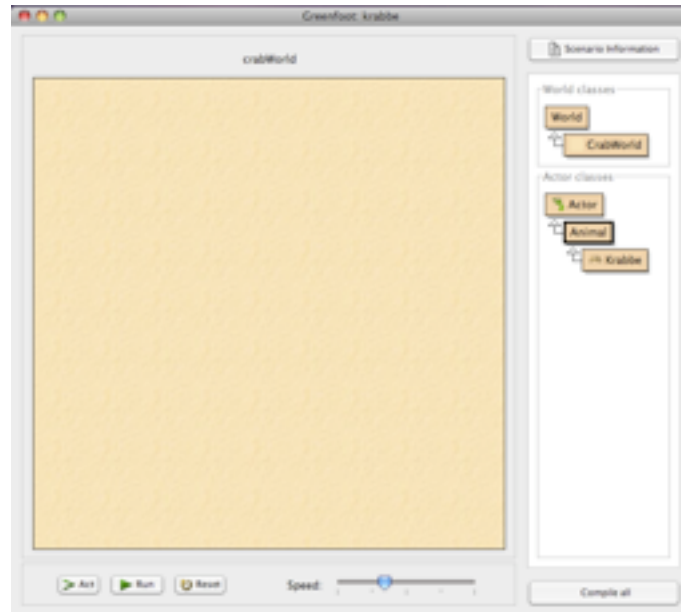


1. Kleine Krabbe

Starte *Greenfoot* und öffne das Szenario *krabbe1*. Es erscheint das Hauptfenster von *Greenfoot* mit dem geöffneten Szenario.



Das Hauptfenster besteht aus drei Bereichen und einigen zusätzlichen Buttons.

- Die **Welt**: Der größte Bereich wird Welt genannt. In diesem Bereich wird das Programm ausgeführt und du kannst die Aktionen beobachten, was du programmiert hast.
- Das **Klassendiagramm**: In diesem Bereich befinden sich die hellbraunen Kästchen, die mit Pfeilen verbunden sind. In diesen so genannten Klassen schreibst du den Quelltext deines Programms.
- Die **Greenfoot-Steuerung**: In diesem Bereich findest du die Buttons Act, Run und Reset sowie einen Schieberegler für die Geschwindigkeit. Diese dienen der Programmsteuerung.

1.1. Untersuchung der Ausgangslage

Die Sprache, mit der du in *Greenfoot* programmierst, ist Java. Da Java eine objektorientierte Sprache ist, sind die Konzepte von Klassen und Objekten von grundlegender Bedeutung.

Im Klassendiagramm findest du alle Klassen dieses Szenarios. Sie sind eingeteilt in

Word classes: World, Krabbenwelt,

Actor classes: Actor, Animal, Krabbe

1.1.1. Objekte und Klassen

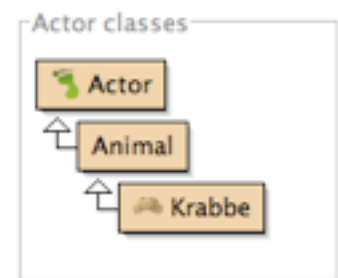
In diesem Szenario existieren bereits einige Klassen. Zuerst wirst du die Klassen *Actor*, *Animal* und *Krabbe* untersuchen.

Bemerkung:

Die Klassen mit englischen Namen sind aus den Beispielpogrammen entnommen, die für Greenfoot mitgeliefert werden. Du wirst in diesen Klassen keinen Quelltext schreiben.

Die Klassen mit deutschem Namen sind die Klassen, mit denen du arbeiten wirst.

Im Klassendiagramm sind diese Klassen mit Pfeilen verbunden. Dies weist auf eine *Ist-ein*-Beziehung hin. Eine Krabbe *ist ein* Animal und ein Animal *ist ein* Actor. Daraus folgt natürlich, dass eine Krabbe auch ein Actor ist.



Zu Beginn wirst du nur mit der Klasse *Krabbe* arbeiten. Die Klasse *Krabbe* ist das allgemeine Konzept einer Krabbe - sie beschreibt somit alle Eigenschaften und Fähigkeiten, die Krabben besitzen. Sobald eine Klasse existiert, kannst du davon Objekte erzeugen. Klickst du mit der rechten Maustaste auf die Klasse *Krabbe*, öffnet sich ein Kontextmenü. Mit der ersten Option *new Krabbe()* erzeugst du ein neues *Krabbe*-Objekt. Es wird ein Bild einer kleinen Krabbe eingeblendet, das du mit der Maus irgendwo in die Krabbenwelt verschieben kannst.

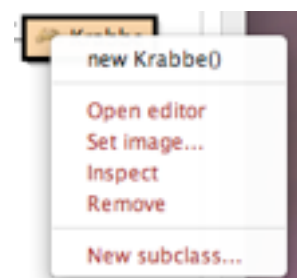
Übung 1.1:

Erzeuge eine Krabbe und platziere diese in der Welt. Erzeuge weitere Krabben. Führe das Programm aus, indem du auf den *Run*-Button klickst. Was kannst du beobachten?

Bemerkung:

Du kannst besonders schnell Objekte in deiner Welt ablegen, wenn du eine Klasse im Klassendiagramm auswählst und dann mit gedrückter *Shift*-Taste in die Welt klickst.

Du wirst bemerkt haben, dass die Krabben nichts machen, wenn du den *Run*-Button drückst. Um dies zu ändern, wirst du den Quelltext der Klasse *Krabbe* verändern. Klicke hierzu mit der rechten Maustaste auf die Klasse *Krabbe* und öffne den Editor mit der Option *Open editor*. Hier findest du den folgenden Quelltext. Schneller geht es mit einem Doppelklick auf das Klassensymbol.



