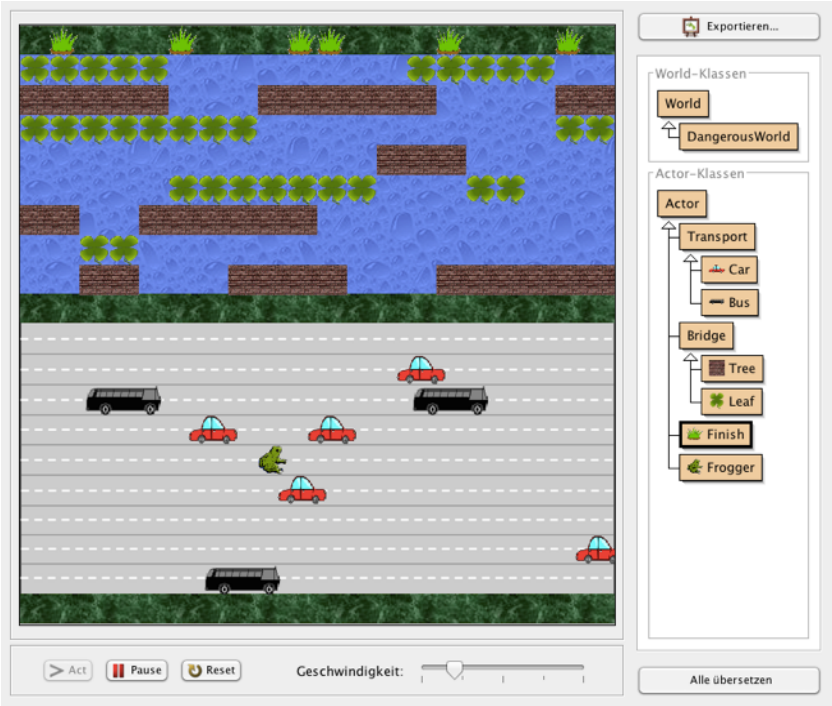
Frogger



Solltest du das Spiel **Frogger** nicht kennen, mache dich auf Youtube damit bekannt.

Wir wollen das Spiel von Grund auf in Greenfoot entwickeln. Lies Dir diese Anleitung genau durch und führe die nötigen Schritte selbst parallel am Rechner durch.

Eine Welt erstellen

Starte Greenfoot und erstelle ein neues Szenario. Erstelle eine neue Unterklasse von Welt und nenne sie **DangerousWorld.** Lade als Bild das Hintergrundbild **background.jpg** von der Festplatte. Es ist 600x600 Pixel groß.

Hinweis: Beachte, dass Klassennamen immer mit einem Großbuchstaben anfangen und aus einem zusammenhängenden Wort bestehen müssen. Üblicherweise beginnt man jedes Teilwort wiederum mit einem Großbuchstaben.

Klickst du nun auf **Alle Übersetzen,** so wird das Bild bereits in den Hintergrund der Welt geladen, allerdings passen die Dimensionen noch nicht. Doppelklicke auf **DangerousWorld.** Das öffnet den Editor, mit dem du die Klasse bearbeiten kannst. Schauen wir uns den Quelltext genauer an:

```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Write a description of class DangerousWorld here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class DangerousWorld extends World
{

    /**
     * Constructor for objects of class DangerousWorld.
     *
     */
    public DangerousWorld()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
    }
}
```

Es handelt sich hierbei um „ganz normalen“ Java-Code. Damit alle Features von Greenfoot auch funktionieren müssen diese erst einmal geladen werden. Das passiert in der ersten Zeile mit der Anweisung `import greenfoot.*;`
Diese Anweisung sorgt dafür, dass nicht nur alle Anweisungen ausgeführt werden können, die zum Java-Grundgerüst-Standard gehören, sondern auch alle Anweisungen verstanden werden, die Greenfoot-spezifisch sind.

Hinweis: Auf diese Weise könnten wir auch noch weitere Klassen importieren, z.B. mit `import java.util.List;` die Klasse Liste, die sich prima eignet, mehrere Objekte in einer einzigen solchen Liste zu speichern.

Hinter dem eigentlichen Java-Code befindet sich ein Kommentar (mit // beginnend), der nicht ausgewertet wird und uns nur dazu dient, den Quelltext zu einem späteren Zeitpunkt noch nachzuvollziehen zu können.

*Hinweis: Auch `/* QUELLTEXT */` **oder** `/** QUELLTEXT */` generiert einen (mehrzeiligen) Kommentar, der ebenso nicht ausgeführt wird.*

Die Definition der Klasse erfolgt dann durch das folgende Gerüst:

```
public class DangerousWorld extends World
{

}
```

public bedeutet öffentlich sichtbar.

*Hinweis: Manchmal möchte man vielleicht Klassen haben, die z.B. nur für Hilfsfunktionen im Hintergrund zuständig sind. Benutzer (oder auch Programmierer) sollen diese eventuell gar nicht benutzen und sehen können. In diesem Fall würde man diese als **private** (nicht öffentlich sichtbar) definieren, was in dem Fall von Weltklassen bei Greenfoot aber übrigens gar nicht geht.*

Mit dem Schlüsselwort **class** beginnt die eigentliche Definition der Klasse. Dahinter folgt der **Name** und (wie in diesem Fall) eventuell die Angabe, von welcher Oberklasse unsere neue Klasse alle Methoden und Attribute erben soll. **extends World** bedeutet also, dass unsere Klasse genau das „können" soll wie die Klasse Welt, plus das, was wir ihr noch beibringen. Wie in Java üblich wird der ganze Block, in dem sich die Klasse befindet mit geschweiften Klammern **{ }** umschlossen.

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

Die einzige „Methode", die wir vorfinden, ist der so genannte **Konstruktor** der Klasse. Hier steht drin, was als erstes passiert, wenn ein Objekt dieser Klasse erstellt wird. Alles was im Konstruktor steht wird zuerst ausgeführt. Der Konstruktor muss den gleichen Namen wie die Klasse tragen, also **DangerousWorld()** plus die Methoden-typischen runden Klammern **()**. Auch hier wird Anfang und Ende des Konstruktors mit geschweiften Klammern **{ }** gekennzeichnet.

Die Anweisung **super(600, 400, 1);** sorgt dafür, dass im Prinzip der Konstruktor der Oberklasse **World** ausgeführt wird. Dieser sorgt wiederum dafür, dass wirklich eine für uns sichtbare Welt auf den Bildschirm gebracht wird. Wir definieren mit den Werten in Klammern noch die Größe der Welt: Bisher 600x400 Zellen mit einer Zellgröße von 1.

Damit die Größe des Szenarios zum (vorgefertigten) Hintergrund passt, ändern wir diese Zeile zu:

```
super(20, 20, 30, false);
```

20 x 20 Zellen mit Zellgröße 30. Der letzte Parameter **false** bedeutet, dass Objekte auch aus der Welt „hinausfliegen" können (ansonsten würden sie immer am Rand „klebenbleiben").

