1. Greenfoot

Greenfoot (http://www.greenfoot.org/) ist eine **Entwicklungsumgebung**, die auf der **Java Programmiersprache** basiert.

Sie ist hauptsächlich dafür designt, das **Erlernen der Java Sprache zu vereinfachen** und spielend leicht **2D Spiele** zu erschaffen.

Greenfoot ist eine frei erhältliche Software und für die meisten Computer verfügbar, da sie sowohl Windows, als auch Mac OS X und Linux unterstützt.

2. Java

Die Java Sprache ist eine "objektorientierte" Programmiersprache, die um das Jahr 1996 herum erschaffen wurde.

Heutzutage ist sie eine der beliebtesten und meistgenutzten Sprachen der Welt. Sie wird genutzt um eine Vielzahl verschiedener Anwendungen zu realisieren (Spiele, Banking-Apps, Roboter, Android Smartphones, …)

Mit Java lassen sich Anwendungen schreiben, die auf verschiedenen „Betriebssystemen" laufen (Windows, Mac, Linux, …).

3. Klassen und Objekte

Bevor wir mit Greenfoot loslegen, müssen wir erst ein paar Worte über **Klassen** und **Objekt** in Java verlieren.

Java ist eine "**objektorientierte**" Programmiersprache: Das bedeutet, dass eine Anwendung aus „Objekten" aufgebaut ist.

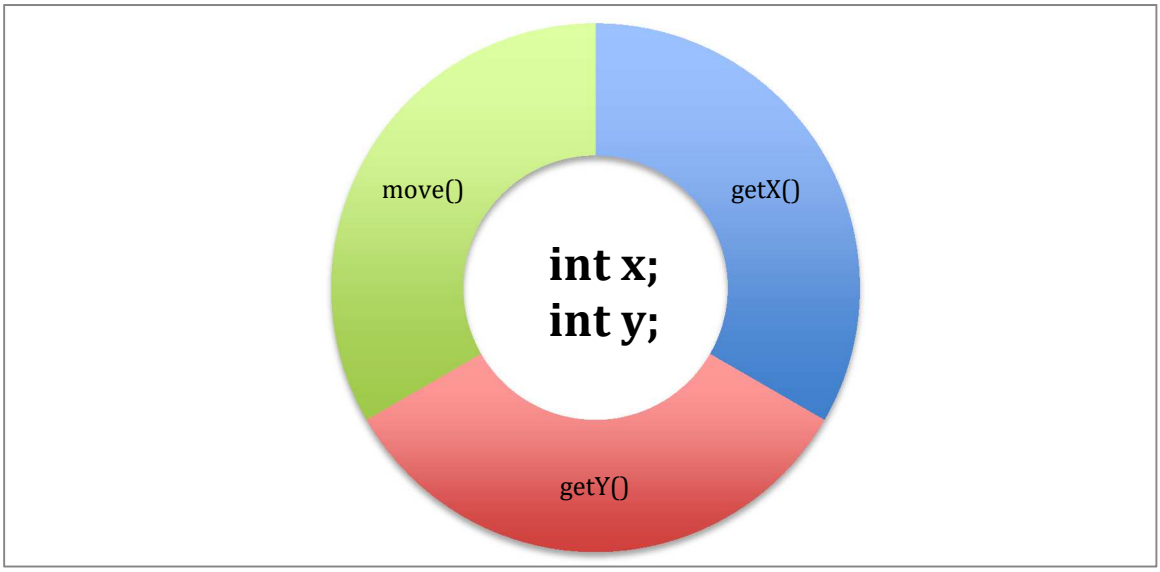
Betrachten wir ein konkretes Beispiel: Einen Spielcharakter.

Jedes Objekt besteht aus zwei Hauptteilen:

- Seinem « state » (Zustand), welcher aus einem Satz von **properties** (**Eigenschaften**) besteht
- Seinem « behaviour » (Verhalten), das ist die Liste von Aktionen, die das Objekt ausführen kann

In unserem Beispiel des Spielcharakters bedeutet das:

- Sein Zustand besteht aus seiner Position im Spiel, seiner Bewegungsrichtung, der Geschwindigkeit etc.
- Sein Verhalten besteht zum Beispiel aus den Bewegungen, die er machen kann, oder einem Angriff, den er ausführen kann.



Um ein Java-Objekt zu beschreiben, definieren wir eine **Klasse**. Eine Klasse ist eine Art „Vorlage“, die genutzt wird, um die Objekte zu konstruieren (diese nennt man auch Instanzen der Klasse).

In diesem Workshop werden wir diese Konzepte und die Grundlagen der Java Sprache mit Codebeispielen in Greenfoot kennenlernen. Aber lass uns erstmal mit einem kleinen Ausschnitt beginnen, um die Grundlagen von Java darzustellen. Du kannst auch später noch einmal zu diesem Teil zurückkehren, falls du während des Workshops dir diese Konzepte noch einmal besser vor Augen führen möchtest.

```java
public class GameElement {

	private int x;
	private int y;

	public GameElement(int initX, int initY) {
		x = initX;
		y = initY;
	}

	public int getX() {
		return x;
	}

	public int getY() {
		return y;
	}

	public void setX(int newX) {
		x = newX;
	}

	public void setY(int newY) {
		y = newY;
	}

	public void move(int xMove, int yMove) {
		x = x + xMove;
		y = y + yMove;
	}
}
```

Die Klasse (class)

- Sie ist als « **public** » (öffentlich) deklariert, um allen anderen Objekten in unserer Anwendung zu erlauben, mit den Objekten dieser Klasse zu interagieren.
- Hat einen **Namen**, der nach der Konvention immer mit einem **Großbuchstaben** beginnt, außerdem gibt es in Java-Namen **nie** Leerzeichen. Falls der Name aus mehreren Wörtern besteht, werden diese zusammen geschrieben, wobei jedes Wort mit einem Großbuchstaben beginnt, damit man das Ganze besser lesen kann.

Der Zustand (state) des Objekts

- Wird normalerweise am Anfang der Klasse deklariert.
- Die einzelnen Eigenschaften sind normalerweise « **private** » (privat) deklariert, was bedeutet, dass nur das Objekt darauf Zugriff hat. Die anderen Objekte des Systems können nicht auf den « internen Zustand » des Objektes zugreifen.
- Jede Eigenschaft hat einen **Namen**, der nach der Konvention mit einem **Kleinbuchstaben** beginnt.
- Jede Eigenschaft hat einen **Typ**. In diesem Falle repräsentiert **int** den « integer » Zahlentyp (0, 1, 2, …). Wir können auch beliebige andere Klassen als Typ der Eigenschaft nutzen.

Die Methoden (methods) des Objekts

- Methoden werden oft « **public** » (öffentlich) deklariert, sodass sie auch von den anderen Objekten in der Anwendung aufgerufen werden können. Sie können allerdings auch « **private** » sein, falls sie nur für das Objekt selbst sichtbar sein müssen (und nicht für die anderen Objekte).
- Jede Methode hat einen **Namen,** der stets mit einem **Kleinbuchstaben** beginnen sollte, dies ist ebenfalls Konvention.

- Jede Methode hat einen **return type** (Rückgabetyp). Die ist der Typ des Ergebnisses, dass die Methode zurückgibt. Gibt die Methode kein Ergebnis zurück, wird der Rückgabetyp als **void** (leer) definiert.
- Eine Methode kann keine, eins oder mehrere input **arguments** (Eingabeparameter) haben, die direkt hinter dem Methodennamen in Klammern deklariert werden. Jedes Argument hat einen **Namen** (welcher konventionell ebenfalls mit einem **Kleinbuchstaben** beginnt) und einen **Typ**.
- Methoden, die ein Ergebnis zurückgeben (also diejenigen, die einen Rückgabetyp definieren) müssen das **return (Rückgabe)** statement nutzen, um die Ergebnisse zurückzugeben.
- Da der Zustand des Objekts meistens private ist, definiert es für gewöhnlich sogenannte « **getter** » (*Ab*holer) Methoden, um **lesenden** Zugriff auf seinen internen Zustand (oder zumindest Teile davon) zu geben. Dies sind Methoden wie **getX()** und **getY()** in unserem Beispiel
    - Sie werden benannt mit: **get<Eigenschaftsname>**
    - Sie haben keinerlei Eingabeparameter
    - Ihr **return type** ist der Eigentschaftstyp
    - Im Grunde führen sie nur ein **return <eigenschaft>** aus.
- Parallel dazu definiert ein Objekt, das **schreibenden** Zugriff auf seinen internen Zustand (oder zumindest Teile davon) geben möchte, sogenannte « **setter** » (*Neu*setzer) Methoden. Dies sind Methoden wie **setX()** und **setY()** in unserem Beispiel.
    - Sie werden benannt mit: **set<Eigenschaftsname>**
    - Sie habe einen einzigen Eingabeparameter, mit dem gleichen Typ wie der **Eigenschaftstyp.**
    - Ihr **Rückgabetyp** ist **void.**
    - Im Grunde führen sie bloß **<Eigenschaft> = <Eingabe>** aus.