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

public class Krabbe extends Animal
{
    public void act()
    {
        //leer
    }
}
```

Dieser Quelltext definiert, was die Krabbe machen kann. In der Methode *act()* steht zwischen den beiden geschweiften Klammern nur der Kommentar *//leer*, deswegen macht auch die Krabbe nichts. Hier solltest du nun einen Quelltext eingeben, der die Aktionen der Krabbe festlegt.

```
public void act()
{
    move();
}
```

Übung 1.2:

Ersetze den Kommentar in der Methode *act()* durch den Befehl *move()*. Kompiliere das Szenario, indem du auf den Button *Compile all* drückst.

Erzeuge eine Krabbe und platziere diese in der Welt. Klicke auf die Buttons *Act* bzw. *Run*.

Erzeuge weitere Krabben. Führe das Programm aus, indem du auf den *Run*-Button klickst. Was kannst du beobachten?

In der Übung 1.2 hast du zwei Krabben erzeugt. Sie bewegen sich beide nach rechts mit der gleichen Geschwindigkeit, wenn du den *Run*-Button drückst. Aber warum machen die beiden Krabben das gleiche?

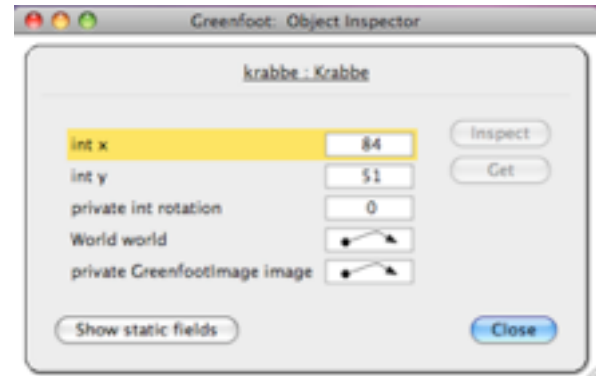
1.1.2. Das Klassendiagramm

Bevor du weiter mit den Krabben experimentierst, solltest du noch einen genaueren Blick auf das Klassenkonzept werfen. Wie bereits oben erwähnt, ist eine Klasse eine Art Bauanleitung. Sie hilft dir beim Erzeugen von Objekten und beschreibt genau deren Eigenschaften und Fähigkeiten. Du kannst von einer Klasse beliebig viele Objekte erzeugen. Da jedoch alle Krabben-Objekte nach der gleichen Bauanleitung erstellt werden, ist nun auch verständlich, dass sie die gleichen Eigenschaften und die gleichen Fähigkeiten besitzen.

Die **Eigenschaften** eines Krabben-Objekts werden in den Datenfeldern festgehalten und du erfährst diese mit Hilfe des Objektinspektors. Dazu klickst du mit der rechten Maustaste auf ein Objekt und wählst die Option *Inspect*. Anschließend öffnet sich der Objektinspektor. Dieser gibt dir Auskunft über die Werte, die in den Datenfeldern des Objekts enthalten sind.



Im Beispiel rechts hat die Krabbe die Positionen $x = 84$, $y = 51$, die Bewegungsrichtung $rotation = 0$ (d.h. sie bewegt sich nach Osten), weiterhin hält sie eine Referenz auf die Welt, in der sie lebt, und auf das Bild, mit der sie in ihrer Welt dargestellt wird.



Übung 1.3:

Öffne den Objektinspektor von mehreren Krabben. Bewege mit der Maus eine Krabbe in eine andere Position. Was beobachtest du?

Die **Fähigkeiten** eines Krabben-Objekts werden in den Methoden festgelegt und du erfährst diese im Quelltext der entsprechenden Klasse. Hier existiert nur eine Methode *act()*, in der du die Methode *move()* implementiert hast. Die Methode *act()* wird einmal aufgerufen, wenn du den *Act*-Button bzw. mehrmals, wenn du den *Run*-Button in der Greenfoot-Steuerung drückst. Vielleicht fragst du dich nun, wieso die Krabbe die Anweisung *move()* versteht?

Hier haben die Greenfoot-Entwickler dir bereits eine Menge Arbeit abgenommen. Im Klassendiagramm erkennst du die durch einen Pfeil dargestellte *ist-ein*-Beziehung. Die Krabbe *ist ein* Animal. Man sagt auch, die Klasse *Animal* ist eine **Superklasse** (Oberklasse) und die Klasse *Krabbe* ist eine **Subklasse** (Unterklasse). Die Superklasse vererbt alle ihre Eigenschaften und Fertigkeiten an die Subklasse weiter.

Übung 1.4:

Öffne den Editor der Superklasse *Animal*. In der Dokumentation erkennst du eine Zusammenfassung der Konstruktoren und der Methoden dieser Klasse. Ein Doppelklick auf *move()* liefert dir eine genauere Beschreibung.

Oben rechts kannst du in einem Auswahlménü zwischen *Documentation* und *Source Code* auswählen. Hier kannst du nachvollziehen, wie die Greenfoot-Entwickler der Krabbe mitteilen, einen Schritt in die angegebene Richtung zu gehen. Du musst jedoch zum jetzigen Zeitpunkt diesen Quelltext nicht verstehen.

Constructor Summary

[Animal\(\)](#)
Konstruktor fuer Animal - ohne Aufgabe.

Method Summary

void	act() Leere Methode.
boolean	atWorldEdge() Prueft, ob wir nahe an einem der R%ender der Welt sind.
boolean	canSee(java.lang.Class cls) Liefert true zurueck, wenn wir genau dort, wo wir sind, ein Objekt der Klasse 'cls' sehen.
void	eat(java.lang.Class cls) Versucht, ein Objekt der Klasse 'cls' zu fressen.
void	move() Rueckt vorwaerts in die aktuelle Richtung.
void	turn(int angle) Dreht 'angle' Grad nach rechts (im Uhrzeigersinn).

In der Superklasse *Animal* ist die Methode *move()* implementiert und gibt diese Fertigkeit an ihre Subklasse *Krabbe* und natürlich alle ihre weiteren Subklassen weiter. Da die Klasse *Animal* weiß, wie Bewegungen ausgeführt werden, weiß das die *Krabbe* auch.

Übung 1.5:

In der Dokumentation der Superklasse *Animal* findest du die Methode *turn()*. Lies die Beschreibung dieser Methode und implementiere diese in die Methode *act()* der Klasse *Krabbe*.

Denke daran: Jedes Mal, wenn du deinen Quelltext änderst, musst du diesen erneut kompilieren.

Bemerkung:

Wenn eine Fehlermeldung durch Hervorhebung einer Zeile und unten im Editorfenster angezeigt wird, versucht eine Meldung den Fehler zu erläutern. Die hervorgehobene Zeile ist oft die Zeile mit dem Problem, doch manchmal tritt das Problem auch in der Zeile davor auf. Rechts unten erscheint auch ein Button mit einem Fragezeichen. Durch Anklicken dieses Buttons erhältst du zusätzliche Informationen zur Fehlermeldung.

Die Klasse *Krabbe* **erbt von** der Klasse *Animal* oder die Klasse *Krabbe* **erweitert** die Klasse *Animal*.

Die Subklasse erweitert demnach die Superklasse. Sie erhält alle Eigenschaften der Superklasse und besitzt zusätzlich ihre speziellen Eigenschaften.

In Java wird für die Erweiterung einer Superklasse zur Subklasse das Schlüsselwort **extends** (engl.: erweitern) verwendet. Für die Vererbungsbeziehung werden üblicherweise Pfeile mit nicht gefüllten Pfeilspitzen verwendet.

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *krabbe2*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

1.2. Am Rande der Welt

Du hast in den letzten Abschnitten der Krabbe zum Laufen und zum Drehen gebracht. Sie blieben allerdings am Rand des Bildschirms hängen. Dieses Verhalten ist von den Greenfoot-Entwicklern standardmäßig so konzipiert.

Nun soll die Krabbe merken, dass sie den Rand ihrer Welt erreicht hat und dort wenden und wieder weiter bewegen. Zum Wenden und Bewegen kennst du bereits die Methoden *turn()* und *move()*. Aber wie erkennt die Krabbe, dass sie sich am Rande der Welt befindet?

Übung 1.6:

Suche in der Dokumentation der Superklasse *Animal* nach einer geeigneten Methode, mit der die Krabbe das Ende ihrer Welt erkennen kann. Welchen Rückgabewert hat diese Methode?

Übung 1.7:

- a) Erzeuge eine Krabbe. Klicke sie mit der rechten Maustaste an und suche die Methode *atWorldEdge()*. Rufe diese Methode auf. Welchen Rückgabewert liefert sie zurück?
- b) Lass die Krabbe an den Rand des Bildschirms laufen (oder bewege sie dorthin) und rufe erneut die Methode *atWorldEdge()* auf. Was liefert sie jetzt zurück?

Durch Aufrufen von Methoden, die einen Rückgabewert liefern (das bedeutet bei denen der Rückgabewert nicht *void* ist), wird kein Befehl ausgegeben, sondern eine Frage gestellt. Wenn du die Methode *atWorldEdge()* verwendest, antwortet diese Methode entweder mit *true* oder *false*. Somit eignet sich diese Methode gut, um zu prüfen, ob die Krabbe den Rand der Welt erreicht hat.

Diese Methode wird mit einer *if*-Anweisung kombiniert:

```
public void act()
{
    if (atWorldEdge()) {
        turn(17);
    }
    move();
}
```

Die *if*-Anweisung ist eine Kontrollstruktur, die es dir ermöglicht, Befehle nur dann auszuführen, wenn eine bestimmte Bedingung erfüllt ist. Nur wenn die Methode *atWorldEdge()* den Wert *true* liefert, also nur wenn die Krabbe den Rand der Welt erreicht hat, soll sie sich wenden.

Übung 1.8:

- Implementiere den oben angegebenen Quelltext . Überprüfe, ob sich die Krabbe beim Erreichen des Weltrandes umdreht.
- Gib für den Parameter der Methode *turn()* verschiedene Parameter ein, bis dir das Ergebnis gefällt.
- Setze die Methode innerhalb der Klammern der *if*-Anweisung. Welche Auswirkungen hat dies und erkläre das zu beobachtende Verhalten. (Anschließend behebe diesen Fehler wieder.)

1.3. Abweichung vom Kurs

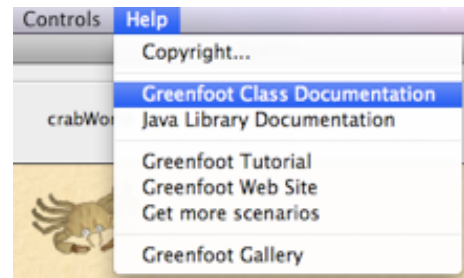
Die Krabbe hat bereits die Fähigkeit, über den Bildschirm zu laufen und am Rande der Welt zu wenden. Aber sie kann sich nur gerade aus bewegen. Dieses Verhalten wirst du nun ändern, indem du eine Zufallskomponente einbaust: die Krabbe soll die meiste Zeit gerade aus laufen und gelegentlich ein wenig vom Kurs abweichen.

Die Greenfoot-Entwickler stellen für dieses Zufallsverhalten einen Zufallszahlen-Generator zur Verfügung. Die Methode *getRandomNumber()* erwartet einen Parameter, der den oberen Grenzwert für die zurückzuliefernde Zahl angibt. Zum Beispiel liefert *getRandomNumber(20)* eine Zufallszahl im Bereich 0 bis 19, der Grenzwert 20 liegt nicht mehr im Bereich.

Die Methode *getRandomNumber()* findest du weder in der Klasse *Krabbe* noch in der Klasse *Animal*, sondern in einer Klasse namens *Greenfoot*.

Übung 1.9:

Wähle im Menü *Help* die Option *Greenfoot Class Documentation*. Suche nach der Methode *getRandomNumber()*. Lies die Dokumentation dieser Methode. Welchen Rückgabebetyp hat diese Methode?



Da die Methode weder zur Klasse *Krabbe* noch zu deren Superklassen gehört, musst du den Methodenaufruf mit der Punktnotation „*Greenfoot*.“ beginnen. So liefert die Anweisung `Greenfoot.getRandomNumber(100)` eine Zufallszahl zwischen 0 und 99.

Die Krabbe soll nun bei jedem Schritt mit einer Wahrscheinlichkeit von 10 % ein wenig vom Kurs abweichen. Ein Ausdruck, der in 10 % aller Fälle wahr ist, könnte zum Beispiel folgendermaßen lauten

$$\text{Greenfoot.getRandomNumber}(100) < 10$$

Diese Anweisung bedeutet, dass die gelieferten Zufallszahlen zwischen 0 und 99 in zehn Prozent aller Fälle unter 10 liegen. Wenn nun eine Zahl kleiner als 10 geliefert wird, soll die Krabbe eine kleine Drehung durchführen.