Hinweis: Die Zellgröße gibt dabei an, wie groß die Sprünge sind, die ein Objekt bei der Bewegung durch die Welt zurücklegt. Ein auf den ersten Blick gleich großes Szenario bekäme man auch mit 60x40 Zellen und einer Zellgröße von 10, oder mit 6x4 Zellen und einer Zellgröße von 100. Im ersten Fall hätte man 240.000 Zellen, was eine sehr flüssige Bewegung zuließe, im letzten Fall hätte man 24 Zellen und ein Actor-Objekt würde immer um 100 Pixel springen, wenn es von einem Feld

zum nächsten ginge. Je nach Anwendungsfall ist die eine oder die andere Konfiguration mehr oder weniger sinnvoll (!).

In der API (Doppelklick auf das Symbol der Klasse World), kann man sich mehr Informationen hierzu ansehen).

BEACHTE: Man sollte immer darauf achten, dass der Quelltext vernünftig formatiert wird. Eine ansehnliche und gut lesbare Version bekommt man, wenn man pro Zeile nur eine Anweisung (oder geschweifte Klammer, etc.) schreibt und das Ganze automatisch einrücken lässt. Das geht im Menü **Bearbeiten -> Auto-Layout** (auch per Tastenbefehl).

Die Klasse Finish

Wir wollen sechs Ziele erschaffen. Dazu legen wir eine Klasse **Finish** an und erzeugen später sechs Objekte von dieser Klasse. Das Ziel an sich soll gar nichts Eigenes können, nur da sein. Die Klasse Frosch soll sich später darum kümmern, was passiert, wenn der Frosch das Ziel erreicht.

Wir erstellen also als Unterklasse von **Actor** eine Klasse **Finish** und wählen als Bild z.B. **nature->grass.png**.

Der Quelltext sieht dann wie folgt aus, wenn man den Editor öffnet:

```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * Write a description of class Finish here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Finish extends Actor
{
    /**
     * Act - do whatever the Finish wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public void act()
    {
        // Add your action code here.
    }
}
```

Die automatisch erstellten Vorgaben von Greenfoot zeigen nun, dass Actor-Klassen erst einmal standardmäßig **ohne expliziten Konstruktor**, dafür aber mit einer **act()-Methode** ausgestattet werden.

Erstellen wir zum Test einmal ein Objekt der Klasse **Finish** und ziehen es ganz nach oben in die Randleiste, wo die Ziele hinkommen sollen. Wir bemerken, dass die Grafik etwas größer als die Felder ist. Wir müssten also beim Erstellen eines Objekts noch die Größe der Grafik an die Feldgröße anpassen:

Wir können uns ganz einfach für unsere Klasse einen Konstruktor selbst bauen und die Skalierung per Anweisung dort einbauen (Kommentare sind jetzt weggelassen):

```
import greenfoot.*;

public class Finish extends Actor
{

    public Finish()
    {
        getImage().scale(30,30);
    }

    public void act()
    {

    }

}
```

Wir werden später noch erfahren, was genau passiert, wenn die Anweisung `getImage().scale(30,30);` aufgerufen wird. Zuerst reicht es, zu verstehen, dass wir damit die Grafik des Objekts beim Erstellen auf 30x30 Pixel skalieren.

Damit ist die Klasse **Finish** fertig.

Zwischenergebnis: Eine (fast fertige) Welt ...

Öffne die Klasse **DangerousWorld** und trage unter die Zeile `super(20, 20, 30, false);` folgende Anweisungen ein:

```
addObject(new Finish(), 1, 0);
addObject(new Finish(), 5, 0);
addObject(new Finish(), 9, 0);
addObject(new Finish(), 10, 0);
addObject(new Finish(), 14, 0);
addObject(new Finish(), 18, 0);
```

Klicke anschließend auf **Alle übersetzen** und betrachte den aktuellen Stand unserer „gefährlichen“ Welt.

Die Klasse Frogger anlegen

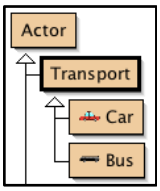
Erstelle eine Klasse **Frogger** (als Unterklasse von Actor) mit einem passenden Bild (unter -> animals zu finden).

Skaliere die Grafik auf 30x30 Pixel (s. oben). Zunächst bearbeiten wir die Klasse nicht weiter.

Gefährliche Fahrzeuge

Autos und Busse haben viel gemeinsam. Warum also in beiden Klassen alle nötigen Features doppelt programmieren? Java bietet hierfür das praktische Konzept der **Vererbung**. Einmal alle Methoden und Attribute in einer Oberklasse programmieren und dann alles an alle Unterklassen automatisch vererben. Das spart Arbeit und ist sehr viel übersichtlicher. Im Übrigen kommt es der Realität deutlich näher. In den jeweiligen Unterklassen programmiert man dann nur noch Dinge, die wirklich **speziell** für die Klasse sind.

Hinweis: Stellt man fest, dass es in zwei oder mehr Unterklassen komplett identisch programmierte Attribute oder Methoden gibt, zieht man diese einfach in die gemeinsame Oberklasse „hoch“.



Daher erstellen wir eine **Actor**-Unterklasse **Transport** und erstellen danach zwei Transport-Unterklassen: **Car** und **Bus** (mit passenden Grafiken).

Da wir keine Objekte der Klasse **Transport** erstellen werden, und diese nur als Ober-Bauplan für unsere Fahrzeuge dienen soll, markieren wir die Klasse mit dem Schlüsselwort **abstract.** Wenn wir nun auf **Alle übersetzen** klicken und versuchen mit der rechten Maustaste ein neues **Transport**-Objekt zu erstellen, stellen wir fest, dass das nicht mehr geht, weil die Klasse als **abstrakte Klasse** definiert wurde. Von abstrakten Klassen kann man keine Objekte ableiten, was manchmal sehr sinnvoll sein kann (hier ist es relativ egal).

Das Ganze geht, indem wir Folgendes in der Klasse **Transport** ändern:

```
public abstract class Transport extends Actor
{
      ...
}
```

Wir wollen nun dafür sorgen, dass sich ein Transportfahrzeug automatisch bewegt. Das geht, indem wir in der **act()-Methode** der Klasse **Transport** die Anweisung

```
move(1);
```

schreiben.

Allerdings müssen wir jetzt die act()-Methoden der Unterklassen Bus und Car löschen! Würden wir das nicht machen, würden sich die Fahrzeuge nicht bewegen. Warum? Weil die act()-Methode der eigenen Klasse immer die der Oberklasse überschreibt. Nur wenn keine eigene act-Methode in einer Klasse vorhanden ist, wird geschaut, ob es in der Oberklasse eine act-Methode gibt. Und nur dann wird diese ausgeführt.

Erstellen wir mit der rechten Maustaste ein Auto und einen Bus und ziehen sie auf die Oberfläche, so können wir nun auf **RUN** drücken und sehen die Fahrzeuge über den Bildschirm sausen.