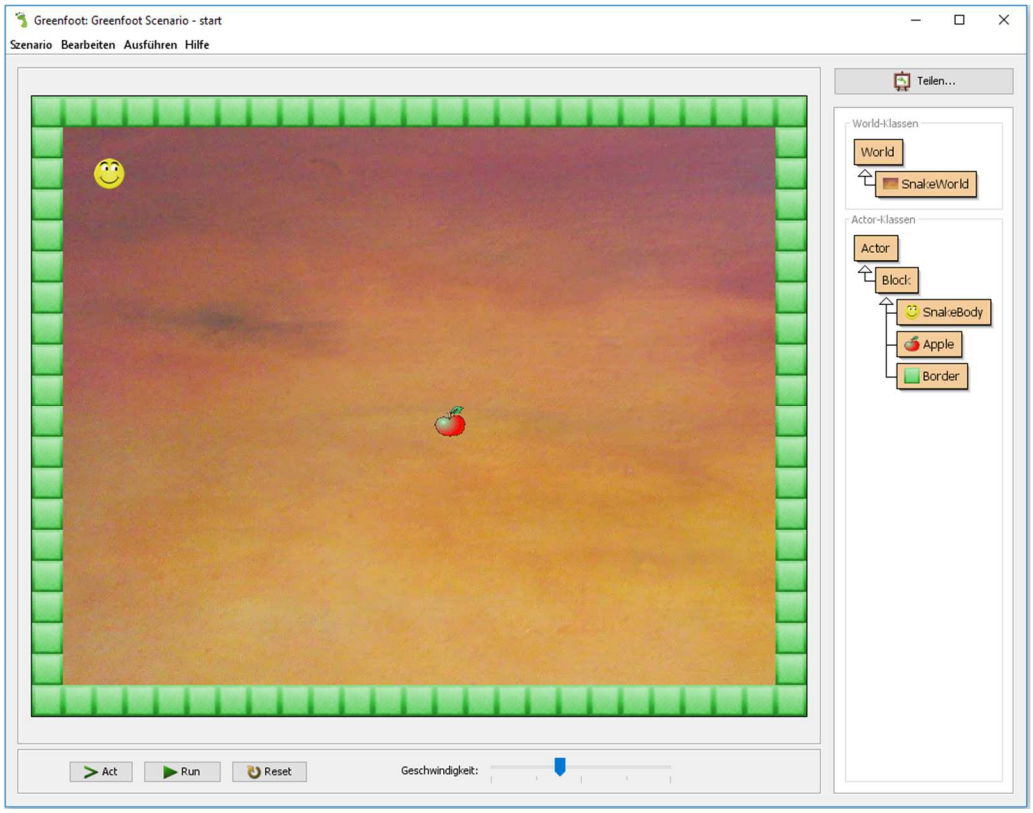
Die Konstruktoren (constructors)

- Konstruktor ähneln den Methoden, außer dass sie **keinen Rückgabetyp haben** und ihr Name immer der gleiche ist, wie der **Klassenname.**
- Ein Konstruktor erlaubt es einem, ein Objekt (oder auch eine „Instanz“) zu erzeugen. Dies geschieht mit einem Ausdruck wie **new GameElement(2, 3) ;**
    - Basierend auf unserem Beispiel würde dies in diesem Falle ein Objekt mit 2 und 3 als Werte für die Eigenschaften x und y.
- Man muss nicht immer einen Konstruktor in einer Klasse haben. Definiert man keinen eigenen Konstruktor, gibt es einen „default“ (Standard) Konstruktor, welcher keine Eingabeparameter hat und es einem erlaubt, Objekte mit einem Ausdruck wie **new GameElement();** zu instanziieren (eine Instanz zu erzeugen)
- Sobald man jedoch einen Konstruktor explizit definiert, existiert der Standard Konstruktor nicht mehr und kann dementsprechend nicht mehr aufgerufen werden.

4. Präsentation des « Bobby Snake » Spiels

Eine Anwendung in Greenfoot wird scenario genannt. Auf der Greenfoot Website finden sich noch viele weitere Szenarien: http://www.greenfoot.org/scenarios

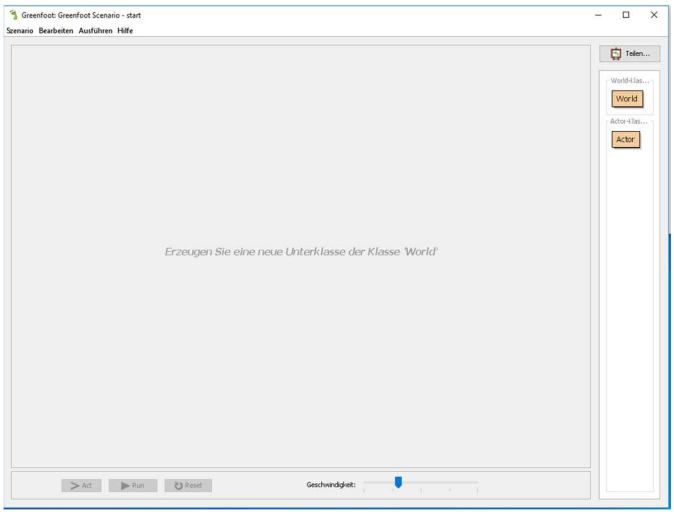
In diesem Workshop werden wir ein Szenario erschaffen, das wir « Bobby Snake » nennen. Du kannst das « Bobby Snake » Szenario mit einem Doppelklick auf die **project.greenfoot** Datei im **Bobby Snake** Ordner öffnen.



Das Ziel des Workshops ist das Entwickeln dieses Snake-Spiels. Du kannst Bobby, die Schlange, mit den Pfeiltasten bewegen und jedes Mal, wenn er einen Apfel ist, wächst Bobby. Doch Achtung, er kann nicht auf die Spielfeldbegrenzung gehen und auch wenn er sich selbst beißt, ist das Spiel vorbei!

5. Ein neues Szenario anfangen

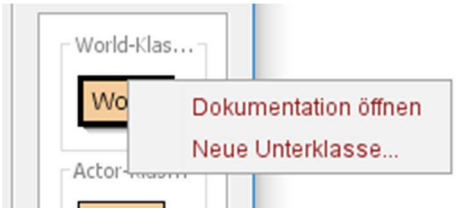
Wir werden mit einem leeren Szenario starten und Schritt für Schritt erklären, wie du das Bobby Snake Spiel erstellen kannst. Nach dem Start eines neuen Szenarios sieht der Greenfoot Bildschirm aus wie folgt:



Auf der rechten Seite des Bildschirms zeigt Greenfoot zwei wichtige Klassen:

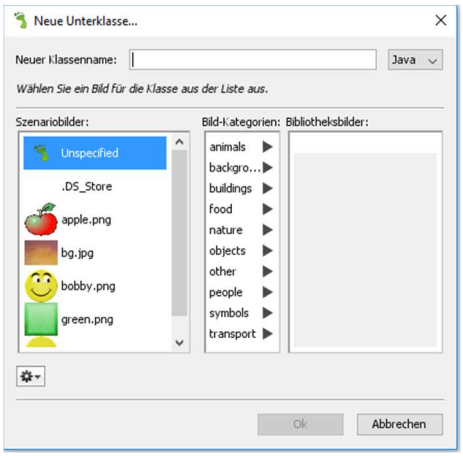
- **World**: Dies ist die « Welt » des Spiels. Das World Objekt wird später den Bildschirm zeichnen.
- **Actor**: Die verschiedenen Spielelemente, die sich später in dem Spiel tummeln, sind die **Actors** (Akteure). (Bobby, Äpfel, Begrenzungsblöcke).

6. Erschaffung der Welt



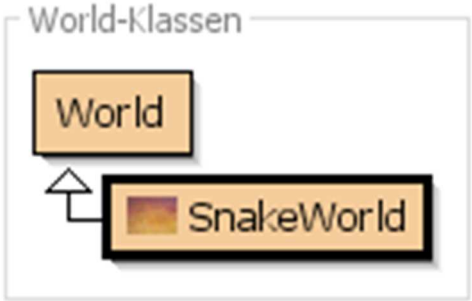
Mache einen « Rechtsklick » auf World, und wähle« Neue Unterklasse… »

Lass uns in dem Dialog Fenster den Klassennamen als **SnakeWorld** definieren.