```
public void act()
{
    if (atWorldEdge()) {
        turn(17);
    }
    if (Greenfoot.getRandomNumber(100) < 10) {
        turn(5);
    }
    move();
}
```

Übung 1.10:

Implementiere diese zufälligen Kursänderungen in der Klasse *Krabbe*. Experimentiere mit verschiedenen Wahrscheinlichkeiten.

Du wirst noch zwei Schönheitsfehler beheben. Einerseits dreht sich die Krabbe, wenn sie sich dreht, immer um den gleichen Winkel 5°, andererseits dreht sie sich immer nach rechts und nie nach links.

Das erste Problem kannst du wieder mit Hilfe einer Zufallszahl beheben:

```
if (Greenfoot.getRandomNumber(100) < 10) {
    turn(Greenfoot.getRandomNumber(45));
}
```

Nun dreht sich die Krabbe um einen zufälligen Winkel, der zwischen 0° und 45° liegt.

Übung 1.11:

Implementiere diese zufälligen Kursänderungen um zufällige Winkel in der Klasse *Krabbe*.

- Dreht sich die Krabbe um verschiedene Winkel, wenn sie sich dreht.
- Setze zwei Krabben in die Welt. Drehen sie sich zur selben Zeit oder unabhängig voneinander? Warum?

Übung 1.12:

Die Krabbe dreht sich noch immer nach rechts, was nicht dem normalen Verhalten einer Krabbe entspricht.

- Ändere deinen Quelltext so, dass sich die Krabbe bei jeder Drehung entweder nach rechts oder nach links um bis zu 45° dreht.
- Setze zwei Krabben in die Welt. Drehen sie sich zur selben Zeit oder unabhängig voneinander?

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *krabbe3*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

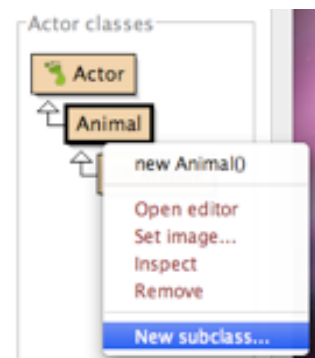
1.4. Das grosse Fressen

Um zu überleben, müssen Krabben auch fressen. Ich gehe davon aus, dass Würmer die Lieblingsspeise von Krabben sind. Deshalb wirst du nun eine neue Klasse *Wurm* erstellen. Da zwischen den Klassen *Wurm* und *Animal* eine *ist-ein*-Beziehung besteht (ein Wurm ist ein Animal), ist diese neue Klasse *Wurm* eine Subklasse der Klasse *Animal*.

Übung 1.13:

Wähle im Kontextmenü der Klasse *Animal* die Option *New subclass ...*

- Wenn du eine neue Subklasse erzeugst, wirst du aufgefordert, einen Namen für die Klasse einzugeben. Wähle den Namen *Wurm*. (Per Konvention beginnt dieser Name mit einem Großbuchstaben und steht in der Einzahl.)



- b) Dann solltest du dieser Klasse ein Bild zuweisen. Wähle hierfür das Bild *worm.png*.
- c) Drücke anschließend auf *Ok*. Damit wird die Klasse *Wurm* in das Szenario eingefügt. Entwerfe nun sinnvolle Kommentare im Quelltext und kompiliere die Klasse.
- d) Füge einige Würmer in die Welt ein. Füge einige Krabben hinzu. Was kannst du beobachten, wenn eine Krabbe auf einen Wurm trifft?

Du musst nun dafür sorgen, dass die beiden Klassen *Krabbe* und *Wurm* miteinander kommunizieren. Wenn eine Krabbe bei ihrer Bewegung über den Strand einen Wurm findet, soll sie diesen fressen.

Übung 1.14:

Suche nach zwei geeigneten Methoden in den Superklassen der Klasse *Wurm*. Studiere deren Beschreibung. Welche Parameter werden benötigt? Welche Rückgabetypern werden geliefert?

Wenn du neugierig bist, wie diese Methoden von den Greenfoot-Entwicklern programmiert wurde, kannst du auf *Source Code* umschalten.

Die Methode *canSee()* liefert *true* zurück, wenn die Krabbe genau dort, wo sie sich gerade befindet, ein Objekt der Klasse *cls* sieht. Die Krabbe kann einen Wurm nur sehen, wenn sie direkt darauf trifft - Krabben sind extrem kurzsichtig. Mit der Methode *eat()* frisst sie ein Objekt der Klasse *cls*.