Nun sollten wir den Bus noch auf 60x30 Pixel und das Auto auf 40x30 Pixel skalieren (s. oben, wie das in den jeweiligen Klassen geht).

Zwar verschwinden die Fahrzeuge am Rand (da wir in der Welt in der Zeile **super(...)** den Wert **false** gesetzt haben), in vielen anderen Situationen ist es aber sinnvoll, die Fahrzeuge verschwinden zu lassen, wenn sie nicht mehr benötigt werden (was ja am rechten Rand der Fall ist).

Wer sorgt aber dafür, dass ein Fahrzeug verschwindet? Lösung: Das Fahrzeug selbst, denn es weiß am besten, wo es sich gerade befindet. Somit können wir in der **act()-Methode** der Klasse **Transport** Folgendes ergänzen:

```
if (getX() == 20)
{
    getWorld().removeObject(this);
}
```

Alles was in der **Entscheidungsanweisung** zwischen den geschweiften Klammern **{ }** steht, wird ausgeführt, wenn die Bedingung zwischen den runden Klammern **( )** wahr ist.

getX() liefert uns den aktuellen X-Wert des Objekts. Ist dieser z.B. 7, dann sieht die Entscheidungsanweisung im Prinzip so aus:

```
if (7 == 20)
{
    getWorld().removeObject(this);
}
```

Da 7 natürlich nicht gleich (==) 20 ist, wird in diesem Fall der Block zwischen den geschweiften Klammern nicht(!) ausgeführt. Nur wenn **getX()** den Wert 20 liefert (und das Objekt genau ein Feld rechts neben dem sichtbaren Bereich (0..19) ist), dann wird der Ja-Block ausgeführt.

Hinweis: == ist der Vergleichsoperator. Zum Zuweisen eines Wertes ist das einfache Gleichzeichen = da.

Mit **getWorld()** können wir auf unsere aktuell geladene Welt zugreifen. Wie dieses im Einzelnen genau funktioniert wird weiter unten noch erläutert.

Hinweis: Man kann auch mehrere Welten erstellen, aber jeweils nur eine laden und anzeigen lassen.

Warum müssen wir auf die Welt zugreifen, wenn wir ein Objekt während der Ausführung des Programms vom Bildschirm entfernen wollen? Ein Blick in die API (Klassen Actor und World) verrät es: Weil nur die Weltklasse die Methoden dafür besitzt. **removeObject(...)** ist das, was wir suchen. In den Klammern muss als **Parameter** angegeben werden, welches Objekt genau entfernt werden soll.

Klar, das Objekt soll sich selbst entfernen, wenn es am Rand ist, daher lautet die Anweisung:

```
getWorld().removeObject(this);
```

this ist ein Verweis auf sich selbst. Später mehr dazu. Diese Zeile veranlasst jedenfalls die Welt dazu, dass das aufrufende Objekt selbst entfernt wird.

Hinweis: Daher muss sichergestellt werden, dass man in der act()-Methode nicht nach dem Entfernen noch versucht, auf das Objekt zuzugreifen. Das würde zu einem Fehler führen.

Kollision Frogger – Fahrzeug

Kommen wir nun zu dem, was passieren soll, wenn der Frosch von einem Fahrzeug angefahren wird. Das führt uns zu der Frage, wie eine Kollision erkannt werden kann. Zunächst der komplette Quellcode der **act()-Methode** der Klasse **Transport** (neu Hinzugekommenes in fett):

```
public void act()
{
    Actor help = getOneObjectAtOffset(1, 0, Frogger.class);
    if (help != null)
    {
        Greenfoot.stop();
    }

    move(1);
    if (getX() == 20) {
        getWorld().removeObject(this);
    }
}
```

Jetzt folgt ein kleiner theoretischer Exkurs zum Thema Referenz. Referenzen sind in Java ein wichtiger Bestandteil. Es lohnt sich, dieses Konzept zu verstehen.

Die Zeile `Actor help = getOneObjectAtOffset(1,0,Frogger.class);` ist höchst interessant. Betrachten wir die Methode `getOneObjectAtOffset(...)` zunächst in einem anderen Zusammenhang:

```
if (getOneObjectAtOffset(0,0,Frogger.class) != null)
{
    ...
}
```

Der Ja-Teil dieser Entscheidungsanweisung wird dann ausgeführt, wenn die Bedingung in runden Klammern **( )** zutrifft, nämlich wenn dieses Transport-Objekt genau auf dem Feld des Frosch-Objekts auftaucht.

Die Parameter `(0,0,Frogger.class)` bedeuten dabei: vom Transport-Objekt aus gesehen wird 0 Felder nach rechts und 0 Felder nach unten nach einem Frosch-Objekt gesucht (also auf dem gleichen Feld, auf dem das Fahrzeug steht). Wird an dieser Stelle jetzt ein Frosch-Objekt gefunden, dann liefert die Methode `getOneObjectAtOffset(...)` einen Verweis auf das gefundene Objekt.

Und dieser Verweis ist **ungleich null** ***(!= null)***, wenn Frosch und Fahrzeug auf dem gleichen Feld stehen, denn **null** wäre er nur, wenn kein Objekt gefunden werden würde. Also liefert der Vergleich mit **null** einen **boole'schen Wert**, der **TRUE** (Frosch-Objekt gefunden) oder **FALSE** (Frosch-Objekt nicht gefunden) sein kann.

In neueren Greenfoot-Versionen sind die beiden folgenden Aufrufe also völlig gleich, da sie beide einen Wahrheitswert (TRUE / FALSE) liefern:

1. `(getOneObjectAtOffset(1,0,Frogger.class) != null)  // : boolean`
2. `(isTouching(Frogger.class))  // : boolean`