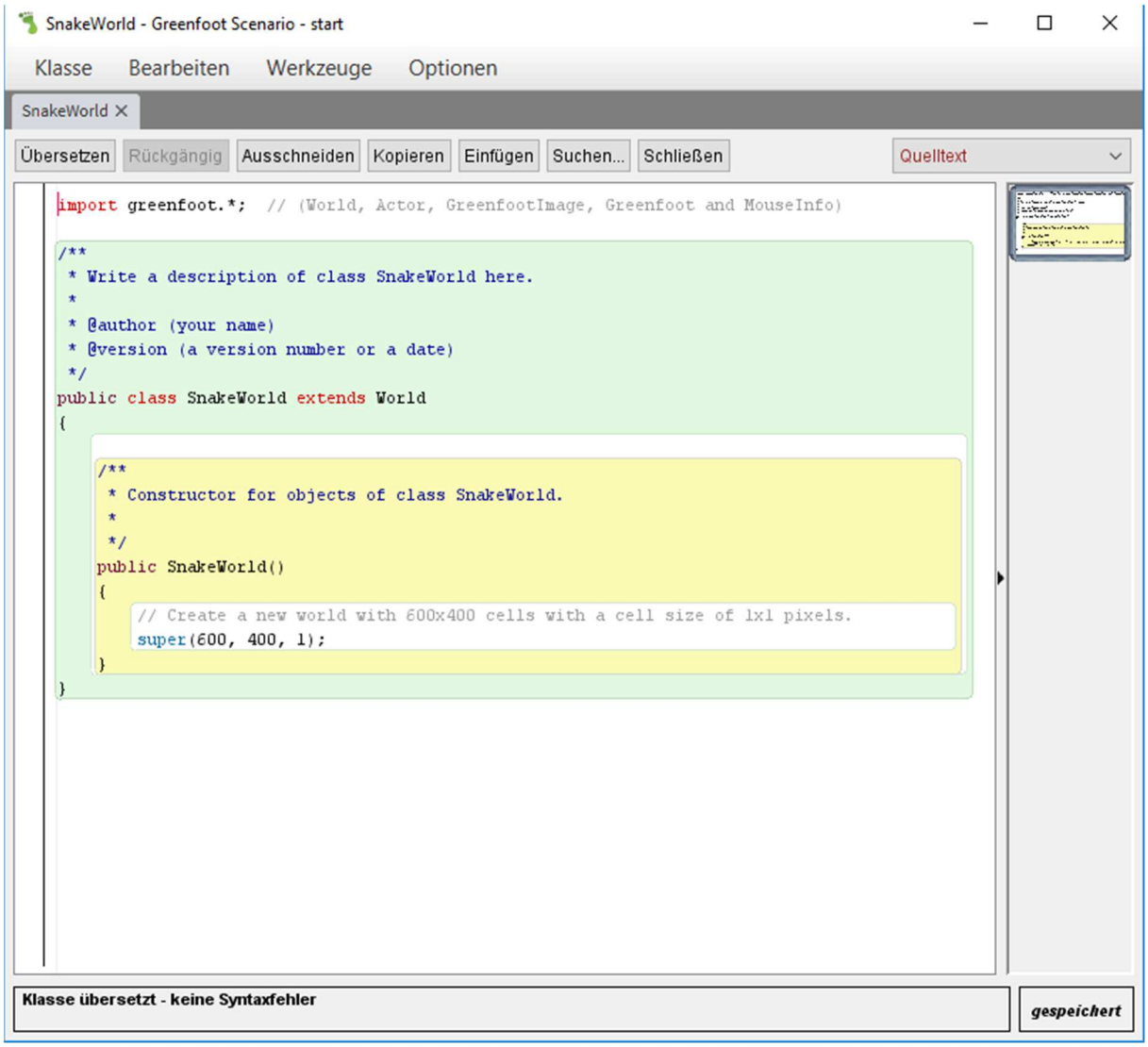


Wir können außerdem einen Hintergrund für unsere Welt wählen. Lass uns hier das Bild namens **bg.jpg** nehmen.

Nach dem Klick auf OK erzeugt Greenfoot eine neue Klasse, die eine « sub-class » (Unterklasse) von World ist.

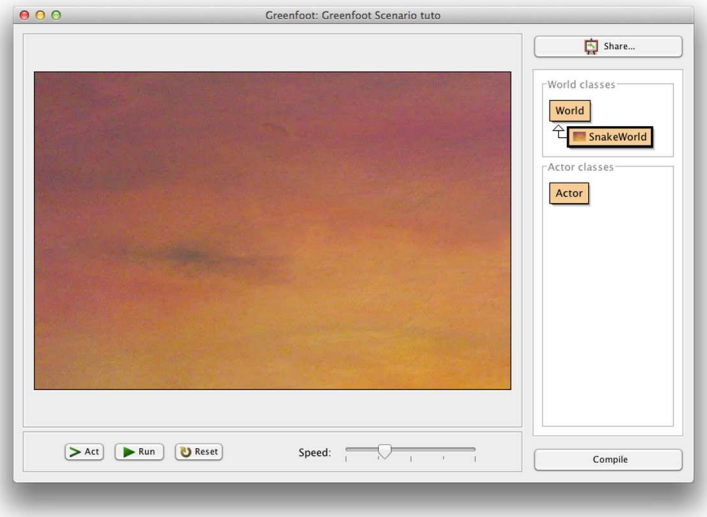


Bei einem Doppelklick auf diese neue Klasse öffnet Greenfoot einen « Java code editor ».



Wir werden gleich auf diesen Code zurückkommen, aber zuerst werden wir ihn einmal ausführen.

Um das zu tun, musste man in älteren Greenfoot-Versionen den Code zuerst per Hand « kompilieren », indem man auf einen « compile » Button klickte (Dieser befand sich unten rechts im Greenfoot Hauptfenster, siehe nebenstehende Abbildung).



Die **Kompilierung (compilation)** analysiert den Code, identifiziert mögliche Fehler, und erzeugt eine « executable » (Ausführbare), die es erlaubt, den Code auszuführen. Dieses kompilieren ist für jeden Java-Code zwingend notwendig, bevor er zur Anwendung kommen kann, Greenfoot führt dies allerdings mittlerweile automatisch aus, nachdem eine Klasse geändert und gespeichert wurde.

In unserem Beispiel stellt dieses Ausführen bloß ein Bild dar.

Within Greenfoot, classes displayed as **hatched** indicates classes that need to be (re)compiled. If you want to execute your code, you first need to do a compilation by clicking the « **compile** » button.

Nun zurück zu unserem Code:

```
public class SnakeWorld extends World
```

Der « extends World » Teil weist darauf hin, dass unsere SnakeWorld Klasse die Charakteristiken(Eigenschaften, Methoden) der **World** Klasse « **erbt**» (inherits). Diese Klasse wird von Greenfoot gestellt und erlaubt es, den Hauptbildschirm des Spiels zu erzeugen.

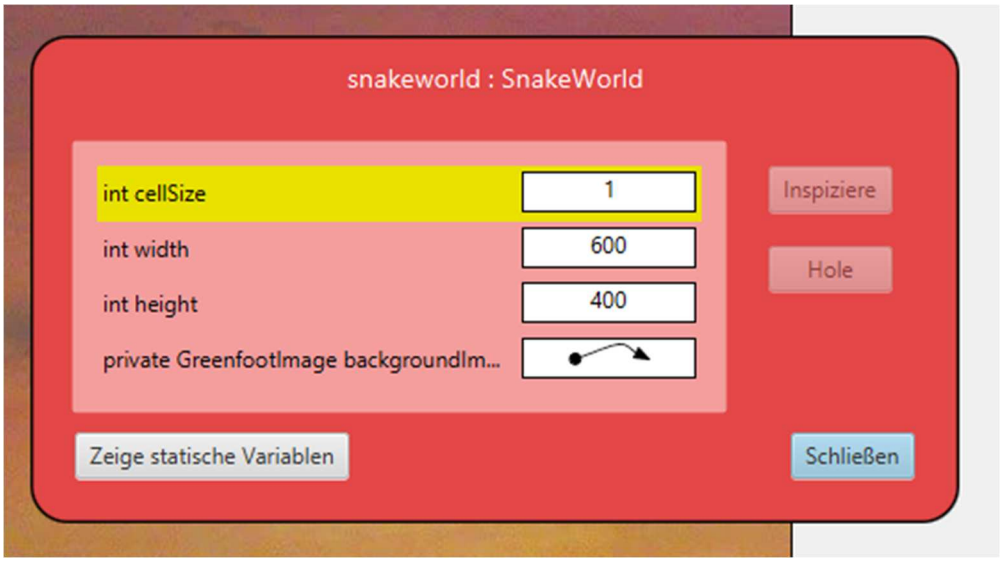
Als nächstes folgt ein « **constructor** » (Konstruktor). Dabei handelt es sich nicht um eine Methode (siehe Seite 4 „Die Konstruktoren“), da der Rückgabetyp fehlt und der Name gleich dem der Klasse ist.

```
public SnakeWorld() {
    super(600, 400, 1);
}
```