```
if (canSee(Wurm.class)) {  
    eat(Wurm.class);  
}
```

Übung 1.15:

Implementiere den oben stehenden Quelltext in die Methode *act()* der Klasse *Krabbe*. Teste dein Szenario. Platziere hierzu einige Würmer ein paar Krabben in die Welt und schaue zu, was passiert.

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *krabbe4*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

1.5. Es werde Ordnung



Die Krabbe besitzt mittlerweile eine große Anzahl von Fertigkeiten - am Rand der Welt wenden, gelegentlich beliebig große Drehbewegungen machen und Würmer fressen. Je mehr die Krabbe für die Welt gerüstet ist, desto unübersichtlicher wird die Methode *act()* in der Klasse. Anstatt alle Fertigkeiten der Krabbe in diese Methode zu legen, wird du Ordnung schaffen. Du schreibst für jede Fertigkeit der Krabbe eine separate Methode, die in der *act()*-Methode aufgerufen wird.

```
/**
 * Prueft, ob die Krabbe auf einen Wurm gestoßen ist.
 * Wenn ja, wird dieser gefressen. Ansonsten passiert nichts.
 */
public void sucheWurm()
{
    if (canSee(Wurm.class)) {
        eat(Wurm.class);
    }
}
```

Diese neu definierte Methode wird nicht sofort ausgeführt, sie definiert lediglich eine potenzielle Aktion, die nur ausgeführt wird, wenn in der Methode *act()* der Aufruf *sucheWurm()*; eingefügt wird. Beachte, dass zu dem Aufruf auch die Klammer für die (leere) Parameterliste gehört.

Übung 1.16:

Implementiere die oben stehende Methode *sucheWurm()* in der Klasse *Krabbe*. Rufe diese in der Methode *act()* auf. Teste dein Szenario.

Beachte, dass diese Änderungen am Quelltext keinen Einfluss auf das Verhalten der Krabben haben. Je mehr Fertigkeiten die Krabbe erhält, desto länger wird die Methode *act()*. Indem du den Quelltext in mehrere kurze Methoden zerlegst, wird er jedoch leichter lesbar.

Bemerkung:

Beachte, dass in Java Methodennamen standardmäßig mit einem Kleinbuchstaben beginnen. Sie dürfen keine Leerzeichen enthalten. Besteht der Name aus mehreren Wörtern, dann verwende Großbuchstaben in der Mitte des Namens, um den Anfang eines jeden Worts zu kennzeichnen.

Übung 1.17:

Implementiere eine weitere neue Methode *dreheZufaellig()* in der Klasse *Krabbe*. Rufe diese in der Methode *act()* auf. Schreibe einen Kommentar zu dieser Methode. Teste dein Szenario.

Übung 1.18:

Implementiere eine weitere neue Methode *wendeAmRand()* in der Klasse *Krabbe*. Rufe diese in der Methode *act()* auf. Schreibe einen Kommentar zu dieser Methode. Teste dein Szenario.

Deine Methode *act()* sollte nun wie folgt aussehen:

```
public void act()
{
    wendeAmRand();
    dreheZufaellig();
    move();
    sucheWurm();
}
```

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *krabbe5*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

1.6. Gefahr in Verzug

Nun wirst du die Simulation interessanter gestalten. Du sollst ein weiteres Tier in das Szenario aufnehmen: einen Hummer. Hummer lieben es, Krabben zu jagen und sie zu fressen. Für die Krabben wird es nun wirklich gefährlich.

Übung 1.19:

Füge deinem Szenario eine neue Klasse hinzu. Sie hat den Namen *Hummer*. Sie ist eine Subklasse von *Animal* und wird mit dem bereits vorgegebenen Bild *lobster.png* verbunden.

Ein Hummer hat das gleiche Verhalten wie eine Krabbe. Er läuft wahllos auf dem Strand umher und wendet, wenn er an den Rand der Welt kommt. Der einzige Unterschied ist, dass der Hummer auf der Suche nach Krabben ist, um seinen Hunger zu stillen.

Übung 1.20:

- a) Kopiere die Methode *act()*, sowie die Methoden *wendeAmRand()*, *dreheZufaellig()* und *sucheWurm()* aus der Klasse *Krabbe* in die Klasse *Hummer*.

- b) Ändere den Quelltext der Klasse *Hummer* so, dass er nicht nach Würmern, sondern nach Krabben sucht. Denke auch daran, deine Kommentare anzupassen.
- c) Platziere eine Krabbe, drei Hummer und viele Würmer in die Welt und führe das Szenario aus. Schafft es die Krabbe, alle Würmer zu fressen, bevor sie von einem Hummer gefressen wird.

Die oben dargestellte Situation ist nicht ganz ungefährlich für die Krabbe. In der Regel wird sie doch recht schnell von einem der drei Hummer gefressen. Doch wenn du als Spieler die Bewegung der Krabbe beeinflussen könntest, hat die Krabbe gegen die drei Hummer vielleicht eine Chance. Das zufällige Verhalten der Krabbe wird nun durch eine Tastatursteuerung ersetzt.

Übung 1.21:

- a) Suche in der Klasse *Greenfoot* (erreichbar unter dem Menüpunkt Help > Greenfoot Class Documentation) nach der Methode *isKeyDown()*. Welchen Rückgabetyt liefert sie? Welcher Parameter muss ihr übergeben werden? Finde auch heraus, welche Tastennamen (key names) erlaubt sind?
- b) Entferne in der Klasse *Krabbe* den Quelltext, der für die zufälligen Drehbewegungen und für die Wende am Rand der Welt zuständig ist.
- c) Entwerfe eine Methode *pruefeTaste()*, die dafür sorgt, dass die Krabbe eine Drehung um 4° nach links durchführt, wenn die linke Cursortaste gedrückt wird. Beachte, dass die linke Cursortaste „left“ genannt wird. Rufe anschließend die Methode *pruefeTaste()* in der Methode *act()* auf.
- d) Füge in der Methode *pruefeTaste()* nun einen ähnlichen Quellext ein, der dafür sorgt, dass die Krabbe nach rechts dreht, wenn die rechte Cursortaste gedrückt wird.
- e) Setze eine Krabbe, ein paar Würmer und einige Hummer in die Welt und teste dein Szenario. Gelingt es dir, dass die Krabbe alle Würmer fressen kann, bevor sie von einem Hummer gefressen wird.

Den Fressvorgang kannst du noch etwas dramatisieren, indem du zum Beispiel Sounds hinzufügst.

Übung 1.22:

- a) Suche in der Klasse *Greenfoot* (erreichbar unter dem Menüpunkt Help > Greenfoot Class Documentation) nach einer Methode, mit der du einen Sound abspielen kannst. Wie heißt diese Methode? Welche Parameter erwartet sie?
- b) In dem Krabben-Szenario sind bereits die beiden Sounddateien *slurp.wav* und *au.wav* enthalten, die folgendermaßen aufgerufen werden können:

```
Greenfoot.playSound(„slurp.wav“);
```

Wenn eine Krabbe eine Wurm frisst, spiele den Sound *slurp.wave*, und wenn ein Hummer die Krabbe frisst, spiele den Sound *au.wav*.

Deine Methode *act()* sollte nun wie folgt aussehen:

```
public void act()
{
    pruefeTaste();
    move();
    sucheWurm();
}
```

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *krabbe6*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

1.7. Die Krabbe wird hyperaktiv

Die Bewegung der Krabbe wirkt natürlicher, wenn sie beim Laufen die Beine bewegt. Der Trick bei der Animation ist recht einfach. Du verwendest zwei verschiedene Bilder der Krabbe, die sich in der Stellung der Beine unterscheiden. Das obere Bild zeigt die Krabbe mit ausgestreckten Beinen, das untere mit angezogenen Beinen. Wenn du diese beiden Bilder abwechselnd verwendest, sieht es so aus, als würden die Krabbe tatsächlich laufen.



Diese beiden Bilder befinden sich bereits im Ordner *images* deines Krabben-Szenarios und heißen *crab.png* und *crab2.png*. Das erste Krabbenbild hast du bereits bei der Erzeugung von Krabben verwendet. Jedes Objekt, das von der Klasse *Krabbe* erzeugt wird, erhält eine Kopie dieses Bildes. Das bedeutet jedoch nicht, dass alle Objekte derselben Klasse immer mit dem gleichen Bild verbunden sein müssen. Jedes Objekt kann selbst entscheiden, sein Bild zu ändern.

Übung 1.23:

Suche in der Dokumentation der Klasse *Actor* nach den Methoden, mir deren Hilfe die Bilder verändert werden können. Wie heißen sie? Welche Parameter brauchen sie? Welche Rückgabetypern liefern sie?

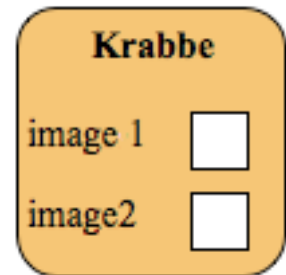
Greenfoot stellt eine Klasse namens *GreenfootImage* zur Verfügung, die die Verwendung und Verwaltung von Bildern erleichtert. Mit

```
new GreenfootImage("crab.png")
```

wird ein `GreenfootImage`-Objekt erzeugt, das in der Methode `setImage(image)` als Parameter verwendet werden kann.

Die Krabbe muss jedoch sich merken, durch welche Bilder sie dargestellt werden will. Solche Informationen werden in Variablen gespeichert. Du wirst später mehrere Arten von Variablen kennen lernen. Vorläufig reicht es aus, wenn du dir merkst, dass diese Bilder in Variablen speicherst, die Objektvariablen oder Datenfelder genannt werden. Alles, was in dieser Art von Variable gespeichert ist, steht so lange zur Verfügung, wie das Objekt existiert.

Es ist guter Stil, wenn die Deklaration von Datenfeldern immer am Anfang einer Klasse erfolgt. Eine Deklaration bedeutet nur, dass irgendwo im Speicher Platz geschaffen wurde, in den später die Werte dieser Variablen gespeichert werden können. Die Abbildung stellt deshalb zwei leere, weiße Kästchen dar, weil sie noch keine Image-Objekte enthalten.