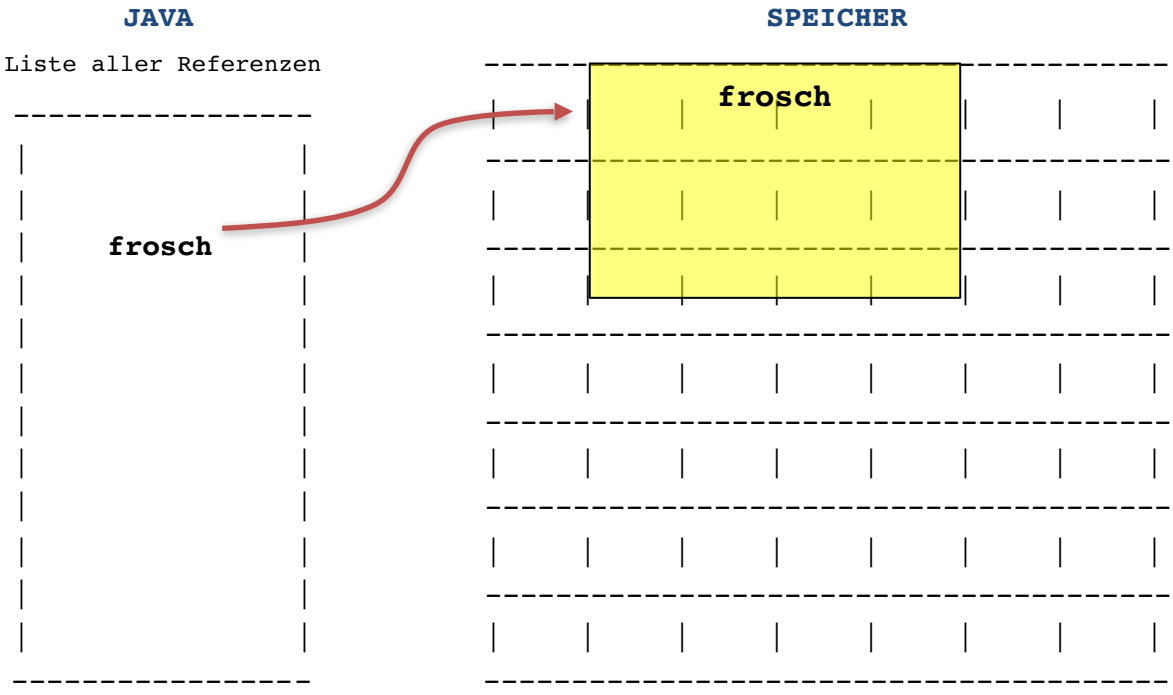
Zurück zu unserem eigentlichen Java-Code:

```
Actor help = getOneObjectAtOffset(1, 0, Frogger.class);
if (help != null)
{
    Greenfoot.stop();
}
```

Irgendwann im fertigen Programm müssen wir ja einmal einen Frosch erzeugt haben, bevor alles losgeht. Das geht z.B. wie folgt:

```
Frogger frosch = new Frogger();
```

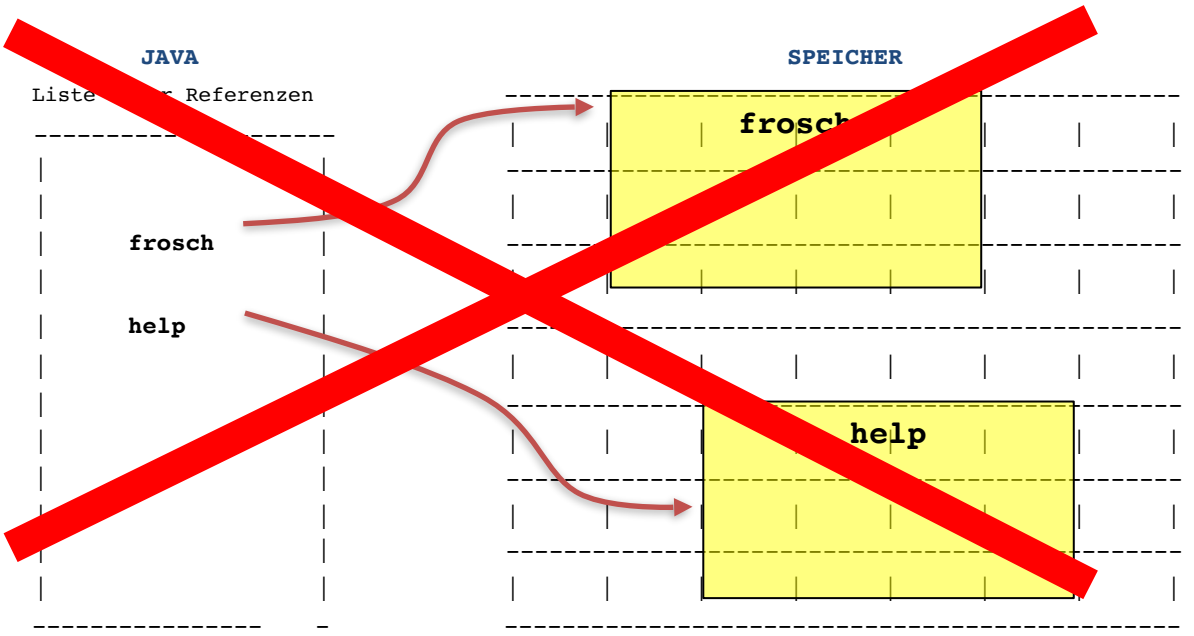
Was passiert dabei genau? Man kann es sich so vorstellen: Java verwaltet zum einen eine Auflistung aller im Szenario vorhandenen Objektnamen und zum anderen liegen diese Objekte dann tatsächlich im Speicher des Computers. Je nach Größe des Objekts wird mehr oder weniger Speicherplatz dafür benötigt. Die folgende Abbildung illustriert dieses:



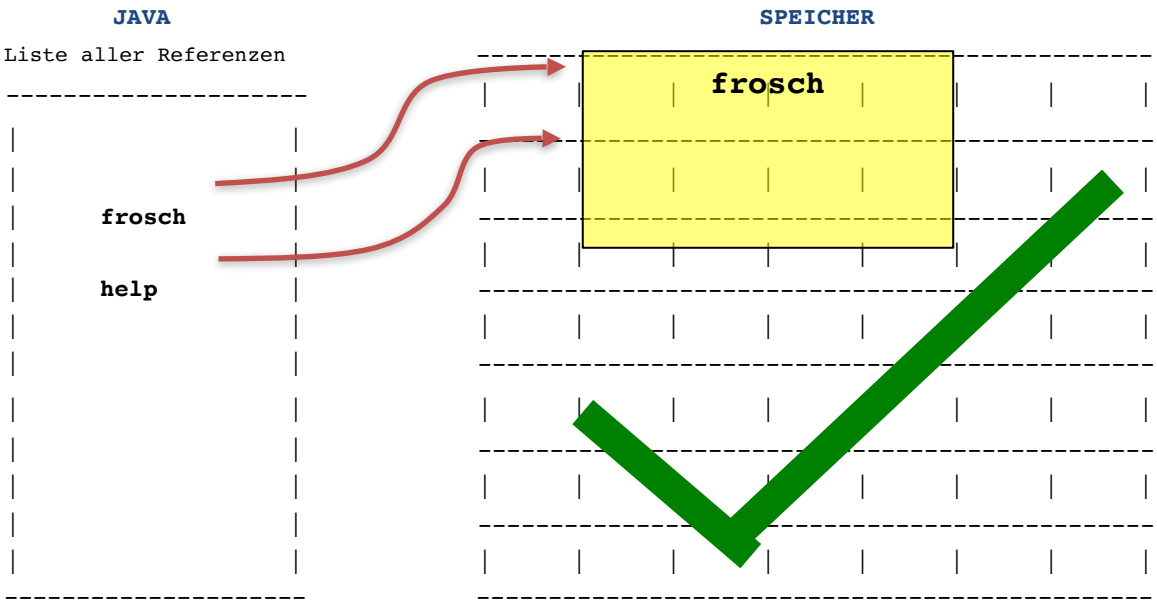
Folgt man dem roten Pfeil, dann landet man genau an der Speicheradresse, an der man das Objekt im Speicher findet. Der Name **frosch** ist in Java also nur ein Verweis auf das im Speicher befindliche Objekt. Dieses Konzept ist in Java omnipräsent und wird als **Referenz** bezeichnet.