Die Aussage « **super(600, 400, 1)** » ruft tatsächlich einen Konstruktor auf, der in dem parent (Vorgänger, in diesem Falle World) definiert ist und es erlaubt, dass World Objekt zu « erzeugen », indem es ihm die Parameter gibt, die die Größe des Spielbildschirms definieren:

- Der erste Parameter entspricht der **Breite** des zu erschaffenden Bildschirms
- Der zweite Parameter entspricht der **Höhe** des zu erschaffenden Bildschirms
- Der dritte Parameter entspricht der Größe eines « Blocks », falls wir einen Bildschirm haben wollen, der aus Blöcken besteht. (andernfalls bleibt dieser Wert 1)

Mittels eines « Rechtsklicks » auf den Bildschirm des Spieles können wir das erzeugte Objekt auch « inspizieren » (Klicke auf « Inspizieren »). Wir können so sehen, dass das erzeugte Objekt mit den Parameter definiert wurde, die dem Konstruktor übergeben wurden.



Lass uns diesen Code nun so verändern, dass er einen Bildschirm erzeugt, der aus Blöcken der Größe 32x32 Pixel besteht und 25 Blöcke breit und 20 Blöcke hoch ist:

```
public SnakeWorld() {
    super(25, 20, 32);
}
```

Überprüfe die Änderung, indem die Welt « inspizierst ».

7. Das Erschaffen eines Blockes

Wir werden jetzt damit beginnen, die « Akteure », die Teil unseres Spiels sein werden, zu erschaffen.

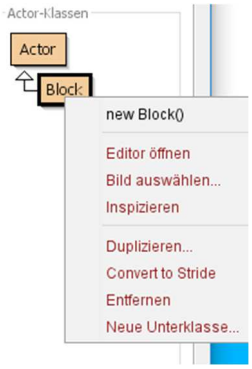
Um das zu tun, mache einen « Rechtsklick » auf die Actor-Klasse und wähle « Neue Unterklasse… ».

Wir werden mehrere Arten von Blöcken erschaffen, die in unserem Spiel Verwendung finden. Deshalb erschaffen wir erstmal eine Klasse namens **Block,** die eine Unterklasse von **Actor** sein wird (und daher alle Charakteristiken der Greenfoot Actor Klasse erbt).

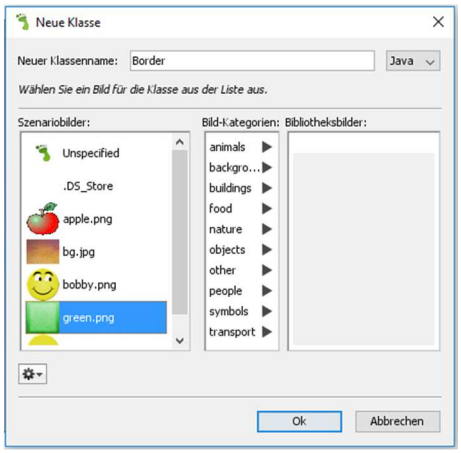
Vorerst können wir die Klasse leer lassen: Mache einen Doppelklick auf sie, um den von Greenfoot erzeugten Code zu sehen und die act()-Methode zu unterdrücken (die brauchen wir nicht).

```
public class Block extends Actor
{

}
```

8.Erschaffen eines « border » (Begrenzungs-) Blocks

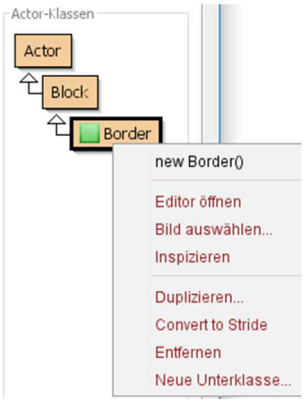


Lass uns nun eine Unterklasse unserer soeben erzeugten « Block» -Klasse erschaffen (Rechtsklick auf Block, dann « Neue Unterklasse… »).



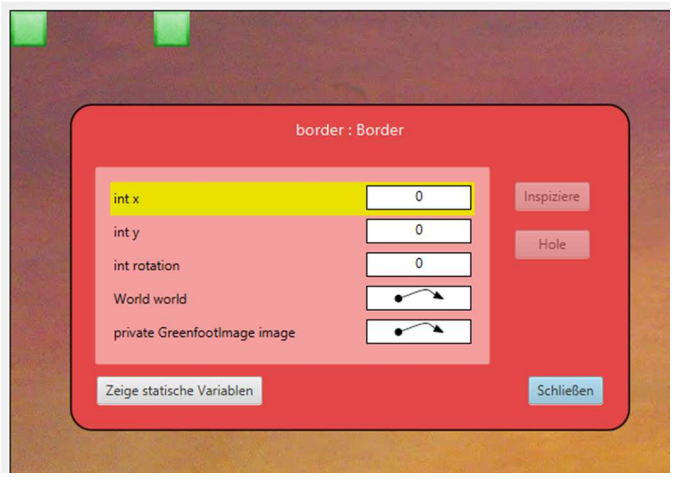
Nenne die neue Klasse « **Border**» und wähle das Bild « green.png ».

Eine neue Klasse « Boder» erscheint unter den Actor und Block Klassen.



Mittels eines Rechtsklicks auf die Border Klasse können wir die « new Border() » Option wählen, die uns erlaubt ein Objekt (oder eine « Instanz ») unserer Klasse zu erzeugen.

Mit Hilfe der Maus kannst du dieses Objekt nun irgendwo auf dem Spielbildschirm platzieren (In einer der Zellen auf dem Bildschirm).



Dies kannst du jederzeit tun, wenn du neue Blöcke erzeugen möchtest. Danach kannst du die « Inspizieren »-Funktion benutzen, um den Zustand eines jeden davon zu beobachten.

In seinem Zustand kannst du die Position des Blocks (x,y) sehen.

Du kannst sehen, dass die Position des Blocks oben links auf dem Bildschirm (0,0) ist und die **x-Position** wächst, wenn man nach rechts geht, und die **y-Position**, wenn man nach unten geht.

Die Blockpositionen sind also so organisiert, wie die folgende Tabelle veranschaulicht:

(0,0)	(1,0)	(2,0)	…
(0,1)	(1,1)	(2,1)	…
(0,2)	(1,2)	(2,2)	…
…	…	…	…

9. Die Begrenzung der Spielwelt anzeigen

Um in unserer Welt eine Begrenzung anzuzeigen, klicke auf die SnakeWorld-Klasse, um sie zu öffnen, und öffne den folgenden Code:

```
public SnakeWorld()
{
    super(25, 20, 32);
    addObject(new Border(), 0, 0);
}
```

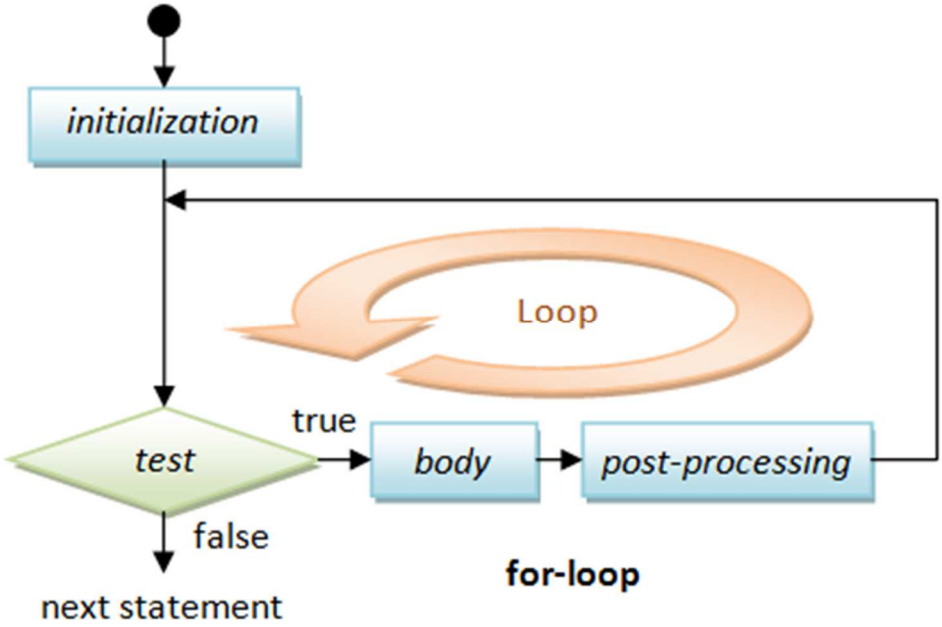
Nachdem du die Klasse wieder schließt, sollte Greenfoot den Code kompilieren und du solltest sehen, wie in der linken oberen Ecke ein Block auftaucht.