```
import greenfoot.*;

public class Krabbe extends Animal
{
    //Datenfelder (Objektvariablen)
    private GreenfootImage image1;
    private GreenfootImage image2;

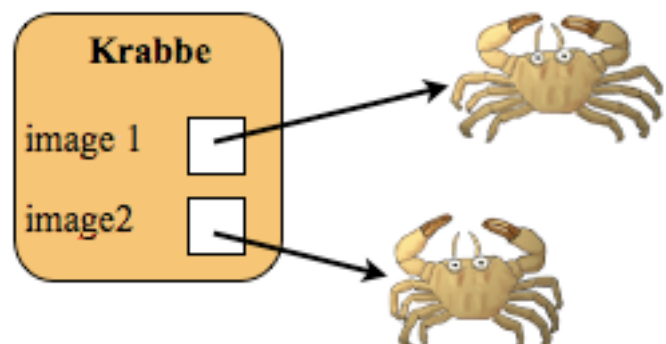
    //Methoden ausgelassen

}
```

Nun werden den beiden Datenfeldern die entsprechenden `GreenfootImage`-Objekte zugewiesen.

```
image1 = new GreenfootImage("crab.png");
image2 = new GreenfootImage("crab2.png");
```

Diese beiden Anweisungen erzeugen die beiden `GreenfootImage`-Objekte und speichern diese in die Datenfelder `image1` bzw. `image2`. Insgesamt hast du nun drei Objekte, nämlich eine Krabbe und zwei Bilder. Die Datenfelder der Krabbe enthalten Verweise (Referenzen) auf die Bilder.



An welcher Stelle deines Quelltextes soll diese Zuweisung erfolgen. Die Methode *act()* ist hierfür nicht geeignet, da es wohl nicht sinnvoll ist, bei jedem Aktionsschritt Bilder zu erzeugen und diese in Datenfelder zu speichern.

Eigentlich soll diese Zuweisung nur einmal ausgeführt werden, und zwar wenn die Krabbe erzeugt wird. Und genau hierfür wirst du nun den Konstruktor kennen lernen:

```
public class Krabbe extends Animal
{
    //Datenfelder (Objektvariablen)
    private GreenfootImage image1;
    private GreenfootImage image2;

    /**
     * Konstruktor erzeugt eine Krabbe und initialisiert ihre beiden Bilder.
     */
    public Krabbe()
    {
        image1 = new GreenfootImage("crab.png");
        image2 = new GreenfootImage("crab2.png");
        setImage(image1);
    }

    // weitere Methoden ausgelassen
}
```

Ein Konstruktor hat große Ähnlichkeit mit einer Methode, es gibt aber zwei wichtige Unterschiede:

Ein Konstruktor besitzt keinen Rückgabotyp, der zwischen dem Schlüsselwort *public* und dem Namen stehen würde.

Der Name der Konstruktors lautet immer genauso wie die Klasse selbst.

Ein Konstruktor ist somit eine besondere Methode, die immer automatisch ausgeführt wird, wenn ein Objekt dieser Klasse erzeugt wird. Dabei kann er alle Befehle ausführen, die nötig sind, um dieses Objekt in den gewünschten Anfangszustand zu versetzen.

Übung 1.24:

Deklariere in der Klasse *Krabbe* die beiden Datenfelder *image1* und *image2* und implementiere den Konstruktor.

- Teste dein Szenario. Stellst du Verhaltensveränderungen fest?
- Öffne den Inspektor eines Krabben-Objekts. Gibt es hier eine Veränderung?

Für die Animation der Bewegung musst du nun zwischen den beiden Bildern hin- und herwechseln. Wenn also gerade *image1* gezeigt wird, wechsele zu *image2* und umgekehrt.