Die Zeile `Actor help = getOneObjectAtOffset(1, 0, Frogger.class);` erzeugt nun **kein neues Objekt im Speicher**, sondern nur eine neue Referenz namens **help**, die auf dasselbe Objekt im Speicher verweist. Die folgenden Grafiken zeigen dieses:

SO NICHT:



SO FUNKTIONERT DAS REFERENZ-KONZEPT:



Das bedeutet, dass nun zwei Referenzen (**frosch** und **help**) auf das gleiche Objekt im Speicher zeigen. Das hat auch gravierende Auswirkungen: Wenn man z.B. die Position von **help** ändern würde (z.B. mit `help.setLocation(2,2)`), „dann wäre die Position vom Objekt **frosch** ebenso verändert (und jetzt 2,2)“.

Jetzt wird der Code auch verständlicher: Wenn die Referenz **help** nicht ins Nirvana zeigt (wo nichts ist), sondern tatsächlich auf ein Frosch-Objekt im Speicher zeigt, dann wird der Ja-Teil der Entscheidungsanweisung ausgeführt und mittels `Greenfoot.stop();` wird Greenfoot gestoppt. Damit ist das Spiel vorbei.

```
Actor help = getOneObjectAtOffset(1, 0, Frogger.class);
if (help != null)
{
    Greenfoot.stop();
}
```

*Hinweis: Die Methode **stop()** gehört zur Klasse **Greenfoot**. Will man sie benutzen, reicht es nicht, einfach **stop()** zu schreiben, man muss den Namen der Klasse **Greenfoot** und einen **Punkt** voranstellen, um auf die Methode **stop()** der Klasse **Greenfoot** zuzugreifen.*

Leben in die Welt bringen

Bisher kann unsere Welt nur Objekte auf den Bildschirm bringen, sie kann aber während des Spiels noch nichts aktiv machen. Damit dieses geht, spendieren wir ihr eine eigene **act()-Methode**, denn auch unsere Welt kann am Geschehen des Spiels aktiv teilnehmen:

```
import greenfoot.*;

public class DangerousWorld extends World
{

    public DangerousWorld()
    {
        super(20, 20, 30, false);
        addObject(new Finish(), 1, 0);
        addObject(new Finish(), 5, 0);
        addObject(new Finish(), 9, 0);
        addObject(new Finish(), 10, 0);
        addObject(new Finish(), 14, 0);
        addObject(new Finish(), 18, 0);
    }

    public void act()
    {

    }

}
```

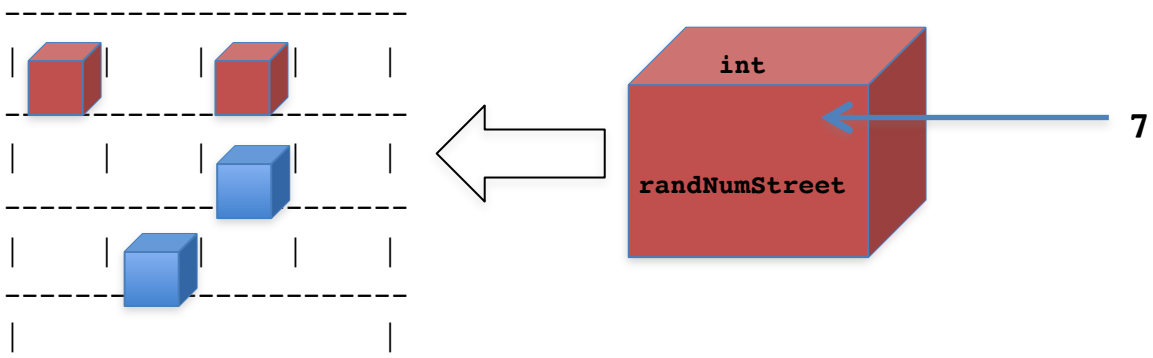
Fahrzeuge sollen automatisch und an zufälligen Y-Positionen von 10 bis 18 am linken Bildschirmrand erscheinen. Dass die Fahrzeuge automatisch fahren und wieder verschwinden, haben wir ja schon in der Klasse **Transport** programmiert.

Eine Zufallszahl erhalten wir mit der Methode `getRandomNumber(...)`, die sich in der Klasse **Greenfoot** befindet:

```
int randNumStreet = Greenfoot.getRandomNumber(9)+10;
```

Es wird also in jedem **ACT-Aufruf** eine **lokale Variable** mit dem **Namen** `randNumStreet` erzeugt, die vom **Typ int** ist und einen **Wert** zugewiesen bekommt, der zwischen 0 (inklusive) und 9 (exklusive) ist. Dann rechnen wir 10 drauf und erhalten somit eine Zahl zwischen 10 und 18.

Hinweis: Variablen kann man sich wie eine Box vorstellen, die in einem großen Regal steht, auf das Java während des Spiels zugreifen kann. Es befinden sich dort unterschiedliche Boxen mit unterschiedlichen Namen (und Datentypen, hier dargestellt durch unterschiedliche Farben), die jeweils einen Wert enthalten.



Wir können diese zufällig erzeugte Zahl nun nutzen, um an dieser Stelle in der Welt beim Aufruf von ACT ein neues Fahrzeug erscheinen zu lassen. Die **act()-Methode** der Klasse **DangerousWorld** sieht dann wie folgt aus:

```
public void act()
{
    int randNumStreet = Greenfoot.getRandomNumber(9)+10;

    if ( (randNumStreet % 2) == 0) {
        addObject(new Bus(), 0, randNumStreet);
    }
    else {
        addObject(new Car(), 0, randNumStreet);
    }
}
```

Zunächst wird die Zufallszahl zwischen 10 und 18 generiert. Danach prüfen wir, ob die Zahl gerade ist `(randNumStreet % 2) == 0`. Dazu nutzen wir die **modulo**-Methode, die immer den Rest einer ganzzahligen Division angibt (Bsp.: 7%2 = 1; 6%2 = 0). Ist die Zahl also gerade, soll ein Bus in der Zeile entstehen, ist die Zahl ungerade, soll ein Auto in der entsprechenden Zeile entstehen.

Hinweis: Zur Erläuterung noch einmal ein paar Beispiele mit dem Modulo:

8 / 2 = 4, Rest 0	*8 % 2 = 0*
8 / 3 = 2, Rest 2	*8 % 3 = 2*
7 / 2 = 3, Rest 1	*7 % 2 = 1*

Dies erzeugt ziemlich viele Autos. Außerdem werden manchmal zwei Autos auf „einem Haufen“ erzeugt. Dieses Problem gilt es später noch aus der Welt zu schaffen.