Die **addObject** Methode ist eine von der World Klasse vererbte Methode, die es erlaubt, einen Actor an einer, als Parameter gegebenen, (x,y) Position darzustellen. Das « **new** » direkt vor dem Namen einer Klasse erlaubt es, ein neues Objekt (oder Instanz) einer Klasse zu erzeugen. Die entspricht einem Aufruf des Konstruktors der entsprechenden Klasse.
Wenn wir entlang des kompletten Bildschirmrandes Blöcke darstellen wollen, wäre es etwas umständlich, für jeden Block eine Programmzeile zu schreiben. Aus diesem Grund werden wir einen sehr nützlichen Befehl dafür nutzen: Die « **for**-Schleife » aus Java.

Die Syntax einer « **for**-Schleife **»** ist die folgende:

```
for ( initialization (Initialisierung) ; Test ; post-processing (Nachverarbeitung)
) {
    body (Körper) ;
}
```

Das folgende Schema stellt die Ausführung dieser Schleife dar:




```
public SnakeWorld()
{
    super(25, 20, 32);

    for (int x = 0; x < getWidth(); x ++ ) {
        addObject(new Border(), x, 0);
        addObject(new Border(), x, getHeight()-1);
    }

    for (int y = 0; y < getHeight(); y ++) {
        addObject(new Border(), 0, y);
        addObject(new Border(), getWidth()-1, y);
    }

}
```

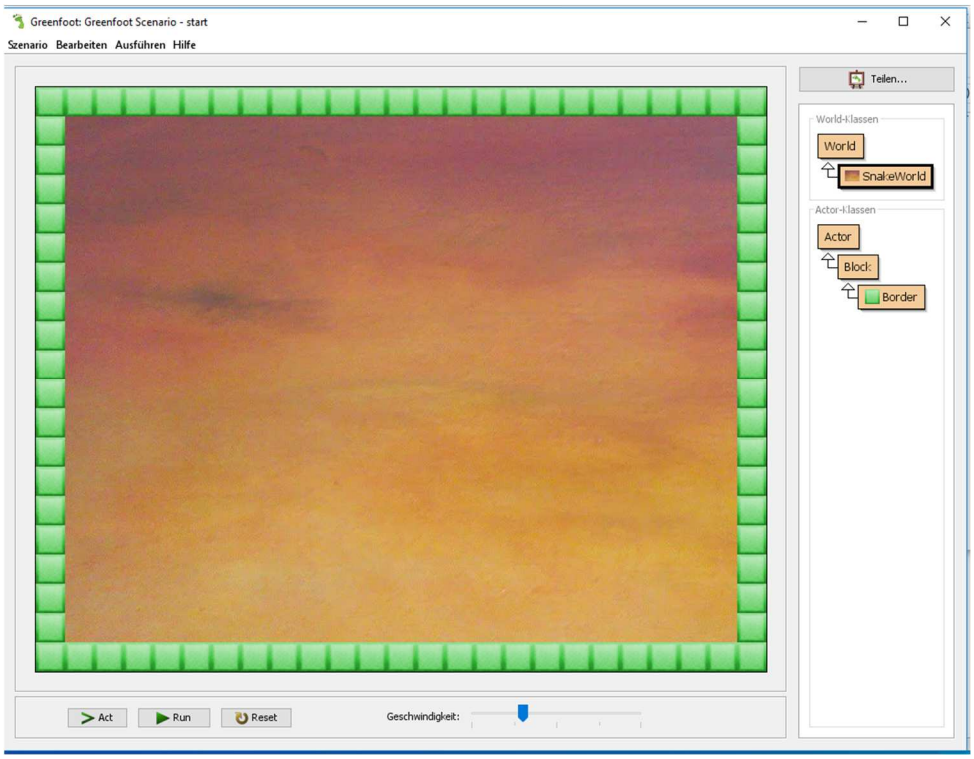
Um die Begrenzung unseres Spieles darzustellen, werden wir eine Schleife schreiben, die eine Variable x von 0 bis zum Breite des Spiels definiert. Diese Schleife nutzen wir, um die Blöcke am oberen und unteren Rand darzustellen (Bedenke: Da der Index der ersten Spalte 0 ist, ist der Index der letzten Spalte die Anzahl der Spalten – 1).

Als nächstes wird eine zweite Schleife eine Variable y von 0 bis zur Höhe des Spiels definieren. In dieser Schleife werden wir die Blöcke am linken und am rechten Rand darstellen.

Nachfolgend ein paar weitere Erklärungen:

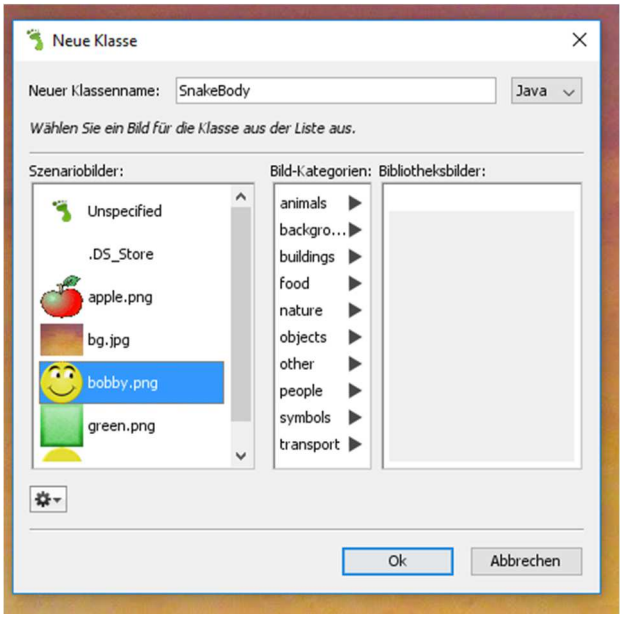
- Variablen in Java sind immer **typed (typisiert)**, in diesem Falle ist der Typ der Variablen x und y **int.** Wir werden diesen Typen oft nutzen, denn er repräsentiert Ganzzahlen (0, 1, 2, …)
- Die Aussage **variable = <ausdruck>;** Diese erlaubt es, den Wert des Ausdrucks der Variable zuzuordnen. Also erlaubt es die Aussage **x = 0,** die variable **x** mit dem Wert **0** zu initialisieren.
- Alle Aussagen in Java enden stets mit einem Semikolon (**;**)
- Die Aussage **x++ ;** Diese erlaubt es, 1 zu der Variable hinzu zu addieren. Sie entspricht der Anweisung **x = x + 1 ;**
- Die Methoden **getWidth()** und **getHeight()** sind aus der **World**–Klasse vererbt und ermöglichen es, die Breite und Höhe der Welt als **int** zu erhalten.

Nachdem der Code kompiliert wurde, solltest du den folgenden Bildschirm sehen:



10. Bobbys Kopf darstellen

Erschaffe eine neue Unterklasse unserer Block Klasse.



Nenne sie **SnakeBody** und wähle das Bild von Bobby, **bobby.png**.

Die Schlange in unserem Spiel wird eine « list » (Liste) von SnakeBody-Objekten sein. Um das zu erreichen, wird die **SnakeWorld** eine Eigenschaft definieren, sodass die SnakeBody-Liste Teil seines « state » (Zustand) wird.

Öffne die **SnakeWorld**–Klasse und füge die folgende Definition hinzu:

```
public class SnakeWorld extends World
{
    private LinkedList<SnakeBody> snake = new LinkedList<SnakeBody>();
```

Diese Definition erzeugt eine leere **SnakeBody**–Liste .