```
public void wechsleBild()
{
    if (getImage() == image1) {
        setImage(image2);
    }
    else {
        setImage(image1);
    }
}
```

Beachte den Operator `==` (zwei Gleichheitszeichen). Er wird dazu verwendet, um einen Wert mit einem anderen zu vergleichen. Das Ergebnis ist entweder ***true*** oder ***false***.

Die *if*-Anweisung liegt auch in einer neuen Form vor. Sie enthält nun zwei Anweisungsblöcke, das heißt zwei Paare geschweifter Klammern: die *if*-Klausel und die *else*-Klausel. Wenn die *if*-Anweisung durchgeführt wird, wird zuerst die Bedingung ausgewertet. Liefert sie den Wert *true*, wird die *if*-Klausel ausgeführt und die Ausführung setzt mit den Anweisungen unterhalb der *else*-Klausel fort. Liefert dagegen die Bedingung den Wert *false*, wird statt der *if*-Klausel die *else*-Klausel ausgeführt. Auf diese Weise wird also immer genau einer der beiden Anweisungsblöcke ausgeführt.

Bemerkung:

Der *else*-Teil ist optional. Wenn du den *else*-Teil weglässt, erhältst du eine kürzere Form der *if*-Anweisung. Diese hast du bereits kennen gelernt und in der Methode *sucheWurm()* verwendet.

Übung 1.25:

Vervollständige den Quelltext der Klasse *Krabbe*, so dass die Bewegung der Krabbe animiert wird.

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *krabbe7*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

1.8. Das bittere Ende



Die letzte Aufgabe ist noch die ordnungsgemäße Beendigung des Spiels, wenn eine der folgenden Bedingungen erfüllt ist:

- die Krabbe wurde von einem Hummer gefangen oder
- die Krabbe hat acht Würmer gefressen.

Übung 1.26:

Suche in der Greenfoot Klassendokumentation eine geeignete Methode, die die Ausführung des laufenden Szenarios stoppt. Erwartet diese Methode Parameter? Wie lautet der Rückgabetyt?

Beim Suchen nach einer geeigneten Methode wirst du auf die Methode *stop()* gestoßen sein, mit der du das Szenario anhalten kannst.

Einfacher ist es, die erste Bedingung zur Beendigung des Spiels zu programmieren, wenn also der Hummer eine Krabbe gefressen hat.

Übung 1.27:

Füge deinem Szenario die Methode *stop()* hinzu, die das Spiel beendet, wenn die Krabbe von einem Hummer gefressen wird. Entscheide, wo du diese Methode einfügen musst. Teste dein Szenario!

Die zweite Bedingung zur Beendigung des Spiels ist etwas umfangreicher. Du benötigst hierzu

- ein Datenfeld, das die Zahl der gefressenen Würmer speichert,
- eine Anweisung, den Wert dieses Datenfelds um Eins zu vergrößern, wenn die Krabbe einen Wurm gefressen hat (inkrementieren) und
- eine Anweisung, die überprüft, ob die Krabbe acht Würmer gefressen hat und, wenn ja, das Spiel beendet.

Die erste Aufgabe erledigst du in folgender Übung.

Übung 1.28:

Deklariere das Datenfeld *anzahlWuermer*. Anschließend setzt du im Konstruktor mit dessen Wert auf Null mit *anzahlWuermer = 0*. Überprüfe deine Änderungen mit Hilfe des Objektinspektors.

Die beiden noch bleibenden Aufgaben müssen zusammen erledigt werden, wenn die Krabbe einen Wurm frisst, also in der Methode *sucheWurm()*. Hier fügst du die Zuweisung

```
anzahlWuermer = anzahlWuermer + 1;
```

die das Datenfeld *anzahlWuermer* um 1 inkrementiert.

Zur Überprüfung, ob die Krabbe bereits acht Würmer gefressen hat, verwendest du eine *if*-Anweisung. Ist dies der Fall, soll der Sound *fanfare.wav* abgespielt werden und die Ausführung beendet werden.

Übung 1.29:

Füge die oben besprochenen Anweisungen in dein Szenario ein. Teste deine *Simulation*. Als zusätzliche Kontrolle kannst du den Objektinspektor für das Krabben-Objekt öffnen. Lass den Inspektor geöffnet, während du spielst und beobachte das Datenfeld *anzahlWuermer*.

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *krabbe8*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

1.9. Zurück zu den Anfängen

Störend ist noch, dass du die Akteur, also die Krabbe sowie die Hummer und Würmer, manuell in der Welt platzieren musst. Dies soll nun beim Start des Spiels automatisch erfolgen.

Hierzu bietet sich die Klasse *CrabWorld* an. Wenn du dessen Editor öffnest, findest du folgenden Quelltext:

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

public class CrabWorld extends World
{
    /**
     * Erzeugt die Krabbenwelt (den Strand). Unsere Welt hat eine Groesse
     * von 560x560 Zellen, wobei jede Zelle nur ein Pixel gross ist.
     */
    public CrabWorld()
    {
        super(560, 560, 1);
    }
}
```

Im Konstruktor dieser Klasse wird die Welt mit der gewünschten Größe und Auflösung erzeugt.

Bemerkung:



Genauer genommen wird die gewünschte Größe und Auflösung mit dem Schlüsselwort *super* an den Konstruktor der Superklasse *World* weiter gereicht, die damit die Welt erzeugt.

Da dieser Konstruktor jedes Mal ausgeführt wird, wenn eine Welt erzeugt wird, ist dies die geeignete Stelle, auch gleichzeitig Akteure zu erzeugen.

```
public CrabWorld()
{
    super(560, 560, 1);
    addObject(new Krabbe(), 150, 100);
}
```

Diese Anweisung erzeugt in der Welt automatisch eine neue Krabbe an der Position $x = 150$ und $y = 100$.

Übung 1.30:

Füge in den Konstruktor der Klasse *CrabWorld* Anweisungen ein, die eine Krabbe und einen Hummer erzeugen.

Ergänze den Konstruktor, so dass insgesamt eine Krabbe, drei Hummer und zehn Würmer erzeugt werden. Diese sollen an zufälligen Positionen in der Welt gesetzt werden. Verwende für die Koordinaten Zufallszahlen, die du mit Hilfe der Klasse *Greenfoot* generierst.

Verschiebe alle Anweisungen, die die Akteur-Objekte erzeugen, in eine separate Methode *bevoelkereWelt()*, die im Konstruktor aufgerufen wird.

Bemerkung:

Eine Implementierung der bisher diskutierten Funktionalität findest du im Szenario *krabbe9*. Nach Beendigung deiner Implementierung (oder wenn du hängen geblieben bist), möchtest du vielleicht deine Lösung mit meiner vergleichen.

1.10. Und nun ...

Ich möchte an dieser Stelle die Programmierung des Spiels beenden, obwohl du bestimmt noch viele Möglichkeiten findest, es zu erweitern.

So könntest du beispielsweise

- weitere Akteure einfügen,
- Rückwärtsbewegungen möglich machen,
- das Spiel für zwei Spieler erweitern,
- für jeden gefressenen Wurm erscheinen weitere Würmer.