Alternativ könnte man auch Zufallszahlen zwischen 0 und z.B. 25 erstellen und nur diejenigen Autos auf den Bildschirm bringen, die Zufallszahlen zwischen 10 und 18 zugeordnet werden können. Man müsste also dafür sorgen, dass der Ja-Teil einer solchen Entscheidungsanweisung nur dann ausgeführt wird, wenn die Zufallszahl, die erstellt wurde, zwischen 10 und 18 liegt. Dazu müssen mehrere Bedingungen miteinander verknüpft werden. Dies ginge wie folgt und bringt zunächst ein gutes Ergebnis:

```
int randNumStreet = Greenfoot.getRandomNumber(25);

if ( (randNumStreet > 9) && (randNumStreet < 19) ) {
     if ( (randNumStreet % 2) == 0) {
         addObject(new Bus(), 0, randNumStreet);
     }
     else {
         addObject(new Car(), 0, randNumStreet);
     }
}
```

&& ist eine Verknüpfung von zwei (oder mehr) Bedingungen. Dies bedeutet **UND.** Nur wenn beide Bedingungen wahr sind, wird der Ja-Teil ausgeführt.

*Hinweis: Mit **UND** und **ODER** kann man Bedingungen verknüpfen. Die Verknüpfungsoperatoren dazu sind **&&** und ||. Die folgende Übersicht zeigt die logischen Wahrheitstafeln für zwei Eingaben und die UND / ODER Funktion:*

x1	***x2***	***&&***	***||***
0	*0*	*0*	*0*
0	*1*	*0*	*1*
1	*0*	*0*	*1*
1	*1*	*1*	*1*

Der Frosch lernt Laufen

Nun ist es an der Zeit, die Tastatursteuerung für den Frosch in der **act-Methode** des **Frosches** zu implementieren:

```
if (Greenfoot.isKeyDown("up") == true) {
    setLocation(getX(),getY()-1);
}
if (Greenfoot.isKeyDown("left") == true) {
    setLocation(getX()-1,getY());
}
if (Greenfoot.isKeyDown("right") == true) {
    setLocation(getX()+1,getY());
}
```

Die Methode **isKeyDown(...)** der Klasse **Greenfoot** eignet sich dazu, Tastatureingaben abzufragen. Ist beispielsweise die Pfeiltaste nach oben gedrückt („up“), dann ist die Bedingung `(Greenfoot.isKeyDown("up") == true)` wahr. In diesem Fall wird die Methode `setLocation(...)` aufgerufen. Als Parameter erwartet die Methode zwei Werte: die neue x- und y-Position, auf die das Objekt gesetzt werden soll. Wir holen uns mit `getX()` die aktuelle x-Position und mit `getY()` die aktuelle y-Position und subtrahieren noch `1`, damit der Frosch sich nach oben bewegt, wenn die Pfeiltaste nach oben gedrückt wird. Dann setzen wir den Frosch auf die neue Position.

Analog verfahren wir mit den anderen Richtungen. Nach unten darf der Frosch bei Frogger nicht bewegt werden.

Erster Zwischentest

Fügen wir in den Konstruktor der DangerousWorld noch eine Zeile mit

```
addObject(new Frogger(), 9, 19);
```

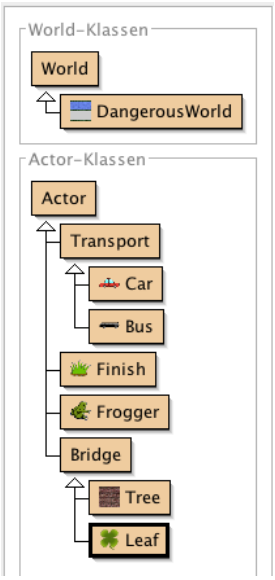
ein, so können wir mit anschließendem Klick auf RUN einen ersten Test wagen.

Baumstämme und Blätter

Wir wollen uns nun darum kümmern, dass im oberen Teil Baumstämme und Blätter ihre Wege ziehen. Der Unterschied zu den Fahrzeugen bei einer „Kollision“ ist, dass man auf ihnen mitfahren können soll. Man muss also diese Objekte berühren im Gegensatz zu den Fahrzeugen, denen man ausweichen muss.

Wir erstellen zunächst eine gemeinsame Oberklasse für die „Brückenteile“ und nennen diese **Bridge** (ohne Bild). Danach erstellen wir zwei weitere Bridge-Unterklassen: **Tree** und **Leaf** mit den entsprechenden Bildern (backgrounds -> brick.jpg und nature -> shamrock.png).

Damit ergibt sich folgende Klassenhierarchie insgesamt:



Hinweis: Die objektorientierte Modellierung versucht immer, möglichst nah an der von den Menschen wahrgenommenen Realität zu arbeiten. Daher werden sinnvollerweise für alle möglichen Objekte, die es später geben soll, Klassen (quasi Baupläne) erstellt. Dabei fasst man zusammen. Man braucht viele Autos, also wird eine Klasse Auto erstellt. Man braucht viele Busse, also wird eine Klasse Bus erstellt. In Oberklassen kann man nun Gemeinsamkeiten implementieren, die für alle Unterklassen gleich sind (Bsp: Transport, gleiche Attribute und Methoden von Car und Bus).

Da wir von der Klasse Bridge selbst keine Objekte erstellen werden (und es auch nicht vorgesehen ist, dass dieses in einer späteren Version evtl. passiert) deklarieren wir die Klasse als **abstract**.

```
import greenfoot.*;

public abstract class Bridge extends Actor
{

    public void act()
    {

    }
}
```

Blätter und Baumstämme sind noch etwas zu groß für die Felder. Wir nutzen diesmal aus, dass sie beide gleich groß sein sollen (30x30 Pixel) und sparen uns Schreibarbeit, indem wir die Größenskalierung nicht in den Klassen **Tree** und **Leaf** selbst implementieren, sondern sie in die Oberklasse **Bridge** hochziehen. In den Konstruktor dieser Klasse (den wir zuerst selbst bauen müssen) schreiben wir folgendes hinein:

```
public Bridge()
{
    getImage().scale(30,30);
}
```

Gegenläufige Bewegung

Baumstämme sollen sich von links nach rechts und Blätter von rechts nach links bewegen. Da dieses nichts Neues mehr ist, im Folgenden der Quelltext. Zu beachten ist, dass diesmal eine andere Variante aufgeführt ist. Statt ein Feld zu gehen und dann zu überprüfen, ob das Objekt am Bildschirm angelangt ist, wird dieses nun in einer Entscheidungsanweisung mit Alternative genauso gut erledigt:

```
import greenfoot.*;

public class Tree extends Bridge
{

    public void act()
    {
        if (getX() > 19) {
            getWorld().removeObject(this);
        }
        else
        {
             move(1);
        }
    }
}
```

```
import greenfoot.*;

public class Leaf extends Bridge
{

    public void act()
    {
        if (getX() < 0) {
            getWorld().removeObject(this);
        }
        else
        {
             move(-1);
        }
    }
}
```

Es wird bei beiden Klassen zunächst die Bedingung in runden Klammern geprüft: Bei Tree, ob das Objekt über das letzte (das 19.) Feld drüber ist (also rechts vom Bildschirm ist) und bei Leaf, ob das Objekt links vom Bildschirm angekommen ist (das erste Feld ist ja das Feld 0). Ist dieses der Fall, wird es entfernt, **ansonsten** wird ein Schritt nach rechts bzw. links gegangen.