Um die **LinkedList**–Klasse von Java nutzen zu können, musst du außerdem noch die folgende Anweisung am Anfang der **SnakeWorld**–Klasse hinzufügen:

```
import greenfoot.*;
import java.util.*;
```

Lass uns als nächstes das erste Element zu dieser Liste hinzufügen, direkt im Konstruktor der SnakeWorld:

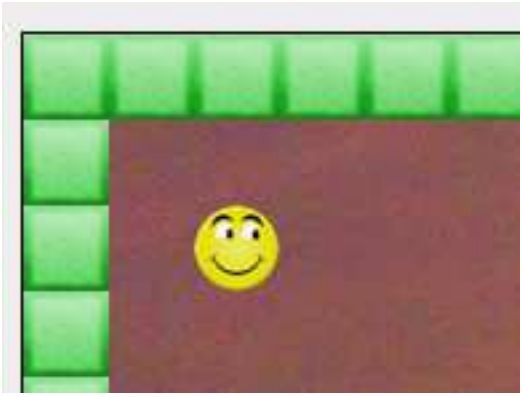
```
public SnakeWorld()
{
    super(25, 20, 32);

    SnakeBody body = new SnakeBody();
    snake.add(body);
    addObject(body, 2, 2);
```

Die erste Aussage erzeugt ein neues **SnakeBody**–Objekt.

Die zweite nutzt die **add()** Methode auf der **snake** Liste und ermöglicht es so, ein neues Element am Ende der Liste hinzuzufügen (dieses wird als Eingabeparameter gegeben).

Und die dritte Aussage stellt mittels der **addObject ()** Methode, die wir bereits benutzt haben, um die Begrenzung darzustellen, das neue Element in der Welt an der Stelle (2,2) dar.

Schließt du die Klasse, solltest du Bobbys Kopf an der Stelle (2,2) sehen.



11. Bobbys Bewegung

Um zu wissen, in welche Richtung sich Bobby bewegt, werden wir zwei neue Eigenschaften hinzufügen, eine für die Richtung entlang der x-Achse und eine für die Richtung entlang der y-Achse.

```
public class SnakeWorld extends World
{
    private LinkedList<SnakeBody> snake = new LinkedList<SnakeBody>();
    private int dx = 1;
    private int dy = 0;
```

Damit Bobby sich am Anfang nach rechts bewegt, wird die Variable dx mit 1 initialisiert, und die Variable dy mit 0.

Lass und nun eine Methode **act()** zu unserer **SnakeWorld** hinzufügen. Diese Methode wird kontinuierlich von Greenfoot aufgerufen, um das Spiel voranschreiten zu lassen. Es ruft sie auf die **World** auf und auf alle **Actors** (Akteure), die der **World** hingefügt wurden.

Nachfolgend eine erste Version der **act()** Methode.

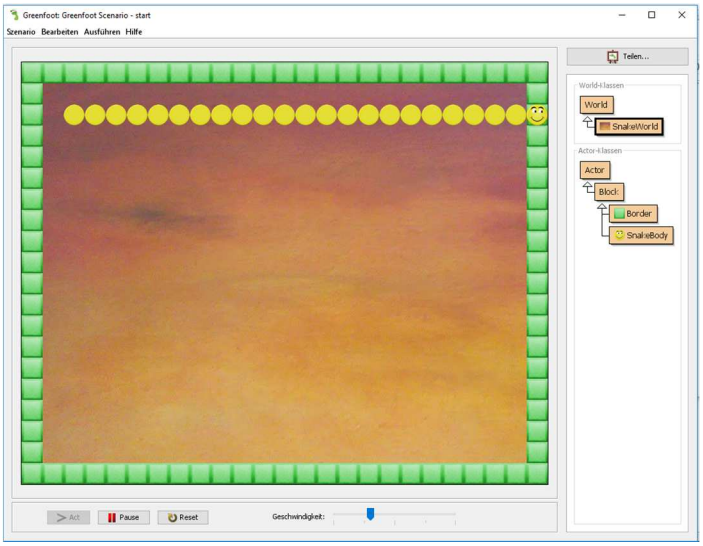
```
public void act()
{
    //Ersetzen des Bildes des vorherigen Elements
    SnakeBody head = snake.getLast();
    head.setImage("tail.png");

    //Erzeugen eines neuen Elements
    SnakeBody newHead = new SnakeBody();
    int newHeadX = head.getX()+dx;
    int newHeadY = head.getY()+dy;

    //Hinzufügen des neuen Elements sowohl zu der Welt, als auch zu der Liste
    addObject(newHead, newHeadX, newHeadY);
    snake.add(newHead);
}
```

In dieser Version arbeiten wir mit dem derzeitigen Kopf (welcher das letzte Element in unserer Liste ist). Die **getLast()** Methode auf einer **LinkedList** gibt das letzte Element dieser Liste zurück.

Als nächstes ändern wir das Bild dieses Element zu einem Bild des Schwanz-Elements der Schlange.

Nun erschaffen wir mit dem Statement **new SnakeBody()** einen neuen Kopf und verarbeiten die Position dieses neuen Kopfes, basierend auf der aktuellen Position und der aktuellen Bewegungsrichtung (dx und dy).

Abschließend fügen wir den neuen Kopf an der Position in der Welt hinzu und fügen ihn auch zu der Liste der Schlangen-Elemente hinzu.



Teste das Spiel, indem du auf « **run** » klickst (um so Greenfoot aufzufordern, kontinuierlich die act-Methode aufzurufen).

Die Schlange bewegt sich tatsächlich nach rechts, allerdings wächst sie auch mit jedem Schritt, bis sie die Begrenzung berührt.

12. Begrenzen der Schlangen-Länge

Um die Länge der Schlange zu begrenzen, werden wir der Welt eine neue Eigenschaft hinzufügen, die anzeigt, wie viele Elemente der Schlange noch erzeugt werden müssen:

```
public class SnakeWorld extends World
{
    private LinkedList<SnakeBody> snake = new LinkedList<SnakeBody>();
    private int dx = 1;
    private int dy = 0;
    private int tailCounter = 5;
```

In der **act()** Methode werden wir, nachdem wir den neuen Kopf erzeugt haben, den Wert dieses Zählers testen:

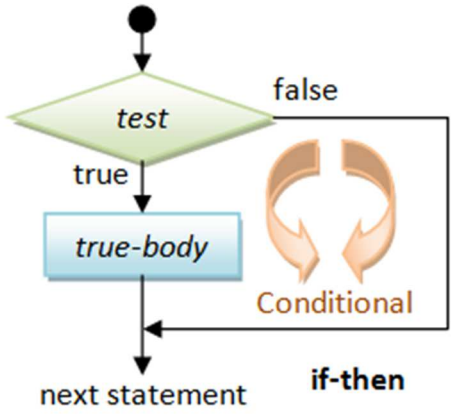
- Falls er gleich 0 ist, werden wir das letzte Element der Schlange (welches tatsächlich das erste Element der Liste ist, da wir den neuen Kopf immer ans Ende der Liste anhängen) ausblenden.
- Anderenfalls reduzieren wir den Zähler um 1

Um einen solchen Test durchzuführen, nutzen wir ein weiteres sehr wichtiges Java Statement: Das **if statement (if-Aussage; „**if“ bedeutet „falls“**)**.

Die Syntax einer **if-Aussage** ist wie folgt:

```
if ( condition (Bedingung) ) {
        true-body („wahr“-Anweisung) ;
}
```

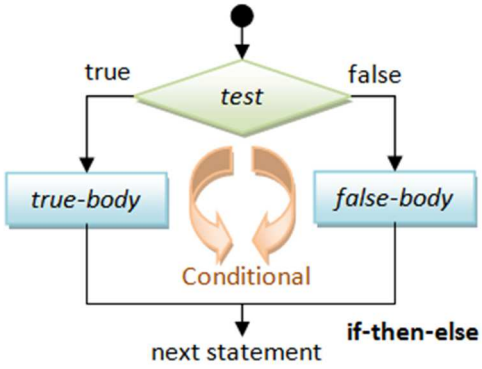
Und seine Ausführung hängt von der **Bedingung** ab:



Eine weitere Form der if-Aussage ist das **if-then-else,** welches es erlaubt, zwei Verhaltensweisen zu definieren:

```
if ( condition (Bedingung) ) {
   true-body („wahr“-Anweisung) ;
} else {
   false-body („falsch“-Anweisung) ;
}
```

Auch hier hängt seine Ausführung von der **Bedingung** ab **:**

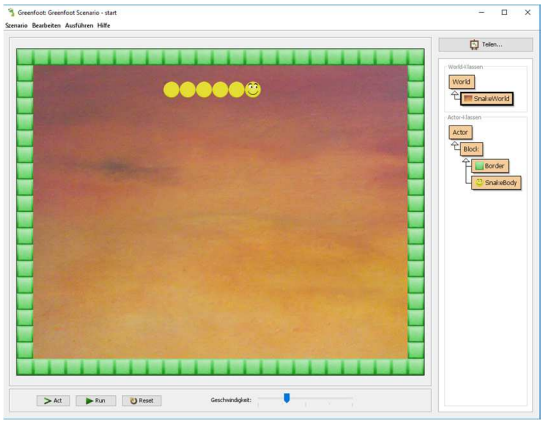


Hier ist der Code, den wir zur **act()** Methode hinzufügen werden:

```
public void act()
{
    //Ersetzen des Bildes des vorherigen Elements
    SnakeBody head = snake.getLast();
    head.setImage("tail.png");

    //Erzeugen eines neuen Elements
    SnakeBody newHead = new SnakeBody();
    int newHeadX = head.getX()+dx;
    int newHeadY = head.getY()+dy;

    //Hinzufügen des neuen Elements sowohl zu der Welt, a
    addObject(newHead, newHeadX, newHeadY);
    snake.add(newHead);

    if (tailCounter == 0){
        SnakeBody tail = snake.removeFirst();
        removeObject(tail);
    } else {
        tailCounter--;
    }
}
```

Die **removeFirst()** Methode auf einer **LinkedList** entfernt das erste Element der Liste und gibt dieses zurück.
Die **removeObject()** Methode, vererbt von World, entfernt den Akteur, der ihr als Parameter gegeben wird aus der Welt (und blendet ihn dementsprechend aus dem Bildschirm aus).

tailCounter-- ; entspricht
tailCounter = tailCounter -1 ;



Für die ersten 5 Aufrufe der **act()** Methode ist der **tailCounter** noch nicht gleich 0, die Schlange wächst also und der **tailCounter** wird jedes Mal um 1 reduziert.

Sobald jedoch der **tailCounter** 0 erreicht, hört die Schlange auf zu wachsen, da wir nun jedes Mal, wenn wir einen Kopf hinzufügen, ein Element entfernen.

13. Ändern der Richtung

Nun sollten wir uns darum kümmern, dass die Schlange mittels eines Tastendrucks die Bewegungsrichtung ändern kann.

Zu diesem Zwecke werden wir in der **SnakeWorld** Klasse eine neue Methode erschaffen:

```
private void changeDirection() {
    if (Greenfoot.isKeyDown("left") && dx == 0 ) {
        dx = -1;
        dy = 0;
    } else if (Greenfoot.isKeyDown("right") && dx == 0 ) {
        dx = 1;
        dy = 0;
    } else if (Greenfoot.isKeyDown("down") && dy == 0 ) {
        dx = 0;
        dy = 1;
    } else if (Greenfoot.isKeyDown("up") && dy == 0 ) {
        dx = 0;
        dy = -1;
    }

}
```

Die Aussage **Greenfoot.isKeyDown(« key »)** ermöglicht es und zu testen, ob eine Taste gedrückt wird.

Die Richtung nach links oder rechts ändern zu können setzt auch voraus, dass die Richtung nicht schon links oder rechts ist, da sich Bobby nicht gegen sich selbst wenden kann!

Wir benutzen Ausdrücke wie den folgenden:

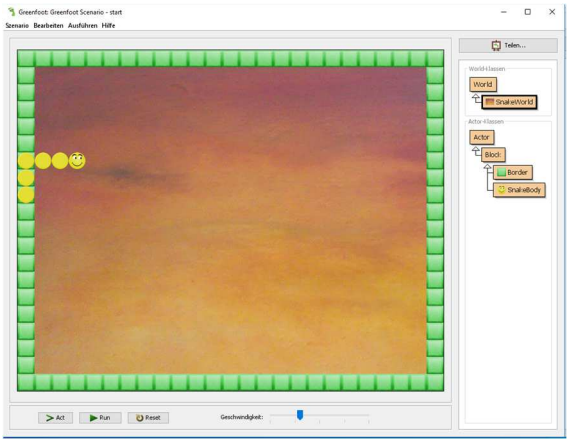
Greenfoot.isKeyDown(“left“) && dx == 0

Was sich übersetzen lässt mit « Die linke Pfeiltaste ist gedrückt **und** dx ist gleich 0 ». Der Operator **&&** entspricht dem **und**, und der Operator **==** vergleicht zwei Werte, um zu sehen, ob sie gleich sind.

Es bleibt nur noch, die Methode am Anfang unserer act() Methode aufzurufen.

```
public void act()
{
    changeDirection();
```

Teste deine Änderungen.



Jetzt kannst du Bobby bewegen, indem du die Pfeiltasten der Tastatur verwendest.

Allerdings kann Bobby momentan überall hingehen, sogar auf die Begrenzung und durch sich selbst.

14. Kollisionsmanagement

Bevor wir uns um die Kollisionen kümmern, fügen wir eine Eigenschaft **dead** zu unserer **SnakeWorld** Klasse hinzu. Diese wird es uns ermöglichen, das Spiel zu beenden, wenn Bobby mit einem Hindernis zusammenstößt.

```
public class SnakeWorld extends World
{
    private LinkedList<SnakeBody> snake = new LinkedList<SnakeBody>();
    private int dx = 1;
    private int dy = 0;
    private int tailCounter = 5;
    private boolean dead = false;
```

Diese Eigenschaft ist ein « **boolean** » (boolescher). Dies ist ein einfacher und doch sehr wichtiger Datentyp in Java. Er kann nur zwei Werte annehmen.

- **true (wahr)**
- **false (falsch)**

Wir können jetzt die **act()** Methode so ändern, dass sie nichts tut, wenn Bobby **dead** ist.

Wenn **dead** den Wert **true** hat, gibt die Methode sofort zurück und das Spiel endet.

```
public void act()
{
    if (dead) {
        return;
    }
    changeDirection();
```

Lass uns außerdem eine **dead()** Methode zur SnakeWorld hinzufügen, um den Wert von **dead** zu ändern:

```
public void dead() {
    dead = true;
}
```

Lass uns nun einen Blick darauf werfen, wie man « Kollisionen » verarbeitet. In der **act()** Methode werden wir, *genau bevor wir den neuen Kopf darstellen*, überprüfen, ob sich bereits ein Block an dieser Stelle befindet. Falls dem so ist, werden wir eine **collision** Methode auf den Block ausführen.

```
//Erzeugen eines neuen Elements
SnakeBody newHead = new SnakeBody();
int newHeadX = head.getX()+dx;
int newHeadY = head.getY()+dy;

List<Block> blocks = getObjectsAt(newHeadX, newHeadY, Block.class);
for (Block block: blocks){
    block.collision(this);
}

//Hinzufügen des neuen Elements sowohl zu der Welt, als auch zu der Liste
addObject(newHead, newHeadX, newHeadY);
snake.add(newHead);
```

Die **getObjectsAt()** Methode, geerbt von der **World** Klasse, gibt die Liste der Blöcke an der übergebenen **x, y** Position, die von dem Typ sind, der als dritter Parameter übergeben wurde (in unserem Falle **Block.**class), zurück.

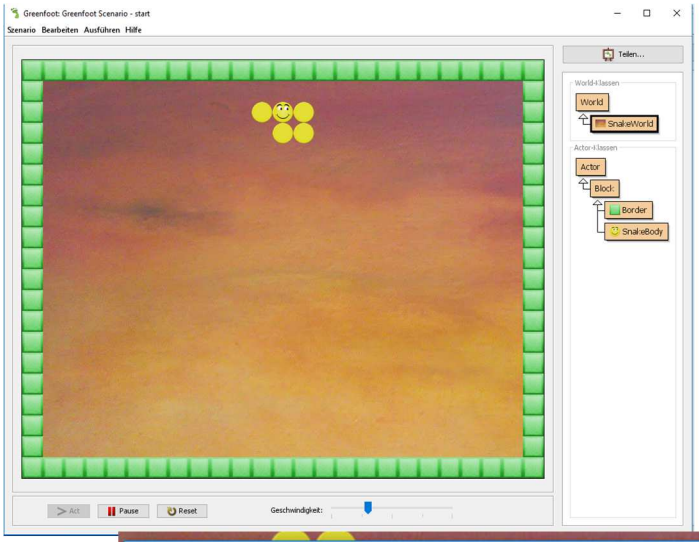
Diese Methode gibt also eine Liste von Blöcken zurück. Das folgende **for**-Statement erlaubt es, eine « Schleife » über alle Blöcke der Liste auszuführen und die **collision**() Methode auf jeden davon auszuführen.
Das « **this** », das als Parameter übergeben wir, erlaubt es, das Objekt selbst (in unserem Fall die **SnakeWorld**) an die aufgerufene Methode zu übergeben.

Zu diesem Zeitpunkt wird das *collision* im Code rot unterstrichen und wenn man die Codeansicht verlässt wird die **SnakeWorld** Klasse rot schraffiert. Dies liegt daran, dass die Klasse nicht kompiliert werden kann, da die **collision()** Methode in der **Block** Klasse noch nicht existiert.

Öffne die Block Klasse und füge die Kollisionsmethode hinzu:

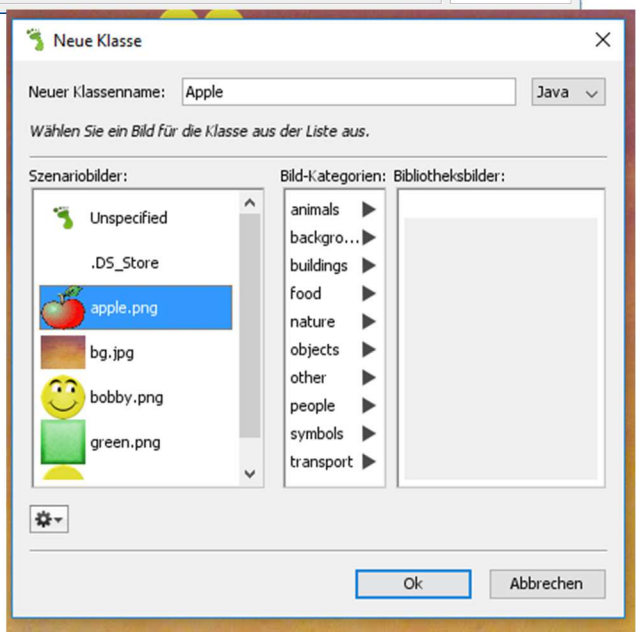
```
public class Block extends Actor
{

    public void collision(SnakeWorld world) {
        world.dead();
    }
}
```

Diese Methode ruft einfach **dead()** auf die Welt auf, was Spiel sofort beendet.



Teste die Änderungen.

Das Spiel endet, sobald Bobby auf die Begrenzung geht, oder sich selbst in den Schwanz beißt.



15. Einen Apfel hinzufügen

Lass uns nun eine neue Unterklasse von **Block** hinzufügen und sie **Apple** nennen. Wähle das **apple.png** Bild.

Öffne die neue **Apple** Klasse und lösche die **act()** Methode.

```
public class Apple extends Block
{

}
```

Kommen wir zurück zur **SnakeWorld** Klasse. Nach der Initialisierung des Spiels (im Konstruktor), nachdem wir Bobby erschaffen haben, erschaffen wir auch einen Apfel und positionieren ihn **zufällig** im Spiel.

```
public SnakeWorld()
{
    super(25, 20, 32);

    SnakeBody body = new SnakeBody();
    snake.add(body);
    addObject(body, 2, 2);

    Apple apple = new Apple();
    addObject(apple,
        Greenfoot.getRandomNumber(getWidth()-2)+1,
        Greenfoot.getRandomNumber(getHeight()-2)+1);
```

Der Apfel wird mit dem **new Apple()** Statement erzeugt. Danach wird es mit **addObject()** von Greenfoot zum Spiel hinzugefügt. Seine Position x, y wird mit einer anderen Greenfoot-Methode, **getRandomNumber()**, definiert. Diese Methode nimmt eine Zahl als Parameter und gibt eine zufällige Zahl zwischen 0 und der gegebenen Zahl zurück. Die Zahlen, die an diese Methode weitergegeben werden sind die Breite und die Höhe des Spielfeldes minus 2 (denn wir zählen die Ränder nicht mit). Danach zählen wir noch 1 dazu, damit wir nicht auf dem linken bzw. oberen Rand landen.

Teste deine Änderungen. Versuche, einen Apfel zu sammeln… Was passiert?

Das Spiel endet! Wir haben unser Spiel schließlich so programmiert, dass es endet sobald die Schlange mit einem Block zusammenstößt… und der Apfel ist ein Block!

16. Kollision mit einem Apfel

Wir werden jetzt die **collision** Methode in der **Apple** Klasse « **überschreiben (override)** ».

In Java können wir eine geerbte Methode **überschreiben,** um ein anderes Verhalten in einer Unterklasse zu definieren.

```
public class Apple extends Block
{
    public void collision(SnakeWorld world) {
        world.grow(2);

        setLocation(
            Greenfoot.getRandomNumber(getWorld().getWidth()-2)+1,
            Greenfoot.getRandomNumber(getWorld().getHeight()-2)+1);
    }
}
```

Wenn die Schlange mit einem Apfel kollidiert, sollten wir nicht die **dead()** Methode aufrufen, sondern lieber die Welt dazu auffordern, die Schlange zu verlängern (um bspw. 2 Elemente).

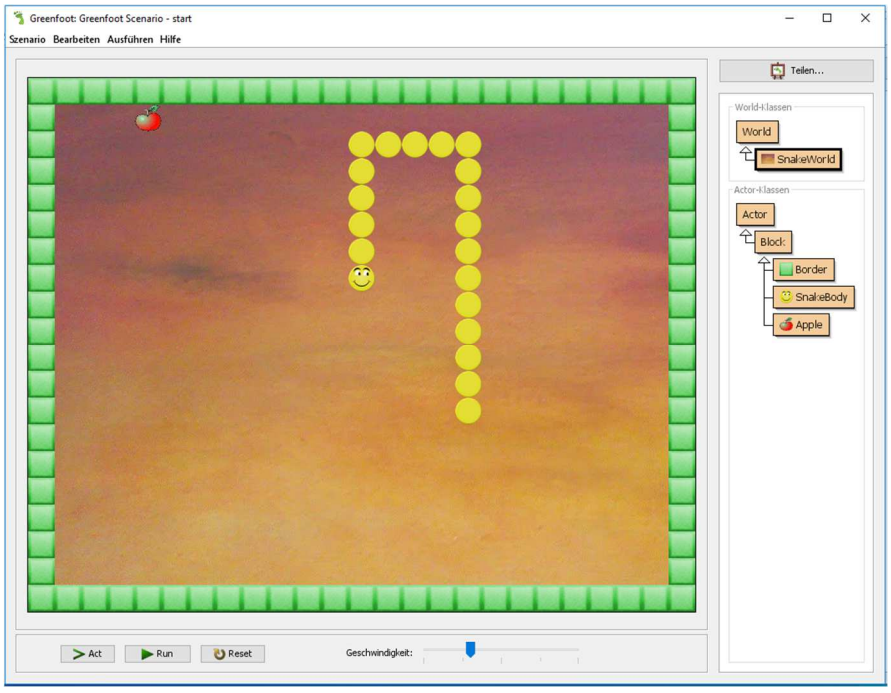
Außerdem sollte der Apfel woanders auf dem Spielbildschirm hinbewegt werden.

Es bleibt nur noch, eine **grow()** Methode in der **SnakeWorld** Klasse einzufügen:

```
public void grow(int i) {
    tailCounter = tailCounter + i;
}
```

Dies muss bloß die gegebene Zahl zum **tailCounter** hinzufügen.

Teste erneut. Nun wächst die Schlange jedes Mal, wenn sie einen Apfel isst!



17. Töne hinzufügen

Um unser Spiel weiter zu verbessern fügen wir jetzt nur Töne hinzu, die gespielt werden, wenn Bobby einen Apfel isst, oder stirbt.

Um einen Ton abzuspielen, wenn Bobby einen Apfel isst, brauchst du bloß den folgenden Code in die Kollisionsmethoden der **Apple** Klasse einzufügen:

```
public void collision(SnakeWorld world) {
    Greenfoot.playSound("slurp.mp3");
    world.grow(2);
    setLocation(
        Greenfoot.getRandomNumber(getWorld().getWidth()-2)+1,
        Greenfoot.getRandomNumber(getWorld().getHeight()-2)+1);
}
```

Probiere es aus!

Die **playSound()** Methode nimmt den Namen einer Tondatei als Eingabeparameter. Diese Datei muss sich im **sounds** Ordner in deinem Greenfoot Projektordner befinden.

Füge außerdem noch einen weiteren Ton hinzu, der abgespielt wird, wenn Bobby stirbt, indem du den folgenden Code in die Kollisionsmethode der **Block** Klasse einfügst:

```
public void collision(SnakeWorld world) {
    Greenfoot.playSound("dead.mp3");
    world.dead();
}
```

Probiere es aus!

18. Ende

Herzlichen Glückwunsch! Du hast dein erstes Greenfoot Spiel fertiggestellt!

Du kannst jetzt damit anfangen, neue Funktionen einzubauen, oder ein ganz neues Spiel zu erschaffen!