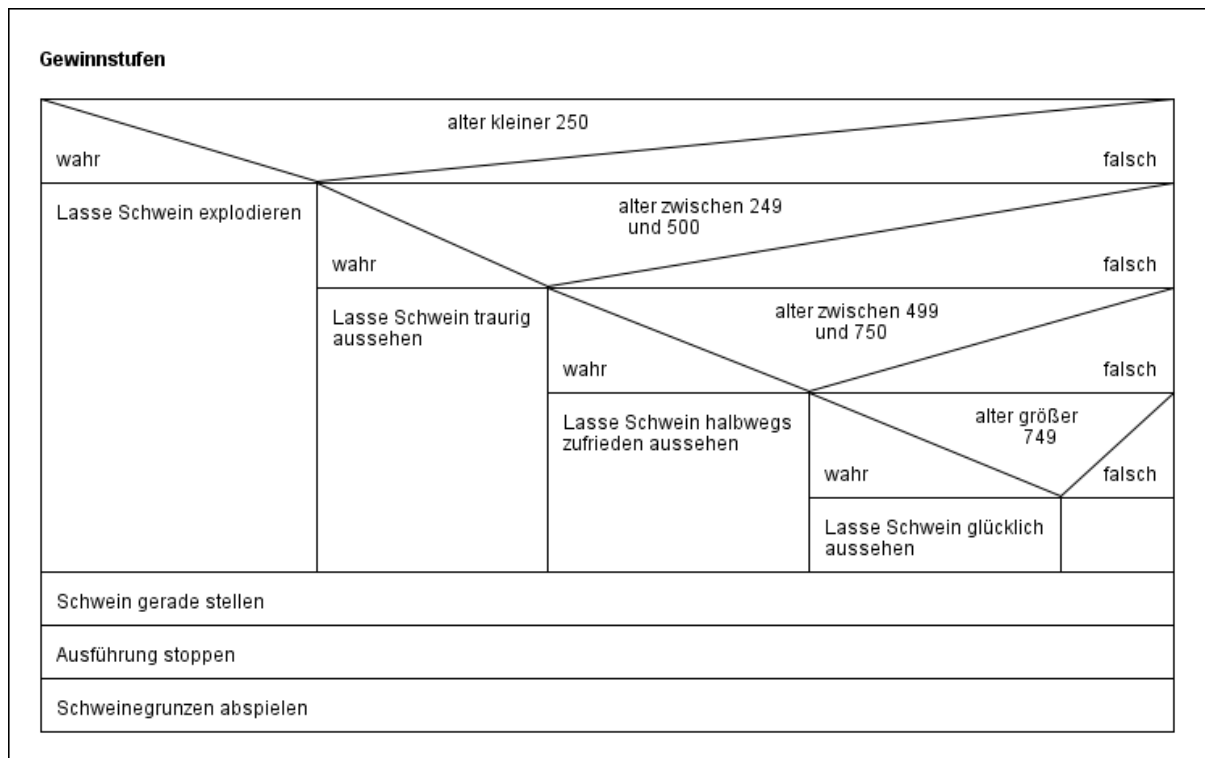
```
1  if (getX() == 0) {  
2      setLocation(599, getY());  
3  } else if (getX() == 599) {  
4      setLocation(0, getY());  
5  }  
6  
7  if (getY() == 0) {  
8      setLocation(getX(), 449);  
9  } else if (getY() == 449) {  
10     setLocation(getX(), 0);  
11 }
```

In eine Schleife dürfte man nämlich nicht alle vier schreiben, da so z.B. der Fall ausgeschlossen wird, wenn das Schwein bei $x == 0$ und $y == 0$ ist.

4.3 Gewinnstufen

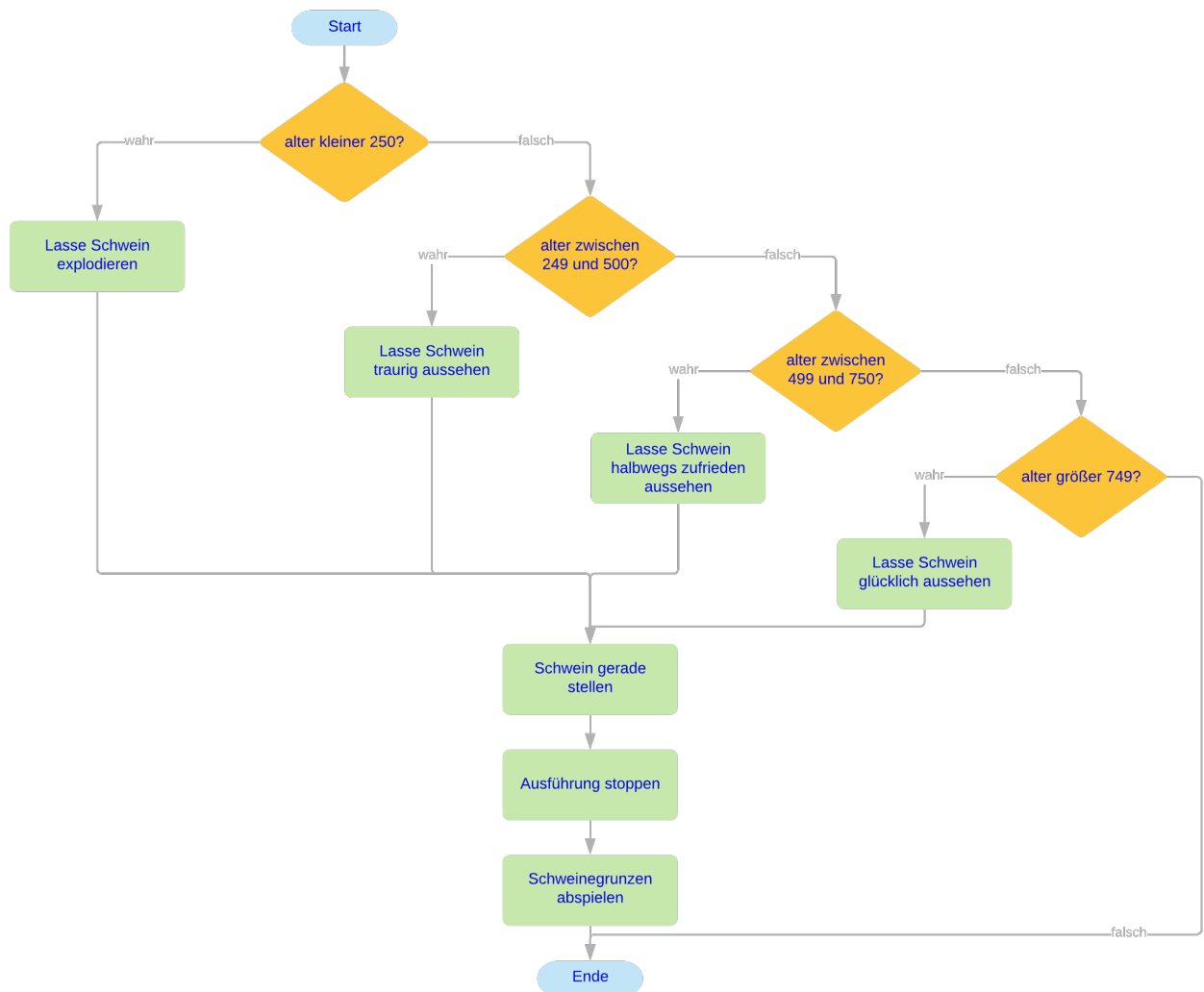
Ich nehme wieder das `alter`-Attribut zur Hilfe.

Ich habe in diesem Fall eine Fallunterscheidung in 250er-Schritten eingebaut, die nach Klicken des Schweins folgendermaßen aussieht:



Der Vorteil eines Struktogramms ist, dass die Blockform eine ordentliche Anschauung ermöglicht und der Ablauf der Aktionen eindeutig ist, Nachteil ist die unklarere Darstellung verschiedener Elemente und dass ein Struktogramm innerhalb weniger Schritte sehr breit werden kann.







Im Vergleich dazu das Flussdiagramm:



Bei Flussdiagrammen ist der Ablauf der Aktionen unklarer, da Pfeile bei Wiederholungen wieder zum Anfang oder zu anderen Elementen weiter am Ende verweisen können, jedoch ist dort klarer zu erkennen, um welche Art von Element es geht (z.B. while-Schleife, for-Schleife, if-Bedingung, ...).

Außerdem ist es leichter, die Form anzupassen, falls das Diagramm zu breit, zu groß oder zu hoch erscheint.

Die Bilder findet man im Projektordner im Unterordner `images`.

Name	Änderungsdatum	Typ	Größe
 boom.png	30.05.14 23:23	PNG-Datei	8 KB
 happy.png	30.05.14 23:23	PNG-Datei	5 KB
 ok.png	30.05.14 23:23	PNG-Datei	4 KB
 Piggie.png	30.05.14 23:23	PNG-Datei	3 KB
 traurig.png	30.05.14 23:23	PNG-Datei	5 KB
 weltraum.jpg	30.05.14 23:23	JPG-Datei	89 KB

Implementiert sieht der Code anschließend z.B. so aus:

```
1  if (Greenfoot.mousePressed(this)) {  
2      if (alter < 250) {  
3          setImage("boom.png");  
4      } else if (alter > 249 && alter < 500) {  
5          setImage("traurig.png");  
6      } else if (alter > 499 && alter < 750) {  
7          setImage("ok.png");  
8      } else if (alter > 749) {  
9          setImage("happy.png");  
10     }  
11  
12     setRotation(0);  
13     Greenfoot.stop();  
14     Greenfoot.playSound("pig.wav");  
15 }
```


4.4 Eigene Ideen

In dieser Aufgabe werde ich einfach nur die Beispiele verwirklichen, du kannst hier deine eigenen Ideen verwirklichen.



```
1  int drehzahl;  
2  
3  public void act() {  
4      ...  
5  
6      int zufallszahl = new java.util.Random().nextInt(2);  
7  
8      drehzahl++;  
9  
10     if (drehzahl == 20) {  
11         if (zufallszahl == 0) {  
12             turn(20);  
13         } else {  
14             turn(-20);  
15         }  
16  
17         drehzahl = 0;  
18     }  
19  
20     ...  
21 }
```

➤ Für zufällige Geschwindigkeit nutzen wir die implementierte Variable `zufallszahl` erneut.

```
1  zufallszahl = new java.util.Random().nextInt(20);  
2  
3  if (zufallszahl == 0) {  
4      Greenfoot.setSpeed(50);  
5  } else if (zufallszahl == 1) {  
6      Greenfoot.setSpeed(55);  
7  }
```

- Ich habe im Beispiel festgelegt, dass bei Fehlklick das Schwein zufällig ein wenig an der x- und y-Achse verschoben wird.

```
1  if (Greenfoot.mousePressed(this)) {
2      ...
3  } else if (Greenfoot.mousePressed(getWorld())) {
4      int zufallszahl1 = new java.util.Random().nextInt(100) - 50;
5      int zufallszahl2 = new java.util.Random().nextInt(100) - 50;
6
7      setLocation(getX() + zufallszahl1, getY() + zufallszahl2);
8  }
```

Ich habe zwei Hilfsvariablen benutzt, da ohne diese die Zeile (welche anfängt mit `setLocation(...)`) zu lang und unübersichtlich werden würde.

Das `getWorld()` in der Bedingung ruft die Methode `getWorld` auf, welche die Welt zurückgibt, wodurch man die Methode wie ein Objekt benutzen kann.

Mit allen Änderungen und Neuerungen sieht die ganze Klasse nun so aus:

```
1  import greenfoot.*;
2
3  public class Schwein extends Actor {
4      int alter;
5      int drehzahl;
6
7      public void act() {
8          if (Greenfoot.mousePressed(this)) {
9              if (alter < 250) {
10                 setImage("boom.png");
11             } else if (alter > 249 && alter < 500) {
12                 setImage("traurig.png");
13             } else if (alter > 499 && alter < 750) {
14                 setImage("ok.png");
15             }
16         }
17     }
18 }
```

```
15         } else if (alter > 749) {
16             setImage("happy.png");
17         }
18
19         setRotation(0);
20         Greenfoot.stop();
21         Greenfoot.playSound("pig.wav");
22     } else if (Greenfoot.mousePressed(getWorld())) {
23         int zufallszahl1 = new java.util.Random().nextInt(100) - 50;
24         int zufallszahl2 = new java.util.Random().nextInt(100) - 50;
25
26         setLocation(getX() + zufallszahl1, getY() + zufallszahl2);
27     }
28
29     if (getX() == 0) {
30         setLocation(599, getY());
31     } else if (getX() == 599) {
32         setLocation(0, getY());
33     }
34
35     if (getY() == 0) {
36         setLocation(getX(), 449);
37     } else if (getY() == 449) {
38         setLocation(getX(), 0);
39     }
40
41     move(2);
42
43     int zufallszahl = new java.util.Random().nextInt(2);
44
45     drehzahl++;
46
47     if (drehzahl == 20) {
48         if (zufallszahl == 0) {
49             turn(20);
50         }
```

```
51         } else {
52             turn(-20);
53         }
54
55         drehzahl = 0;
56     }
57
58     zufallszahl = new java.util.Random().nextInt(20);
59
60     if (zufallszahl == 0) {
61         Greenfoot.setSpeed(50);
62     } else if (zufallszahl == 1) {
63         Greenfoot.setSpeed(55);
64     }
65
66     alter++;
67 }
68 }
```

4.5 Mathematische Funktionen

Wahrscheinlich haben die Java-Entwickler sich dazu entschieden, da man durch Benutzung von Klassenmethoden das Erstellen von Objekten vermeiden kann.

Ein Objekt der Klasse `Math` würde meistens auch keinen Sinn machen.

4.6 GeoCalc

- a. Zum einen gibt es die Klasse **Geofigur** mit mehreren Attributen und Methoden.

Des Weiteren gibt es die Klasse **Rechteck**, Unterklasse von **Geofigur**, welcher alle Variablen und Methoden von **Geofigur** *vererbt* wurden.

Die in **Rechteck** implementierten Attribute und Methoden werden zu den vererbten Variablen und Methoden *ergänzt*. Als Ausnahme ist zu erwähnen, dass *Vererbtes überschrieben* wird, falls in der Unterklasse Variablen bzw. Methoden mit demselben Namen neu implementiert werden.

- b. Zur Berechnung des Abstandes habe ich den Satz des Pythagoras benutzt:

$$a^2 + b^2 = c^2$$

a ist in unserem Falle $x + \text{die Hälfte der Breite}$, b ist $y + \text{die Hälfte der Höhe}$, da wir ja den Abstand zwischen Ursprung und Mittelpunkt des Rechtecks errechnen möchte.

Zum Ausrechnen nehmen wir also die Quadratwurzel aus dem Quadrat von a und dem Quadrat aus b.

```
1  class Rechteck extends GeoFigur {
2      double breite;
3      double hoehe;
4
5      void setzeAusmasse(double b, double h) {
6          breite = b;
7          hoehe = h;
8      }
9
10     double getFlaeche() {
11         return breite * hoehe;
12     }
13
14
15
16
```

```
17     double getAbstandZumUrsprung() {  
18         return Math.sqrt(Math.pow(x + breite / 2, 2) + Math.pow(y +  
           hoehe / 2, 2));  
19     }  
20 }
```

- c. Zu dieser Aufgabe sind meines Erachtens nach keine Anleitungen und/oder Erläuterungen nötig.

Wenn doch Fragen aufkommen, schreib' einfach an **nikodiamond3@gmail.com**.

d. ➤ **1. Zeile**

Es wird ein Objekt `figur` der Klasse `GeoFigur` erstellt.

➤ **2. Zeile**

Am Objekt `figur` wird die Methode `setzeKoordinaten` mit den Parametern 0 und 2 aufgerufen, für `x` wird also 0 und für `y` wird 2 gesetzt.

➤ **3. Zeile**

Es wird der Rückgabewert der Methode `getAbstandZumUrsprung` vom Objekt `figur` in der Konsole ausgegeben.

➤ **4. Zeile**

Es wird ein Objekt `rechteck` der Klasse `Rechteck` erstellt.

➤ **5. Zeile**

Am Objekt `rechteck` wird die Methode `setzeKoordinaten`, welche von der Klasse `GeoFigur` vererbt wurde, mit den Parametern -2 und 2 aufgerufen, für `x` wird also -2 und für `y` wird 2 gesetzt.

➤ **6. Zeile**

Am Objekt `rechteck` wird die Methode `setzeAusmasse` mit den Parametern 4 und 4 aufgerufen, für `breite` wird also 4 und für `hoehe` wird auch 4 gesetzt.

➤ **7. Zeile**

Am Objekt `rechteck` wird die Methode `verschieben`, welche von der Klasse `GeoFigur` vererbt wurde, mit den Parametern 0 und 2 aufgerufen, `x` bleibt also gleich und für `y`

wird von 4 nun zu 6 geändert.

➤ **8. Zeile**

Es wird der Rückgabewert der Methode `getAbstandZumUrsprung` vom Objekt `rechteck` in der Konsole ausgegeben.

➤ **9. Zeile**

Es wird ein Objekt `g` der Klasse `GeoFigur` erstellt, jedoch mithilfe des Konstruktors der Klasse `Rechteck`, was erstmal keinen Fehler beim Compilieren hervorruft, da ein Rechteck, welches durch den Konstruktor aufgerufen wird, in eine `GeoFigur` verwandelt werden kann, da es eine Unterklasse ist, uns aber in den nächsten Schritten Probleme bereiten wird.

➤ **10. Zeile**

Am Objekt `g` wird die Methode `setzeKoordinaten` mit den Parametern 0 und 1 aufgerufen, für `x` wird also 0 und für `y` wird 1 gesetzt.

➤ **11. Zeile**

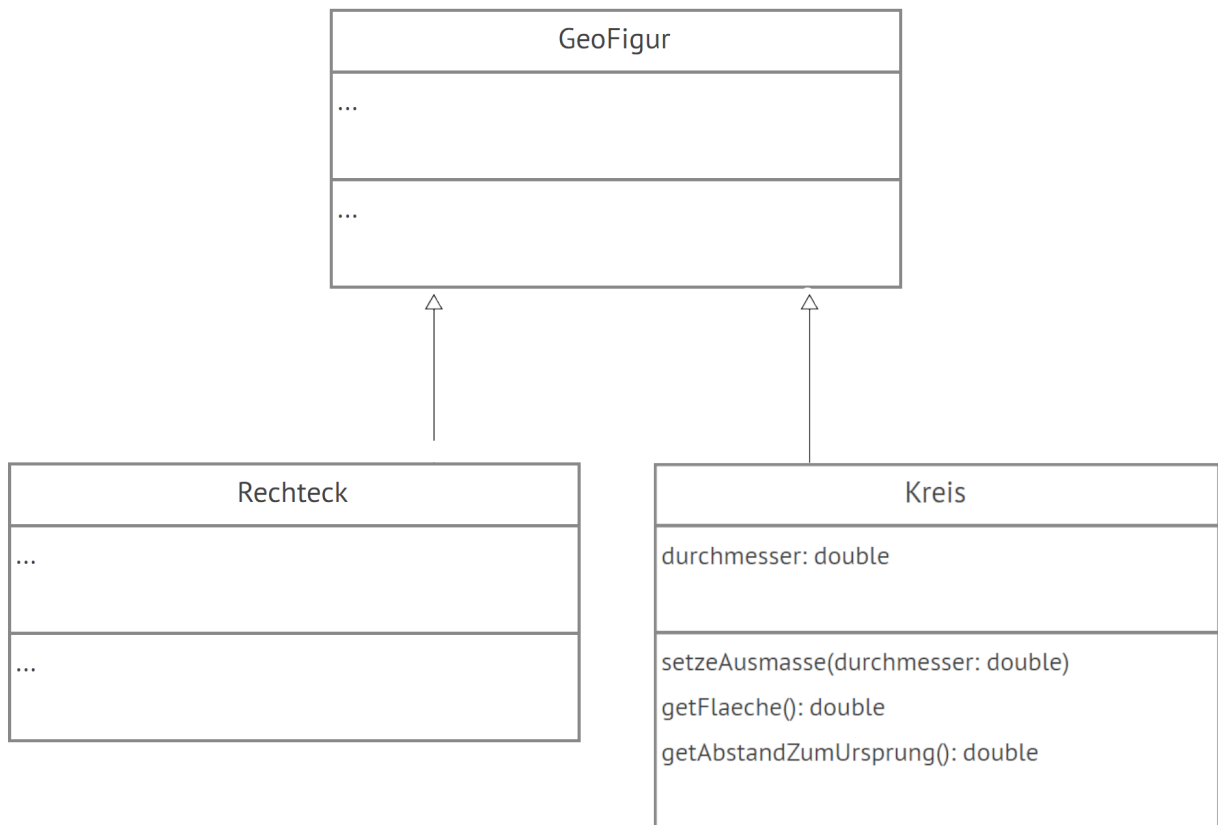
Am Objekt `g` wird versucht, die Methode `setzeAusmasse` mit den Parametern 1 und 4 aufzurufen, aber da `g` in ein Objekt der Klasse `GeoFigur` umgewandelt worden ist, hat `g` keinen Zugriff auf eine Methode dieses Namens.

Um das Problem zu lösen, könnte man in Zeile 9 „GeoFigur“ durch „Rechteck“ ersetzen. Zeile 10 wird keinen Fehler hervorrufen, da die Methode `setzeKoordinaten` auch wieder vererbt wird.

➤ **12. Zeile**

Es wird versucht, ein Objekt `r` der Klasse `Rechteck` zu erstellen, jedoch mithilfe des Konstruktors der Klasse `GeoFigur`, was nicht funktioniert und eine Fehlermeldung hervorruft, da die `GeoFigur`, welche durch Aufruf des Konstruktors aufgerufen wird, nicht zurück in eine Unterklasse verwandelt werden kann.

- e. Da beim Rechteck die gesamte Breite genommen wurde, habe ich mich beim Kreis für den Durchmesser entschieden. Zu beachten ist, dass jedoch mit dem Radius gerechnet wird. Die Koordinaten beschreiben die linke untere Ecke eines Quadrats, welches man anliegend an den Kreis um ihr herum zeichnen könnte.



```
1  class Kreis extends GeoFigur
2      double durchmesser;
3
4      void setzeAusmasse(double durchmesser) {
5          this.durchmesser = durchmesser;
6      }
7
8      double getFlaeche() {
9          return Math.PI * Math.pow(durchmesser / 2, 2);
10     }
11
```



```
12     double getAbstandZumUrsprung() {  
13         return Math.sqrt(Math.pow(x + durchmesser / 2, 2) + Math.pow(y +  
            durchmesser / 2, 2));  
14     }  
15 }
```

`Math.PI` ist einfach nur genauer als 3,14. Zur Berechnung der Fläche habe ich die folgende Formel benutzt:

$$A = \pi r^2$$

f. i.

```
1  class Linie extends GeoFigur {  
2      double laenge;  
3  
4      void setzeAusmasse(double laenge) {  
5          this.laenge = laenge;  
6      }  
7  
8      double getAbstandZumUrsprung() {  
9          return Math.sqrt(Math.pow(x + laenge / 2, 2) + Math.pow(y,  
            2));  
10     }  
11  
12     void skalieren(double faktor) {  
13         laenge *= faktor;  
14     }  
15 }
```

ii. ➤ Rechteck

```
1  void skalieren(double faktor) {  
2      breite *= faktor;  
3      hoehe *= faktor;  
4  }
```

► Kreis

```
1 void skalieren(double faktor) {  
2     durchmesser *= faktor;  
3 }
```

► Linie

```
1 void skalieren(double faktor) {  
2     laenge *= faktor;  
3 }
```

- iii. Nein, es macht keinen Sinn, denn alle Attribute und Methoden von **GeoFigur** sind ohne Probleme auch auf eine zweidimensionale Figur anzuwenden.

Bei der Linie z.B. nimmt man als Koordinaten wieder den Punkt unten links an der Figur, der Mittelpunkt in die Mitte der Linie und den Abstand kann man dann auch berechnen.