Sinnvolle Brücken

Was nun noch fehlt, ist die automatische Erzeugung von Brücken (Baumstämme oder Blätter). Schließlich sollen Blöcke aus 1 bis 5 Brückenelementen gleichzeitig erzeugt werden. Eine erste Version, die dieses umsetzt könnte wie folgt strukturiert sein:

- Wir erzeugen eine Zufallszahl von 1 bis 5. Diese Zahl soll die Länge unserer Brücke sein (also die Anzahl der Brückenelemente).
- Dann erzeugen wir eine Zufallszahl von 1 bis 8 für die vertikale Position der Brücke.
- Dann platzieren wir diese Brücke zunächst außerhalb des Bildschirms (rechts oder links, je nach Typ) auf die soeben zufällig generierte vertikale Position. Die act()-Methode der beiden Klassen sorgt dann schließlich dafür, dass sich alle Elemente gleichzeitig immer um ein Feld verschieben, also dafür dass die Brücke als Ganzes über den Bildschirm scrollt. *(Hinweis: Außerhalb des Bildschirms Platzieren funktioniert nur, weil wir in der Welt* `super(20, 20, 30, false);` *gewählt haben).*

Damit gestaltet sich der Quellcode, den wir in die Klasse **DangerousWorld** innerhalb der **act()-Methode einfügen**, wie folgt:

```
        // Trees and leafs
        int randNumRiver = Greenfoot.getRandomNumber(8)+1;
        int randCount = Greenfoot.getRandomNumber(5)+1;
        if ( (randNumRiver % 2) == 0)
        {
            for (int i=0; i<randCount; i=i+1)
            {
                addObject(new Tree(), 0-i, randNumRiver);
            }
        }
        else
        {
            for (int i=0; i<randCount; i=i+1)
            {
                addObject(new Leaf(), 20+i, randNumRiver);
            }

        }
```

In der Variablen `randNumRiver` wird nun eine Zufallszahl aus dem Bereich [1..8] generiert.
In der Variablen `randCount` wird eine Zufallszahl aus dem Bereich [1..5] generiert.
Genau wie bei den Bussen und Autos kommen die Bäume in die gerade Zeilen und die Blätter in die ungeraden Zeilen. Dafür soll die Entscheidungsanweisung sorgen, indem der Ja-Teil nur ausgeführt wird, wenn der Rest der ganzzahligen Division von `randNumRiver` durch 2 den Wert 0 hat.
Im Ja-Fall (gerade Zeile) sorgt eine **for-Schleife** dafür, dass mehrere Bäume-Teile nebeneinander erzeugt werden, im Nein-Fall sorgt eine **for-Schleife** dafür, dass mehrere Blätter-Teile nebeneinander erzeugt werden.

Betrachten wir die obere **for-Schleife** genauer:

```
for (int i=0; i<randCount; i=i+1)
{
      addObject(new Tree(), 0-i, randNumRiver);
}
```

Mittels **for-Schleifen** kann all das, was zwischen den geschweiften Klammern **{ }** steht sooft ausgeführt werden, wie in den runden Klammern **( )** angegeben wurde. Die for-Schleife ist daher eine **Zählschleife.**

Zwischen den runden Klammern werden drei Dinge angegeben, getrennt von Semikola:

(*Deklaration* **;** *Laufzeitbedingung* **;** *Inkrement* **)**

Zunächst wird eine **lokale Variable** deklariert, die nur zwischen den geschweiften Klammern **{ }** der for-Schleife existiert(!). In unserem Fall heißt die Variable **i**, ist vom Typ **int** und bekommt den Wert **0**. Solange **i** kleiner als der Wert der Zahl `randCount` ist, wird **i** um 1 erhöht.

Betrachten wir dazu ein konkretes Beispiel und ersetzen die Variable `randCount` durch den festen Wert 10:

```
for (int i=0; i<10; i=i+1)
{
    System.out.println(i); // gibt den Wert von i auf der Textkonsole aus
}
```

Die Werte von i werden nacheinander auf der **Java-Konsole** ausgegeben und betragen: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Die Schleife läuft also 10 Mal.

Hinweis: Gültige Operatoren für den Vergleich in der Laufzeitbedingung sind u.a.

- *< (kleiner)*
- *> (größer)*
- *<= (kleiner gleich)*
- *>= (größer gleich)*

Zurück zum eigentlichen Code:

```
for (int i=0; i<randCount; i=i+1)
{
    addObject(new Tree(), 0-i, randNumRiver);
}
```

Die for-Schleife läuft genauso oft, wie Brückenteile erzeugt werden sollen. Ist die zufällig gewählte Brückenlänge z.B. 3, dann werden die einzelnen Brückenteile nun an die X-Positionen 0, -1, und -2 gesetzt.

Die zweite for-Schleife läuft sehr ähnlich:

```
for (int i=0; i<randCount; i=i+1)
{
    addObject(new Leaf(), 20+i, randNumRiver);
}
```

Wäre die Brückenlänge wieder 3, dann würden die einzelnen Brückenteile auf die Positionen 20, 21 und 22 gesetzt.

Nun bewegen sich alle Objekte wie gewünscht über den Bildschirm.

Testen wir das Programm, wird der Frosch bereits von den Autos überfahren, jedoch können wir immer noch durch das Wasser zum Ziel laufen. Ferner verdecken die Brückenteile den Frosch, was nicht schön aussieht. Den letzten Punkt können wir ganz einfach eliminieren, indem wir in den **Konstruktor** der Klasse **DangerousWorld** folgendes unter die Zeile `super(20, 20, 30, false);` einfügen:

```
setPaintOrder(Transport.class, Frogger.class, Bridge. class, Finish.class);
setActOrder(Frogger.class, Bridge.class);
```

Diese Anweisung legt die Zeichnungsreihenfolge fest. Damit wird ein Brückenelement über einem Ziel gemalt, ein Frosch über einem Brückenelement und einem Ziel, und ein Transport-Objekt wird ganz oben drauf gezeichnet.

Die zweite Zeile `setActOrder(...)` sorgt noch schnell dafür, dass zuerst immer die act()-Methode vom Frogger aufgerufen wird, bevor die act()-Methoden der Brückenelemente aufgerufen werden.

Hinweis: An dieser Stelle ist es nicht so wichtig, genau zu verstehen, warum diese Anweisung notwendig ist. Ohne sie, würde der Frosch aber immer von den Brücken fallen, wenn er sich bewegt.

Mit den Brückenteilen schwimmen

Damit der Frosch auf den Brückenteilen schwimmt, sind wieder einige Kollisionsabfragen notwendig. Fassen wir die Überlegungen, die notwendig sind, wie folgt zusammen:

Unser Algorithmus soll nur ablaufen, wenn sich der Frosch auf den Wasser-Feldern befindet, also die y-Koordinaten 1 bis 8 hat. Daher verknüpfen wir zwei Bedingungen mit einem logischen **UND:** Wenn der Frosch auf einem Feld > 0 und gleichzeitig auf einem Feld < 9 ist.

```
if ( (getY() < 9) && (getY() > 0) )
{
      ...
}
```

Innerhalb der geschweiften Klammern prüfen wir nun, ob unter ihm ein Baum- oder ein Blatt-Objekt ist.

- Ist ein Baum-Objekt vorhanden, soll er mit diesem nach rechts wandern
- Ist ein Blatt-Objekt vorhanden, soll er mit diesem nach links wandern
- Ist kein Objekt vorhanden, soll Greenfoot gestoppt werden.

Damit es nicht zu viele Verschachtelungen gibt oder zu viele Bedingungen auf einmal verknüpft werden (und man vielleicht wichtige Fälle vergisst), ist die Reihenfolge nicht unwichtig.

Am besten erstellt man zunächst zwei Referenzen. Eine auf ein mögliches Baum-Objekt und eine auf ein mögliches Blatt-Objekt. Dann schaut man, ob eine der beiden Referenzen ungleich **null** ist (also tatsächlich auf ein reales Objekt im Speicher zeigt) und danach prüft man dann ob eventuell sogar beide Referenzen **null** sind, denn dann befindet man sich im Wasser (ohne Objekte unter dem Frosch), und dann soll Greenfoot gestoppt werden:

Es wird also folgender Quellcode in die **act()-Methode** der Klasse **Frogger** eingefügt:

```
        if ( (getY() < 9) && (getY() > 0) )
        {

            Actor tree = getOneObjectAtOffset(0, 0, Tree.class);
            if (tree != null)
            {
                setLocation(getX()+1, getY());
            }

            Actor leaf = getOneObjectAtOffset(0, 0, Leaf.class);
            if (leaf != null)
            {
                setLocation(getX()-1, getY());
            }

            if ( (tree == null) && (leaf == null) )
            {
                Greenfoot.stop();
            }

        }
```

`Actor tree = getOneObjectAtOffset(0, 0, Tree.class);` erstellt eine Referenz auf ein mögliches Baum-Objekt. Die Entscheidungsanweisung

```
        if (tree != null)
        {
            setLocation(getX()+1, getY());
        }
```

verzweigt dann in den Ja-Teil, wenn sich wirklich ein Objekt unter dem Frosch befindet und die Referenz nicht leer (null) ist. In diesem Fall wird der Frosch ein Feld nach rechts gerückt (es werden mit getX() und getY() die aktuellen Koordinaten geholt und dann wird die x-Koordinate um 1 erhöht, schließlich wird der Frosch dann auf die neue Position gesetzt).

Analog funktioniert das für ein Blatt-Brückenteil.

Wenn beide Referenzen leer sind, dann soll Greenfoot gestoppt werden. Das ist dann der Fall, wenn gilt: `( (tree == null) && (leaf == null) )`

Hinweis: Klammere immer ordentlich. Du kennst das Prinzip aus der Mathematik, dass einige Operatoren stärkere Bindung haben (z.B. Punkt- vor Strichrechnung).

Nächstes Level

Wir vereinfachen Frogger so, dass es reicht, eines der Ziele zu erreichen. Wird ein (beliebiges) **Finish-Objekt** berührt, dann soll der Frosch wieder auf die Anfangsposition gesetzt werden und die Geschwindigkeit soll um 5 erhöht werden (damit sieht es so aus, als gäbe es ein nächstes Level mit höherer Geschwindigkeit).

```
// finish
if (getY() == 0)
{
    Actor finish = getOneObjectAtOffset(0, 0, Finish.class );
    if (finish == null)
    {
        Greenfoot.stop();
    }
    else
    {
        setLocation(10, 19);
        speed = speed + 5;
        Greenfoot.setSpeed(speed);
    }
}
```

Die Abfrage nach dem Berühren eines Finish-Elements soll natürlich nur in der Zeile 0 stattfinden, daher wird der ganze Blok nur ausgeführt, wenn die Bedingung wahr `(getY() == 0)` ist. Es ist immer gut, sich genau vorher zu überlegen, wann man diesen Block nur braucht.
Anschließend erstellen wir eine Referenz auf ein mögliches Finish-Objekt unter dem Frosch. Die folgende Entscheidungsanweisung mit Alternative sorgt dafür, dass Greenfoot gestoppt wird, falls wir doch neben einem Frosch auf den grünen Moosrand gesprungen sind, und andernfalls dafür, dass der Frosch wieder an die Startposition rückt.

Die Anweisung `Greenfoot.setSpeed(speed = speed + 5);` verwendet die Methode `setSpeed(...)` der Klasse Greenfoot, die einen **int** als Parameter erwartet und die Anzahl der ACT Aufrufe pro Sekunde steuert. Wir übergeben hier den Wert, der in der Variablen **speed** gespeichert ist. Vorher wird dieser Wert um 5 erhöht, damit das nächste Level schneller läuft (das sieht man auch an dem Speed-Balken unten im Szenario.

Wenn wir das Programm ausführen, gibt es eine Fehlermeldung, weil wir die Variable **speed** noch nicht deklariert haben.

Man sieht schnell ein, dass wir sie nicht innerhalb der geschweiften Klammern der act()-Methode von Frogger deklarieren können, da sie ja so jedes Mal den gleichen Wert bekommt, wenn die act()-Methode aufgerufen wird. Daher schreiben wir sie „zwischen die geschweiften Klammern der ganzen Klasse“, denn wir wissen ja, dass Variablen immer nur zwischen den geschweiften Klammern existieren, wo sie deklariert wurden. Damit ist die Variable **speed** in der ganzen Klasse verfügbar (auch in jeder Methode).

```
public class Frogger extends Actor
{
    private int speed = 20;

    public Frogger()
    {
        getImage().scale(30,30);
    }

      ...
      ...
}
```

Nun müssen wir noch dafür sorgen, dass die Geschwindigkeit von Greenfoot beim ersten Start auch 20 ist. Dazu schreiben wir in den **Konstruktor** der Klasse **DangerousWorld** die Zeile:

```
Greenfoot.setSpeed(20);
```

*Hinweis: Beachte, dass die Anweisung super(...) immer ganz oben als **erste Anweisung** im Konstruktor stehen muss!*

VIEL SPAß BEIM PROGRAMMIEREN UND SPIELEN!

ANHANG: