

KAPITEL

4

Das Spiel „Little Crab“ fertigstellen



Lernziele

Themen: Initialisierung der Welt, Setzen von Bildern, Animieren von Bildern

Konzepte: Konstruktoren, Zustand, Variablen (Instanzvariablen und lokale Variablen), Zuweisung, **new** (Objekte im Quelltext erzeugen)

In diesem Kapitel werden wir unser Krabben-Spiel abschließen. Wir betrachten unser Programm damit als „fertig“, da wir es nicht weiter besprechen. Natürlich ist ein Programm niemals fertig – jedes Programm lässt sich nachträglich um Funktionalität erweitern. Wir werden dir am Ende des Kapitels selbst einige Vorschläge dazu unterbreiten. Doch zuerst wollen wir unsere Verbesserungen ausführlich besprechen.

4.1 Objekte automatisch erzeugen

Allmählich nähert sich unser kleines Spiel einer spielbaren Version, auch wenn es noch einige Mängel zu beheben gilt. Als Erstes sollten wir das Problem angehen, dass wir die Akteure (d.h. die Krabbe sowie die Hummer und Würmer) immer manuell in der Welt platzieren müssen. Es wäre schöner, wenn dies automatisch geschähe.

Es gibt etwas, das bei jeder erfolgreichen Übersetzung automatisch abläuft: Die Welt selbst wird neu erzeugt. Das Welt-Objekt, wie wir es auf dem Bildschirm sehen (ein sandfarbener quadratischer Bereich), ist ein Objekt der Klasse **CrabWorld**. Welt-Objekte erfahren in Greenfoot eine Sonderbehandlung: Während wir die Objekte unserer Akteur-Klassen selbst erzeugen müssen, erzeugt das Greenfoot-System immer automatisch ein Objekt unserer Welt-Klasse und zeigt es auf dem Bildschirm an.

Werfen wir deshalb einen Blick in den Quelltext der Klasse **CrabWorld** (Listing 4.1). (Wenn du das Krabben-Spiel nicht nachvollzogen hast und zu diesem Zeitpunkt nicht über eine eigene Version verfügst, kannst du für dieses Kapitel *little-crab-4* verwenden.)

In dieser Klasse finden wir oben in der ersten Zeile wie gehabt die **import**-Anweisung. (Wir werden später noch ausführlich auf diese Anweisung zu spre-

chen kommen – im Moment reicht es zu wissen, dass diese Zeile immer oben in unseren Greenfoot-Klassen steht.)

Danach folgt der Klassenkopf und ein Kommentar (der Zeilenblock, der mit Sternchen beginnt; wir haben ihn bereits im vorherigen Kapitel angesprochen). Kommentare beginnen in der Regel mit dem Symbol `/**` und enden mit `*/`.

Listing 4.1:

Quelltext für die Klasse **CrabWorld**.

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

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

Erst danach kommt der interessante Teil:

```
public CrabWorld()
{
    super(560, 560, 1);
}
```

Dieser Teil wird auch als *Konstruktor* einer Klasse bezeichnet. Ein Konstruktor hat große Ähnlichkeit mit einer Methode, es gibt aber auch Unterschiede:

- Ein Konstruktor besitzt keinen Rückgabetypp, der zwischen dem Schlüsselwort **public** und dem Namen stehen würde.
- Der Name des Konstruktors lautet immer genauso wie die Klasse selbst.

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

In unserem Fall setzt der Konstruktor die Welt auf die von uns gewünschte Größe (560 mal 560 Zellen) und legt eine Auflösung fest (1 Pixel pro Zelle). Wir werden später im Buch noch näher auf die Auflösung der Welt eingehen.

Da dieser Konstruktor jedes Mal ausgeführt wird, wenn eine Welt erzeugt wird, können wir uns seiner bedienen, um unsere Akteure zu erzeugen. Wenn wir Code zur Erzeugung eines Akteurs in den Konstruktor einfügen, wird dieser Code automatisch mit ausgeführt. Zum Beispiel:

```
public CrabWorld()
{
    super(560, 560, 1);
    Crab myCrab = new Crab();
    addObject( myCrab, 250, 200 );
}
```

Konzept

Der **Konstruktor** einer Klasse ist eine besondere Art Methode, die immer automatisch ausgeführt wird, wenn ein Objekt dieser Klasse erzeugt wird.

Dieser Code erzeugt in der Welt automatisch eine neue Krabbe an der Position $x=250, y=200$. Die Position $(250,200)$ befindet sich 250 Zellen vom linken Rand der Welt entfernt und 200 Zellen vom oberen. Der Ursprung – d.h. der Punkt $(0,0)$ – unseres Koordinatensystems ist also die obere linke Ecke der Welt (Abbildung 4.1).

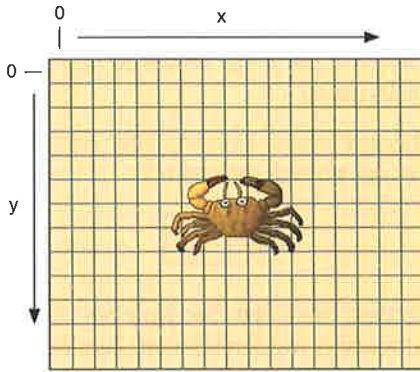


Abbildung 4.1
Das Koordinatensystem der Welt.

Wir führen hier vier neue Dinge ein: eine Variable, eine Zuweisung, die Anweisung **new** zur Erzeugung einer neuen Krabbe und die Methode **addObject**. Wir wollen diese Elemente nun eines nach dem anderen besprechen.

4.2 Neue Objekte erzeugen

Wenn wir eine neue Krabbe in die Welt setzen wollen, benötigen wir als Erstes eine Krabbe. In unserem Fall ist die Krabbe ein Objekt der Klasse **Crab**. Bisher haben wir Krabben-Objekte interaktiv erzeugt, indem wir einen Rechtsklick auf der Klasse **Crab** durchgeführt und dann **new Crab()** aus dem Kontextmenü ausgewählt haben.

Nun möchten wir, dass unser Konstruktorcode das neue Krabben-Objekt automatisch für uns erzeugt.

Mit dem Java-Schlüsselwort **new** können wir von jeder beliebigen Klasse neue Objekte erzeugen. So erzeugt zum Beispiel der Ausdruck

```
new Crab()
```

ein neues Objekt der Klasse **Crab**. Der Ausdruck, mit dem neue Objekte erzeugt werden, beginnt immer mit dem Schlüsselwort **new** gefolgt von dem Namen der Klasse, von der wir Objekte erzeugen wollen, und einer Parameterliste (die in unserem Beispiel leer ist). Die Parameterliste ermöglicht es uns, dem Konstruktor des neuen Objekts Parameter zu übergeben. Da wir keinen Konstruktor für unsere **Crab**-Klasse angegeben haben, ist die Standard-Parameterliste leer. (Du hast vielleicht bemerkt, dass der Befehl, den wir aus dem Kontextmenü der Klasse zum Erzeugen von Objekten ausgewählt haben, genau dieser Anweisung entspricht.)

In unserem obigen Konstruktor-Code findest du den Befehl **new Crab()** als rechte Hälfte der ersten Zeile, die wir eingefügt haben.

Wenn wir ein neues Objekt erzeugen, müssen wir etwas damit tun. In unserem Fall weisen wir ihn einer Variablen zu.

Konzept

Java-Objekte können im Programmquelltext mithilfe des Schlüsselwortes **new** erzeugt werden.

4.3 Variablen

Konzept

Variablen können benutzt werden, um Informationen (Objekte oder Werte) für die spätere Verwendung zu speichern.

Bei der Programmierung müssen wir oft Informationen (irgendwo) speichern, um sie später wiederverwenden zu können. Dazu verwenden wir *Variablen*.

Eine Variable ist ein kleiner Speicherplatz, der immer einen Namen besitzt, mit dem man auf ihn zugreifen kann. In unseren Zeichnungen von Objekten oder Codefragmenten stellen wir Variablen in der Regel als weiße Kästchen dar, den Namen der Variablen notieren wir links davon.

Abbildung 4.2 zeigt beispielsweise eine Variable namens **age**. (Wir möchten darin eventuell das Alter der Krabbe speichern.)

Abbildung 4.2
Eine (leere) Variable.



Variablen besitzen außerdem einen *Typ*. Der Typ einer Variablen sagt uns, welche Art von Daten in ihr aufbewahrt werden können. Zum Beispiel kann eine Variable vom Typ **int** ganze Zahlen speichern, eine Variable vom Typ **boolean** kann wahr/falsch-Werte speichern und eine Variable vom Typ **Crab** kann Krabben-Objekte speichern.

Konzept

Variablen können erzeugt werden, indem man eine **Variablendeklaration** für sie schreibt.

Wenn wir in unserem Quelltext eine Variable benötigen, dann erzeugen wir sie, indem wir eine *Variablendeklaration* schreiben. Eine Variablendeklaration ist sehr einfach: Wir schreiben nur den Typ und den gewünschten Namen der Variable auf, dann folgt noch ein Semikolon. Möchten wir zum Beispiel eine Variable **age** wie in Abbildung 4.2 haben, um ganze Zahlen darin zu speichern, dann können wir schreiben:

```
int age;
```

Damit wird unsere **age**-Variable erzeugt, die bereit ist, **int**-Werte aufzunehmen.

4.4 Zuweisungen

Sobald wir eine Variable besitzen, können wir etwas in ihr speichern. Dazu verwenden wir eine *Zuweisung* (*assignment*).

Eine Zuweisung ist ein Java-Befehl, der mit einem Gleichheitszeichen geschrieben wird.

Um zum Beispiel die Zahl 12 in unserer Variablen **age** zu speichern, können wir schreiben

```
age = 12;
```

Zuweisungen liest man am besten von rechts nach links: *Der Wert 12 wird in der Variable **age** gespeichert.* Nachdem diese Zuweisung ausgeführt wurde, enthält unsere Variable den Wert 12. In unserem Diagramm schreiben wir dazu den Wert in das weiße Kästchen (Abbildung 4.3).

Konzept

Wir können Werte in Variablen speichern, indem wir eine **Zuweisung** (=) verwenden.

```
age 12
```

Abbildung 4.3

Eine Variable, die einen Integerwert speichert.

Die allgemeine Form einer Zuweisung lautet

```
variable = expression;
```

Das heißt, auf der linken Seite steht immer der Name der Variablen und auf der rechten Seite ist ein Ausdruck, der ausgewertet wird. Das Ergebnis dieser Auswertung wird in der Variablen gespeichert.

In unseren Programmen möchten wir häufig Variablen deklarieren und Werte darin speichern. Deshalb tauchen Variablendeklarationen und Zuweisungen oft zusammen auf:

```
int age;  
age = 12;
```

Weil dies so häufig vorkommt, erlaubt uns Java, diese beiden Anweisungen zusammen in einer Zeile zu schreiben:

```
int age = 12;
```

Diese Zeile erzeugt die Integervariable und weist ihr den Wert 12 zu. Sie macht somit genau dasselbe wie die Zwei-Zeilen-Version oben.

Zuweisungen überschreiben jeden Wert, der vorher dort gespeichert wurde. Wenn wir also in unserer **age**-Variablen den Wert 12 gespeichert haben und dann schreiben

```
age = 42;
```

so wird die **age**-Variable jetzt den Wert 42 enthalten. Die 12 ist überschrieben und es gibt keine Möglichkeit, sie wiederzubekommen.

4.5 Objektvariablen

Wir haben oben schon erwähnt, dass Variablen nicht nur Zahlen, sondern auch Objekte speichern können.

Java unterscheidet *primitive Typen* und *Objektypen*. Primitive Typen sind einige wenige häufig verwendete Datentypen wie **int**, **boolean** und **char**. In **Anhang D** werden alle primitiven Typen von Java aufgeführt.

Auch jede Klasse in Java definiert einen Typ, dieser wird **Objektyp** genannt. So erhalten wir beispielsweise mit unserer Klasse **Crab** einen Typ **Crab**, unsere **Lobster**-Klasse definiert einen Typ **Lobster** und so weiter. Wir können Variablen von diesen Typen so deklarieren:

```
Crab myCrab;
```

Beachte auch hier wieder, dass wir wie eben zuerst den Typ schreiben (**Crab**), dann den Namen, den wir uns selbst ausdenken können (**myCrab**), und ein Semikolon.

Konzept

Variablen von **primitiven Typen** speichern Zahlen, boolesche Werte und Zeichen, Variablen von **Objektypen** speichern Objekte.

Sobald wir eine Objektvariable besitzen, können wir Objekte darin speichern. Zusammen mit unserer Anweisung aus Abschnitt 4.2, um ein Krabben-Objekt zu erzeugen, erhalten wir:

```
Crab myCrab;  
myCrab = new Crab();
```

Auch hier können wir dies in einer Zeile schreiben:

```
Crab myCrab = new Crab();
```

Diese Codezeile macht drei Dinge:

- Sie erzeugt eine Variable mit dem Namen **myCrab** vom Typ **Crab**.
- Sie erzeugt ein Krabben-Objekt (ein Objekt vom Typ **Crab**).
- Sie weist das Krabben-Objekt der Variablen **myCrab** zu.

Bei einer Zuweisung wird die rechte Seite der Zuweisung immer zuerst ausgeführt (das Krabben-Objekt wird erzeugt), dann findet die Zuweisung an die Variable auf der linken Seite statt.

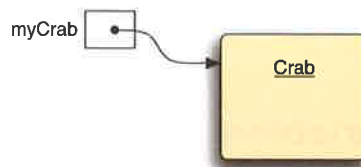
In unseren Diagrammen zeichnen wir Objektvariablen, in denen Objekte gespeichert sind, mithilfe eines Pfeils (Abbildung 4.4). Hier speichert die Variable **myCrab** eine Referenz auf das Krabben-Objekt. Die Tatsache, dass Objektvariablen immer Referenzen zu den Objekten speichern (und nicht die Objekte selbst), wird später wichtig werden. Deshalb sind wir sehr darauf bedacht, immer wie hier akkurat zu zeichnen.

Konzept

Objekte werden in Variablen gespeichert, indem eine **Referenz** auf das Objekt gespeichert wird.

Abbildung 4.4

Eine Objektvariable, die eine Referenz auf ein Objekt speichert.



Der Typ der Variable und der Typ des Werts, der ihr zugewiesen wird, müssen immer zueinander passen. Du kannst einen **int**-Wert einer **int**-Variablen zuweisen und du kannst ein **Crab**-Objekt einer **Crab**-Variablen zuweisen. Doch es ist nicht möglich, ein **Crab**-Objekt einer **int**-Variablen zuzuweisen (oder irgendeine andere nicht zueinander passende Kombination zu wählen).¹

¹ Wenn wir sagen, „die Typen müssen zueinander passen“, bedeutet dies nicht, dass sie identisch sein müssen. Es gibt Situationen, in denen Typen zueinander passen, die nicht identisch sind. Zum Beispiel können wir eine Unterklasse einem Oberklassentyp zuweisen, z.B. eine Krabbe einer **Actor**-Variablen (weil eine Krabbe ein Akteur *ist*). Doch dies sind Feinheiten, die wir später besprechen wollen.

Übung 4.1 Schreibe eine Variablendeklaration für eine Variable vom Typ **int**, wobei die Variable den Namen **score** haben soll.

Übung 4.2 Deklariere eine Variable namens **isHungry** vom Typ **boolean** und weise ihr den Wert **true** zu.

Übung 4.3 Deklariere eine Variable mit Namen **year** und weise ihr den Wert 2014 zu. Dann weise ihr den Wert 2015 zu.

Übung 4.4 Deklariere eine Variable vom Typ **Crab** namens **littleCrab** und weise ihr ein neues Krabben-Objekt zu.

Übung 4.5 Deklariere eine Variable vom Typ **Control** mit Namen **inputButton**, dann erzeuge ein Objekt vom Typ **Button** und weise es der Variablen zu.

Übung 4.6 Was ist falsch an folgender Anweisung? `int myCrab = new Crab();`

4.6 Variablen verwenden

Sobald wir eine Variable deklariert und ihr einen Wert zugewiesen haben, können wir sie benutzen, indem wir einfach den Namen der Variable schreiben.

Zum Beispiel deklariert der folgende Code zwei Integervariablen und weist ihnen Werte zu:

```
int n1 = 7;
int n2 = 13;
```

Wir können sie dann auf der rechten Seite einer anderen Anweisung einsetzen:

```
int sum = n1 + n2;
```

Nach Ausführung dieser Anweisung enthält **sum** die Summe von **n1** und **n2**. Wenn wir

```
n3 = n1;
```

schreiben, dann wird der Wert von **n1** (in diesem Fall 7) nach / zu **n3** kopiert (vorausgesetzt, es wurde vorher eine Variable **n3** deklariert). Die Variablen **n1** und **n3** enthalten nun beide den Wert 7.

Übung 4.7 Deklariere eine Variable mit dem Namen **children** (vom Typ **int**). Schreibe dann eine Zuweisung, die dieser Variable die Summe von zwei anderen Variablen zuweist, die **daughters** und **sons** heißen.

Übung 4.8 Deklariere eine Variable namens **area** vom Typ **int**. Dann schreibe eine Anweisung, die **area** das Produkt von zwei Variablen mit den Namen **width** und **length** zuweist.

Übung 4.9 Deklariere zwei Variablen **x** und **y** vom Typ **int**. Weise **x** den Wert 23 zu und **y** den Wert 17. Dann schreibe einen Code, der diese Werte vertauscht (sodass anschließend **x** den Wert 17 und **y** den Wert 23 enthält).

4.7 Objekte zur Welt hinzufügen

Wir haben jetzt gesehen, wie wir eine neue Krabbe erzeugen und sie in einer Variablen speichern können. Nun fehlt nur noch, diese neue Krabbe zu unserer Welt hinzuzufügen.

In dem Codefragment des Konstruktorcodes aus Abschnitt 4.1 haben wir gesehen, dass wir dazu die folgende Zeile verwenden können:

```
addObject(myCrab, 250, 200);
```

Die Methode **addObject** ist eine Methode der Klasse **World**, die es uns erlaubt, der Welt ein Akteur-Objekt hinzuzufügen. Wir können dies überprüfen, indem wir einen Blick in die Klassendokumentation der Klasse **World** werfen. Dort erfahren wir außerdem, dass die Signatur wie folgt lautet:

```
void addObject(Actor object, int x, int y)
```

Von Anfang bis Ende gelesen verrät uns diese Signatur Folgendes:

- Die Methode liefert kein Ergebnis zurück (Rückgabotyp **void**).
- Der Name der Methode ist **addObject**.
- Die Methode übernimmt drei Parameter: **object**, **x** und **y**.
- Der Typ des ersten Parameters ist **Actor**, der Typ der beiden letzten Parameter **int**.

Diese Methode kann dazu verwendet werden, einen neuen Akteur in der Welt anzulegen. Da die Methode zur Klasse **World** gehört und **CrabWorld** von **World** abgeleitet wurde (d.h., **CrabWorld** erbt von **World**), steht diese Methode auch in unserer Klasse **CrabWorld** zur Verfügung und wir können sie einfach aufrufen.

Wir haben jetzt eine neue Krabbe erzeugt und in unserer **myCrab**-Variablen gespeichert. Nun können wir diese Krabbe als ersten Parameter des **addObject**-Methodenaufrufs verwenden (über die Variable, in der sie gespeichert ist). Die

anderen beiden Parameter geben die **x**- und die **y**-Koordinate der Position an, an der wir das Objekt einfügen möchten.

All diese Konstrukte (Variablendeklaration, Objekterzeugung, Zuweisung und Hinzufügen des Objekts zur Welt) sehen gemeinsam so aus:

```
Crab myCrab = new Crab();  
addObject(myCrab, 250, 200);
```

Wir dürfen ein Objekt vom Typ **Crab** für den **Actor**-Parameter benutzen, da eine Krabbe ein Akteur ist (die Klasse **Crab** ist eine Subklasse der Klasse **Actor**).

Übung 4.10 Füge in den **CrabWorld**-Konstruktor deines eigenen Projekts Code ein, um wie oben beschrieben automatisch eine Krabbe zu erzeugen.

Übung 4.11 Füge in der **CrabWorld** Code ein, um automatisch drei Hummer zu erzeugen. Du kannst dafür beliebige Positionen in deiner Welt wählen.

Übung 4.12 Füge Code hinzu, um zwei Würmer an beliebigen Positionen in der **CrabWorld** zu erzeugen.

4.8 Die Welt speichern

Wir werden nun eine einfachere Methode einführen, um dasselbe zu erreichen.

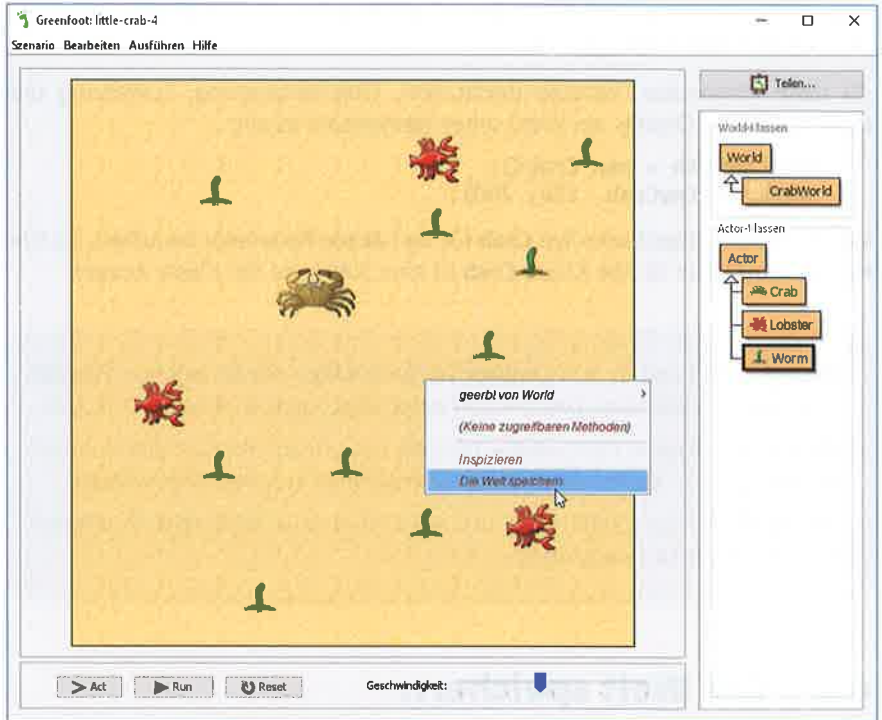
Zunächst entfernen wir den Code wieder, den wir in den letzten Übungen eingefügt haben, sodass keine Objekte automatisch erzeugt werden. Wenn du dein Szenario noch einmal kompilierst, sollte die Welt leer sein. Dann bearbeite die folgenden Übungen.

Übung 4.13 Kompiliere dein Szenario. Dann platziere die folgenden Akteure (interaktiv) in deine Welt: eine Krabbe, drei Hummer und zehn Würmer.

Übung 4.14 Führe einen Rechtsklick auf dem Welthintergrund aus. Das zur Welt gehörende Kontextmenü wird erscheinen. Wähle aus diesem Menü **DIE WELT SPEICHERN** aus.

Wenn du ein paar Objekte in deine Welt gesetzt hast und die Funktion **DIE WELT SPEICHERN** auswählst (Abbildung 4.5), wirst du bemerken, dass sich dein **CrabWorld**-Quelltext öffnet und ein wenig neuer Code in dieser Klasse eingefügt wurde. Untersuche diesen Code gründlich.

Abbildung 4.5
Die Funktion DIE WELT
SPEICHERN.



Du wirst feststellen, dass der Code Folgendes macht:

- Der Konstruktor enthält einen Aufruf an eine neue Methode namens **prepare()**.
- Für diese Methode ist eine Methodendefinition hinzugekommen.
- Die **prepare()**-Methode enthält Code, der alle Akteure erzeugt und hinzufügt, die wir eben interaktiv in der Welt platziert hatten.

Was passiert hier also?

Wenn wir Objekte interaktiv erzeugen und dann DIE WELT SPEICHERN auswählen, schreibt Greenfoot Code in unsere Welt-Klasse, der genau die Situation nachbildet, die wir von Hand eingegeben hatten. Dazu wird eine Methode namens **prepare()** erzeugt und aufgerufen.

Dies bewirkt, dass nun jedes Mal, wenn wir ÜBERSETZEN oder RESET anklicken, die Akteure sofort wieder erzeugt werden.

Unsere letzten Übungen, in denen wir den Code manuell geschrieben haben, um die Akteure zu erzeugen und zu platzieren, hilft uns jetzt zu verstehen, wie diese Methode arbeitet. In vielen Fällen müssen wir diesen Code nicht von Hand schreiben – und können stattdessen DIE WELT SPEICHERN verwenden –, aber es ist wichtig, ihn im Detail verstanden zu haben. Es gibt andere Situationen, in denen wir einen anspruchsvolleren Aufbau benötigen und den Initialisierungscode noch von Hand schreiben werden.

4.9 Bilder animieren

Nachdem wir es nun geschafft haben, unser Spiel mit einer guten Einstellung automatisch starten zu lassen, können wir jetzt ein bisschen Zeit investieren, einige Details zu verbessern.

Als Nächstes arbeiten wir daran, das Bild der Krabbe zu animieren. Damit die Bewegung der Krabbe etwas natürlicher wirkt, wollen wir die Krabbe so überarbeiten, dass sie beim Laufen die Beine bewegt.

Für die Animation wenden wir einen sehr einfachen Trick an: Wir nehmen zwei verschiedene Bilder der Krabbe (in unserem Szenario heißen sie `crab.png` und `crab2.png`) und wechseln einfach für die Darstellung der Krabbe sehr schnell zwischen diesen beiden Versionen hin und her. Die beiden Bilder unterscheiden sich lediglich in der Stellung der Krabbenbeine (Abbildung 4.6).

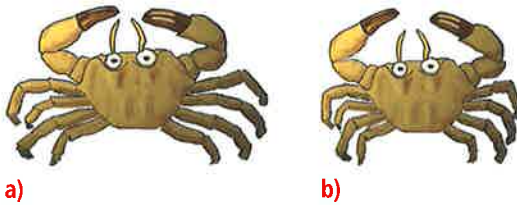


Abbildung 4.6

Zwei leicht abgewandelte Bilder der Krabbe
a) mit ausgestreckten Beinen
b) mit angezogenen Beinen.

Durch das Alternieren der Bilder wird der Eindruck erweckt, als wenn die Krabbe ihre Beine bewegen würde. Hierzu benötigen wir ein paar weitere Variablen und müssen uns außerdem ansehen, wie wir mit Greenfoot-Bildern arbeiten können.

4.10 Greenfoot-Bilder

Greenfoot stellt eine Klasse namens **GreenfootImage** bereit, die die Verwendung und Verwaltung von Bildern erleichtert. Wir erhalten ein Bild, indem wir mithilfe des Java-Schlüsselwortes **new** ein neues **GreenfootImage**-Objekt erzeugen („konstruieren“) und dazu dem Konstruktor den Namen der Bilddatei als Parameter übergeben. Um zum Beispiel auf das Bild `crab2.png` zuzugreifen, schreiben wir:

```
new GreenfootImage("crab2.png")
```

Die hier angeführte Datei muss in dem `images`-Ordner des Szenarios stehen.

Alle Greenfoot-Akteure verfügen über ein Bild. Standardmäßig erhalten die Akteure ihr Bild von ihrer Klasse. Wir weisen der Klasse ein Bild bei ihrer Erzeugung zu und anschließend erhält jedes Objekt, das von dieser Klasse erzeugt wird, eine Kopie dieses Bildes. Das bedeutet jedoch nicht, dass alle Objekte derselben Klasse immer mit dem gleichen Bild verbunden sein müssen. Jeder einzelne Akteur kann jederzeit entscheiden, sein Bild zu ändern.

Konzept

Greenfoot-Akteure verwalten ihr sichtbares Bild in Form eines Objekts vom Typ **GreenfootImage**.

Übung 4.15 Werf einen Blick in die Dokumentation der **Actor**-Klasse. Dort findest du zwei Methoden, die es uns erlauben, das Bild eines Akteurs zu ändern. Wie heißen sie und wie lauten ihre Parameter? Was liefern sie zurück?

Wenn du die Übung oben nachvollzogen hast, dann ist dir sicher aufgefallen, dass eine der Methoden, die ein Bild mit einem Akteur verbinden, einen Parameter vom Typ **GreenfootImage** erwartet. Diese Methode wollen wir hier verwenden. Wir können ein **GreenfootImage**-Objekt wie oben beschrieben aus einer Bilddatei erzeugen und diesem einer Variable vom Typ **GreenfootImage** zuweisen. Dann verwenden wir die **setImage**-Methode des Akteurs, um es für den Akteur zu benutzen. Der Code dazu sieht folgendermaßen aus:

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

Um das Bild zurück auf das Original zu setzen, schreiben wir:

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

Dieser Code erzeugt das Bildobjekt aus den Bilddateien namens *crab.png* und *crab2.png* und weist sie den Variablen **image1** und **image2** zu. Dann benutzen wir diese Variablen, um unser neues Bild als Akteur-Bild einzusetzen. Um den Animationseffekt zu erzeugen, müssen wir nur irgendwie bewerkstelligen, dass diese beiden Codefragmente abwechselnd ausgeführt werden: erst das eine, dann das andere, immer hin und her.

Wir könnten nun fortfahren und der **act**-Methode weiteren Code wie hier hinzufügen. Doch vorher wollen wir noch eine Verbesserung einführen: Wir möchten die Erzeugung der Bildobjekte von der Initialisierung des Bilds trennen.

Dies geschieht aus Gründen der Effizienz: Wenn unser Programm mit der Bildanimation ausgeführt wird, dann werden wir das Bild sehr oft ändern, mehrere Male pro Sekunde. Mit dem Code, so wie wir ihn geschrieben haben, würden auch das Lesen des Bildes aus der Bilddatei und die Erzeugung des Bildobjekts viele Male durchgeführt. Das ist nicht notwendig, es ist sogar verschwenderisch. Es reicht aus, die Bildobjekte einmal zu erzeugen und dann nur das Hin- und Herwechseln viele Male zu wiederholen. Mit anderen Worten, wir möchten die Codefragmente folgendermaßen voneinander trennen:

Führe dies nur einmal am Anfang aus:

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

Führe dies immer wieder aus:

```
setImage(image1);
```

oder

```
setImage(image2);
```

Wir erzeugen also zuerst die Bilder und speichern diese, später werden wir die gespeicherten Bilder dann immer wieder verwenden (ohne sie neu zu erzeugen), um das animierte Bild darzustellen.

Hierzu benötigen wir ein neues Konstrukt, das wir bisher noch nicht benutzt haben: eine Instanzvariable.

4.11 Instanzvariablen (Zustandsfelder)

Java stellt verschiedene Arten von Variablen zur Verfügung. Die Variablen, die wir bislang gesehen haben, heißen *lokale Variablen* und die Variablen, die wir nun besprechen wollen, sind *Instanzvariablen*. (Instanzvariablen werden manchmal auf *Zustandsfelder* genannt.)

Der erste Unterschied zwischen diesen beiden ist der Ort, an dem sie in unserem Quelltext deklariert werden (Listing 4.2): die lokalen Variablen werden innerhalb einer Methode deklariert, während Instanzvariablen innerhalb der Klasse deklariert werden, aber noch vor allen anderen Methoden.

```
public class Crab extends Actor
{
    private GreenfootImage image1;
    private GreenfootImage image2;
    private int age;

    /**
     * Lässt unsere Krabbe agieren.
     */
    public void act()
    {
        boolean isAlive;
        int n;

        // Code für die Aktionen der Krabbe ausgelassen
    }
}
```

Listing 4.2

Instanzvariablen und lokale Variablen in einer Klasse.

Konzept

Instanzvariablen (auch **Zustandsfelder** genannt) sind Variablen, die zu einem Objekt gehören (anstatt zu einer Methode).

Der nächste leicht sichtbare Unterschied ist, dass vor den Instanzvariablen das Schlüsselwort **private** steht (Listing 4.2).

Wichtiger ist jedoch der Unterschied im Verhalten: lokale Variablen und Instanzvariablen verhalten sich unterschiedlich, besonders hinsichtlich ihrer *Lebenszeit*.

Lokale Variablen gehören zu der Methode, in der sie deklariert sind, und verschwinden, sobald die Methode die Ausführung beendet hat. Jedes Mal, wenn wir die Methode aufrufen, werden die Variablen auf ein Neues erzeugt und können verwendet werden, während die Methode ausgeführt wird, aber zwischen Methodenaufrufen existieren sie nicht. Werte oder Objekte, die in ihnen gespeichert sind, sind verloren, wenn die Methode endet.²

Instanzvariablen andererseits gehören zu dem Objekt, in dem sie deklariert sind, und überleben so lange, wie das Objekt existiert. Sie können immer wieder benutzt werden, in vielen Methodenaufrufen und von mehreren Methoden. Wenn wir also möchten, dass ein Objekt Information eine längere Zeit speichert, dann benötigen wir eine Instanzvariable.

Konzept

Lebenszeit von Instanzvariablen: Instanzvariablen bestehen so lange, wie das Objekt existiert, das diese Variable hält.

Konzept

Lebenszeit von lokalen Variablen: Lokale Variablen existieren nur während einer einzigen Methodenaufrufung.

² Um genau zu sein: lokale Variablen gehören zum Gültigkeitsbereich, in dem sie deklariert sind, und existieren nur bis zum Ende dieses Bereichs. Häufig ist dies eine Methode, aber falls die Variable zum Beispiel innerhalb einer **if**-Anweisung deklariert ist, dann wird sie am Ende der **if**-Anweisung verworfen.

Instanzenvariablen werden am Anfang der Klasse, direkt nach dem Klassen-Header, unter Verwendung des Schlüsselworts **private** definiert,³ gefolgt vom Typ der Variable und dem Variablennamen:

private *variablentyp* *variablenname*;

Da wir hier Objekte vom Typ **GreenfootImage** speichern möchten, ist der Variablentyp **GreenfootImage** und wir benutzen die Namen **image1** und **image2** wie in unserem Codefragment oben (Listing 4.3).

Listing 4.3:

Die **Crab**-Klasse mit zwei Instanzvariablen.

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

public class Crab extends Actor
{
    private GreenfootImage image1;
    private GreenfootImage image2;

    // Methoden ausgelassen
}
```

Übung 4.16 Bevor du diesen Code hinzufügst, klicke mit der rechten Maustaste auf ein Krabben-Objekt in deiner Welt und wähle aus dem aufspringenden Kontextmenü der Krabbe den Befehl **INSPIZIEREN**. Notiere dir alle Variablen, die im Krabben-Projekt angezeigt werden.

Übung 4.17 Warum glaubst du, hat die Krabbe überhaupt irgendwelche Variablen, auch wenn wir keine in unserer Klasse **Crab** deklariert haben?

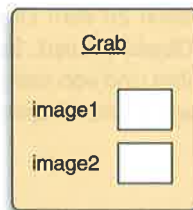
Übung 4.18 Füge die Variablendeklarationen aus Listing 4.3 in deine Version der Klasse **Crab** ein. Achte darauf, dass sich die Klasse übersetzen lässt.

Übung 4.19 Nachdem du die Variablen ergänzt hast, erzeuge und inspiziere erneut dein Krabben-Objekt. Notiere dir die Variablen und ihre Werte (die in den weißen Kästchen angezeigt werden).

In unseren Abbildungen stellen wir Objekte als farbige Kästen mit abgerundeten Ecken und Instanzvariablen als weiße Kästchen innerhalb eines Objekts dar (Abbildung 4.7). Beachte, dass wir durch die Deklaration dieser beiden **GreenfootImage**-Variablen noch keine zwei **GreenfootImage**-Objekte erhalten, sondern lediglich zwei leere Kästchen, in denen wir zwei Objekte speichern können.

Abbildung 4.7

Ein Krabben-Objekt mit zwei leeren Instanzvariablen.



³ Es ist in Java nicht unbedingt notwendig, dass Instanzvariablen am Anfang der Klasse stehen, aber wir werden dies trotzdem immer tun, da es als guter Stil gilt und uns hilft, die Variablendeklarationen leicht zu finden, wenn wir einen Blick darauf werfen möchten.

Als Nächstes müssen wir die beiden Bildobjekte erzeugen und in den Instanzvariablen speichern. Wie die Anweisung aussieht, mit der diese Objekte erzeugt werden, wurde bereits oben gezeigt. Das von uns dazu verwendete Codefragment lautete:

```
new GreenfootImage("crab2.png")
```

Jetzt müssen wir nur noch beide Bildobjekte erzeugen und diesen unseren Instanzvariablen zuweisen:

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

Nach Ausführung dieser Anweisungen haben wir drei Objekte (eine Krabbe und zwei Bilder), wobei die Variablen der Krabbe Verweise auf die Bilder enthalten. Abbildung 4.8 verdeutlicht dies.

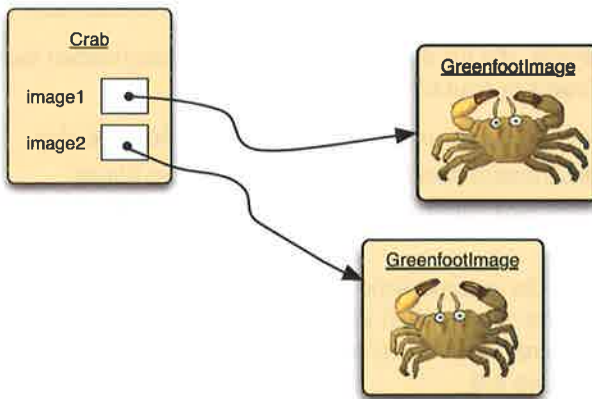


Abbildung 4.8
Ein Krabben-Objekt mit zwei Variablen, die auf Bildobjekte zeigen.

Die letzte offene Frage ist nun, wo der Code einzufügen ist, der die Bilder erzeugt und in Variablen speichert. Da dies nur einmal bei der Erzeugung des Krabben-Objekts erfolgen soll und nicht bei jedem Aktionsschritt, können wir den Code nicht in der `act`-Methode unterbringen. Stattdessen fügen wir den Code in einen Konstruktor ein.

4.12 Die Konstruktoren der Akteur-Klassen

Am Anfang dieses Kapitels haben wir gesehen, wie mithilfe des Konstruktors der Welt-Klasse die Welt initialisiert wurde. Auf ähnliche Weise können wir mit einem Konstruktor einer Akteur-Klasse den Akteur initialisieren. Der Code im Konstruktor wird nur einmal ausgeführt, und zwar wenn der Akteur erzeugt wird. Listing 4.4 zeigt einen Konstruktor für die Klasse `Crab`, der die beiden Instanzvariablen initialisiert, indem er Bilder erzeugt und diese den Variablen zuweist.

Listing 4.4:

Die Variablen im Konstruktor initialisieren.

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

public class Crab extends Actor
{
    private GreenfootImage image1;
    private GreenfootImage image2;

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

    // Methoden ausgelassen
}
```

Die gleichen Regeln, die für den **World**-Konstruktor beschrieben wurden, gelten auch für den **Crab**-Konstruktor:

- Die Signatur eines Konstruktors beinhaltet keinen Rückgabetyp.
- Der Name des Konstruktors lautet wie der Name der Klasse.
- Der Konstruktor wird automatisch ausgeführt, wenn ein Krabben-Objekt erzeugt wird.

Die letzte Regel – die automatische Ausführung des Konstruktors – stellt sicher, dass die Bildobjekte automatisch erzeugt und zugewiesen werden, wenn eine Krabbe erzeugt wird. Nach der Erzeugung der Krabbe entspricht die Situation also der in Abbildung 4.8.

Fallstricke

Beachte, dass bei der Zuweisung im Konstruktor vor dem Variablennamen keine Typangabe steht. Die Variable wird vor dem Konstruktor mit der Anweisung

```
private GreenfootImage image1;
```

definiert, im Konstruktor wird ihr mit der Zeile

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

ein Wert zugewiesen.

Schreiben wir stattdessen im Konstruktor

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

dann passiert etwas völlig anderes: Wir hätten im Konstruktor eine *zusätzliche* lokale Variable mit Namen **image1** deklariert (wir haben dann zwei Variablen, die **image1** heißen: eine *lokale* und eine *Instanzvariable*) und würden unser Bild der lokalen Variable zuweisen. Sobald der Konstruktor endet, wäre das Bild dann verloren, während unsere Instanzvariable weiterhin leer ist.

Dies ist ein sehr subtiler Fehler, der schnell passiert und schwierig zu finden ist. Stelle also sicher, dass deine Variablendeklarationen oben stehen und im Konstruktor nur eine Zuweisung ohne Deklaration ist.

Die letzte Zeile des Konstruktors legt das erste der beiden erzeugten Bilder als das aktuelle Bild der Krabbe fest:

```
setImage(image1);
```

Wir können später einen ähnlichen Methodenaufruf verwenden, um zwischen den Bildern in der **act**-Methode zu wechseln.

Übung 4.20 Füge diesen Konstruktor in deine **Crab**-Klasse ein. Du wirst keine Verhaltensänderungen bei der Krabbe beobachten können, aber die Klasse sollte sich übersetzen lassen und du solltest in der Lage sein, Krabben zu erzeugen.

Übung 4.21 Inspiziere noch einmal dein Krabben-Objekt. Beachte auch hier wieder die Variablen und ihre Werte. Vergleiche sie mit denen, die du zuvor notiert hast.

4.13 Die Bilder wechseln

Damit sind wir jetzt so weit, dass uns zwei Bilder der Krabbe für die Animation zur Verfügung stehen. Doch die Animation selbst haben wir noch nicht in Angriff genommen. Dies ist jetzt allerdings relativ einfach.

Für die Animation müssen wir lediglich zwischen unseren beiden Bildern hin- und herwechseln. Mit anderen Worten: Bei jedem Schritt wollen wir, wenn wir gerade **image1** zeigen, zu **image2** wechseln und umgekehrt. In Pseudocode sähe das folgendermaßen aus:

```
if (wenn unser aktuelles Bild image1 ist) then  
    verwende jetzt image2  
else  
    verwende jetzt image1
```

Pseudocode, wie er hier verwendet wird, ist eine Technik, eine Aufgabe so auszudrücken, dass die Struktur zum Teil echter Java-Code ist und zum Teil einfaches Deutsch. Oft ist Pseudocode eine gute Hilfe, um herauszufinden, wie der reale Code aussehen sollte. Wir können unser Problem jetzt in richtigem Java-Code ausdrücken (Listing 4.5).

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

Listing 4.5
Zwischen zwei
Bildern wechseln.

In diesem Codefragment fallen uns einige neue Elemente auf:

- Die Methode **getImage** kann dazu verwendet werden, um das aktuelle Bild des Akteurs zu erhalten.
- Der Operator **==** (zwei Gleichheitszeichen) kann dazu verwendet werden, um einen Wert mit einem anderen zu vergleichen. Das Ergebnis ist entweder **true** oder **false**.
- Die **if**-Anweisung hat eine Form, der wir bisher noch nicht begegnet sind. Bei dieser Form folgen auf den ersten Rumpf der **if**-Anweisung das Schlüsselwort **else** und ein weiterer Anweisungsblock. Im nächsten Abschnitt wollen wir diese neue Form der **if**-Anweisung genauer unter die Lupe nehmen.

Konzept

Mithilfe des doppelten Gleichheitszeichens (==) können wir prüfen, ob zwei Dinge **gleich** sind.

Fallstricke

Ein immer wiederkehrender Fehler ist es, den Anweisungsoperator (=) mit dem Operator, der auf Gleichheit prüft (==), zu verwechseln. Wenn du prüfen möchtest, ob zwei Werte oder Variablen gleich sind, musst du zwei Gleichheitszeichen schreiben.

4.14 Die if/else-Anweisung

Bevor wir fortfahren, wollen wir die **if**-Anweisung einmal näher betrachten. Wie wir gerade gesehen haben, kann eine **if**-Anweisung auch in der folgenden Form geschrieben werden:

```

if ( bedingung )
{
    anweisungen;
}
else
{
    anweisungen;
}
    
```

Konzept

Die **if/else**-Anweisung führt ein Codefragment aus, wenn eine gegebene Bedingung wahr ist, und ein anderes Codefragment, wenn die Bedingung falsch ist.

Diese **if**-Anweisung enthält zwei Blöcke (d.h. zwei Paare geschweifeter Klammern um jeweils eine Liste von Anweisungen): die *if-Klausel* und die *else-Klausel* (in dieser Reihenfolge).

Wenn diese **if**-Anweisung ausgeführt wird, wird zuerst die Bedingung ausgewertet. Ist die Bedingung wahr, wird die **if**-Klausel ausgeführt und die Ausführung setzt mit dem Code unterhalb der **else**-Klausel fort. Ist die Bedingung falsch, wird statt der **if**-Klausel die **else**-Klausel ausgeführt. Auf diese Weise wird immer einer der beiden Anweisungsblöcke ausgeführt, aber nie beide.

Der **else**-Teil mit dem zweiten Block ist optional. Wenn wir ihn weglassen, erhalten wir die kürzere Version der **if**-Anweisung, die wir bereits von früher kennen.

Damit verfügen wir über alle nötigen Informationen, um unsere Aufgabe zu meistern. Jetzt ist es Zeit, wieder zur Tastatur zurückzukehren und das Gelernte in die Praxis umzusetzen.

Übung 4.22 Füge den Code aus Listing 4.5 zum Alternieren der Bilder in die `act`-Methode deiner `Crab`-Klasse ein. Versuche es einmal selbst. (Wenn du einen Fehler einbaust, behebe ihn. Das Beispiel sollte funktionieren.) Klicke in Greenfoot auch ab und zu auf den ACT-Button anstatt auf den RUN-Button – damit kannst du das Verhalten etwas besser beobachten.

Übung 4.23 In *Kapitel 3* haben wir darüber gesprochen, dass man für Unteraufgaben eigene Methoden definieren sollte, anstatt immer mehr Code in die `act`-Methode zu packen. Übertrage dieses Prinzip auf den Code zum Alternieren des Bildes: Erzeuge eine neue Methode namens `switchImage`, verschiebe in diese deinen Code und rufe die Methode von der `act`-Methode aus auf.

Übung 4.24 Rufe die Methode `switchImage` interaktiv von dem Kontextmenü der Krabbe aus auf. Funktioniert das?

4.15 Würmer zählen

Kommen wir jetzt zu unserem letzten Thema im Zusammenhang mit den Krabben: dem Zählen. Wir wollen Funktionalität hinzufügen, die die Krabbe mitzählen lässt, wie viele Würmer sie gefressen hat. Wenn sie acht Würmer gefressen hat, haben wir das Spiel gewonnen. Im letzteren Fall möchten wir dann auch noch einen kurzen „Gewonnen-Sound“ abspielen.

Um unsere Pläne zu realisieren, werden wir einige Ergänzungen an unserem Krabben-Code vornehmen. So benötigen wir

- eine Instanzvariable, um die Zahl der aktuell gefressenen Würmer zu speichern,
- eine Zuweisung, die diese Variable am Anfang mit 0 initialisiert,
- Code, um unsere Zählung jedes Mal, wenn wir einen Wurm gefressen haben, um eins zu erhöhen (inkrementieren),
- und Code, der prüft, ob wir acht Würmer gefressen haben, und, wenn ja, das Spiel anhält und einen Sound abspielt.

Lass uns die Aufgaben in der Reihenfolge erledigen, in der wir sie aufgeführt haben.

In Anlehnung an das obige Beispiel definieren wir eine neue Instanzvariable. Unter unseren bereits vorhandenen Instanzvariablen fügen wir die folgende Zeile ein:

```
private int wormsEaten;
```

Hier weist der Typ `int` darauf hin, dass wir Integer (ganze Zahlen) speichern wollen, und der Name `wormsEaten` verrät uns, welchen Zweck diese Variable erfüllt.

Als Nächstes fügen wir folgende Zeile am Ende unseres Konstruktors ein:

```
wormsEaten = 0;
```

Damit initialisieren wir die Variable `wormsEaten` bei der Erzeugung der Krabbe mit 0. Eigentlich ist diese Zeile überflüssig, da Instanzvariablen vom Typ `int` automatisch mit 0 initialisiert werden. Doch manchmal benötigen wir einen Anfangswert ungleich 0, sodass das Schreiben einer eigenen Initialisierungsanweisung ruhig zur Gewohnheit werden sollte.

Was uns noch bleibt, ist, die Würmer zu zählen und zu prüfen, ob wir acht erreicht haben. Dies muss jedes Mal erfolgen, wenn wir einen Wurm fressen. Deshalb suchen wir nach unserer Methode **lookForWorm**, die den Code zum Fressen der Würmer enthält. Hier fügen wir eine Codezeile hinzu, die die Wurmmenge inkrementiert:

```
wormsEaten = wormsEaten + 1;
```

Wie immer bei einer Zuweisung wird zuerst die Seite rechts des Zuweisungssymbols ausgewertet (**wormsEaten + 1**). Wir lesen also den aktuellen Wert von **wormsEaten** und addieren dazu eine 1. Anschließend weisen wir diesen Wert wieder der Variablen **wormsEaten** zu. Als Folge wird die Variable um 1 inkrementiert.

Jetzt benötigen wir noch eine **if**-Anweisung, die prüft, ob wir schon acht Würmer gefressen haben, und in diesem Fall den Sound abspielt und die Ausführung anhält. Listing 4.6 enthält den vollständigen Code der Methode **lookForWorms**. Die hier verwendete Sounddatei (*fanfare.wav*) liegt bereits im *sounds*-Ordner deines Szenarios, sodass sie einfach abgespielt werden kann.

Listing 4.6:

Würmer zählen und prüfen, ob wir gewonnen haben.

```
/**
 * Prüft, ob wir auf einen Wurm gestoßen sind.
 * Wenn ja, wird er gefressen. Wenn nein, passiert nichts.
 * Wenn wir 8 Würmer gefressen haben, haben wir gewonnen.
 */
public void lookForWorm()
{
    if ( isTouching(Worm.class) )
    {
        removeTouching(Worm.class);
        Greenfoot.playSound("slurp.wav");

        wormsEaten = wormsEaten + 1;
        if (wormsEaten == 8)
        {
            Greenfoot.playSound("fanfare.wav");
            Greenfoot.stop();
        }
    }
}
```


Übung 4.25 Füge den zuvor besprochenen Code in dein Szenario ein, teste ihn und stelle sicher, dass er funktioniert.

Übung 4.26 Als zusätzliche Kontrolle kannst du den Objektinspektor für dein Krabben-Objekt öffnen (wähle dazu einfach *INSPIZIEREN* aus dem Kontextmenü der Krabbe), bevor du mit dem Spielen anfängst. Lass den Inspektor geöffnet, während du spielst, und beobachte die Variable **wormsEaten**.

4.16 Weitere Ideen

Das Szenario *little-crab-5* im Szenarien-Ordner des Buchkapitels ist eine Version des Projekts, das alle hier besprochenen Erweiterungen beinhaltet.

Wir werden uns jetzt von diesem Szenario verabschieden und einem anderen Beispiel zuwenden, auch wenn es noch viele naheliegende (oder weniger naheliegende) Möglichkeiten für Verbesserungen gibt. So könntest du zum Beispiel

- verschiedene Bilder für den Hintergrund und die Akteure verwenden;
- weitere Arten von Akteuren einführen;
- dich nicht automatisch vorwärtsbewegen, sondern nur, wenn die Taste  gedrückt wird;
- ein Spiel entwickeln, das von zwei Spielern gespielt wird, indem du eine zweite tastaturgesteuerte Klasse einführst, die auf andere Tasten reagiert;
- festlegen, dass für jeden gefressenen Wurm (oder zu beliebigen Zeiten) neue Würmer auftauchen
- und dir noch vieles mehr überlegen.

Übung 4.27 Das Bild der Krabbe ändert sich beim Laufen sehr schnell, was die Krabbe ein wenig hyperaktiv wirken lässt. Vielleicht wäre es netter, wenn sich das Bild der Krabbe nur bei jedem zweiten oder dritten **act**-Schritt ändern würde. Versuche einmal, dies zu implementieren. Dazu könntest du einen Zähler hinzufügen, der in der **act**-Methode inkrementiert wird. Jedes Mal, wenn dieser Zähler 2 (oder 3) erreicht, ändert sich das Bild und der Zähler wird wieder auf 0 zurückgesetzt.

Zusammenfassung der Programmiertechniken

In diesem Kapitel haben wir uns mit einer Reihe von wichtigen neuen Programmierkonzepten vertraut gemacht. Wir haben gesehen, wie Konstruktor dazu genutzt werden können, um Objekte zu initialisieren – Konstruktor werden immer ausgeführt, wenn ein neues Objekt erzeugt wird.

Wir haben Instanzvariablen und lokale Variablen kennengelernt. Instanzvariablen (auch *Zustandsfelder* genannt) werden – zusammen mit Zuweisungsanweisungen – verwendet, um Informationen in Objekten zu speichern, auf die man später zugreifen kann. Lokale Variablen werden eingesetzt, um Informationen für eine kurze Zeit zu speichern – innerhalb einer einzelnen Methodenausführung –, sie werden verworfen, wenn die Methode endet.

Wir haben die Anweisung **new** verwendet, um neue Objekte im Quelltext zu erzeugen, und zum Schluss haben wir eine vollständige Version der **if**-Anweisung kennengelernt, die einen **else**-Teil umfasst, der ausgeführt wird, wenn die Bedingung nicht wahr ist.

Mithilfe all dieser Techniken können wir bereits eine ganze Menge Code schreiben.





Zusammenfassung der Konzepte

- Der **Konstruktor** einer Klasse ist eine besondere Art Methode, die immer automatisch ausgeführt wird, wenn ein Objekt dieser Klasse erzeugt wird.
- Java-Objekte können im Programm Quelltext mithilfe des Schlüsselwortes **new** erzeugt werden.
- **Variablen** dienen zum Speichern von Informationen (Objekte oder Werte), die später benötigt werden.
- Variablen können durch das Schreiben einer **Variablendeklaration** erzeugt werden.
- Wir können Werte in Variablen mithilfe einer **Zuweisungsanweisung** (=) speichern.
- Variablen von **primitiven Typen** speichern Zahlen, boolesche Werte und Zeichen; Variablen von **Objekttypen** speichern Objekte.
- Objekte werden in Variablen gespeichert, indem eine **Referenz** auf das Objekt gespeichert wird.
- Greenfoot-Akteure verwalten ihr sichtbares Bild in Form eines Objekts vom Typ **GreenfootImage**.
- **Instanzvariablen** (auch **Zustandsfelder** genannt) sind Variablen, die zu einem Objekt gehören (und nicht zu einer Methode).
- **Lebenszeit von Instanzvariablen:** Instanzvariablen bestehen so lange, wie das Objekt existiert, zu dem sie gehören.
- **Lebenszeit von lokalen Variablen:** Lokale Variablen bestehen nur während einer einzigen Methodenausführung.
- Mithilfe des doppelten Gleichheitszeichens (==) können wir prüfen, ob zwei Dinge **gleich** sind.
- Die **if/else**-Anweisung führt ein Codefragment aus, wenn eine gegebene Bedingung wahr ist, und ein anderes Codefragment, wenn die Bedingung falsch ist.

Vertiefende Aufgaben

Dieses Mal machen wir mehr Übungen mit Methodenaufrufen, einschließlich einer neu geerbten **Actor**-Methode, und üben den Umgang mit sowie den Einsatz von Variablen.

Noch einmal Krabben

Übung 4.28 Die Hummer sollen ein bisschen gefährlicher werden. Die **Actor**-Klasse besitzt eine Methode namens **turnTowards**. Verwende diese Methode, damit sich die Hummer ab und an der Mitte des Bildschirms zuwenden. Experimentiere sowohl mit der Häufigkeit dieser Richtungswechsel als auch mit unterschiedlichen Laufgeschwindigkeiten für die Hummer und die Krabbe.

Übung 4.29 Statte die Krabbe mit einem Zeitzähler aus. Dazu kannst du eine **int**-Variable hinzufügen, die jedes Mal, wenn sich die Krabbe bewegt, hochgezählt wird. (Tatsächlich zählst du dann die **act**-Schritte.) Sollte dies eine lokale Variable oder eine Instanzvariable sein? Warum?

Übung 4.30 Führe dein Spiel aus. Sobald du einen Sieg geschafft hast (also acht Würmer zu fressen), untersuche dein Krabben-Objekt und prüfe, wie lange du gebraucht hast. Wie viele **act**-Schritte sind verbraucht?

Übung 4.31 Verschiebe deinen Zeitzähler aus der **Crab**-Klasse in die Klasse **CrabWorld**. (Es ist sinnvoller, dass die Welt anstatt eine einzelne Krabbe die Zeit verwaltet.) Die Variable ist einfach zu verschieben. Für das Verschieben der Anweisung, die die Zeit hochzählt, musst du eine **act**-Methode in der **CrabWorld**-Klasse definieren. **World**-Unterklassen können ebenso wie **Actor**-Unterklassen **act**-Methoden besitzen. Um eine neue **act**-Methode in **CrabWorld** zu erzeugen, kopiere einfach die Signatur der **act**-Methode der Krabbe und platziere hier deine Anweisung für den Zeitzähler.

Übung 4.32 Mache aus dem Zeitzähler deines Spiels einen Spielzähler. Das heißt: Initialisiere die Zeitvariable aus irgendeinen Wert (z.B. 500) und zähle bei jedem **act**-Schritt rückwärts (ziehe 1 vom Wert der Variable ab). Wenn der Zähler 0 erreicht, dann beende das Spiel mit einem Sound für „Zeit ist abgelaufen“. Experimentiere mit unterschiedlichen Werten für die Spielzeit.

Übung 4.33 Untersuche die Methode **showText** in der **World**-Klasse. Wie viele Parameter hat sie? Welche sind es? Was gibt sie zurück? Was macht sie?

Übung 4.34 Zeige den Spielzähler auf dem Bildschirm an, indem du die **showText**-Methode verwendest. Du kannst dies in der **act**-Methode von **CrabWorld** machen. Dazu benötigst du eine Anweisung wie diese:

```
showText("Time left: "+ time, 100, 40);
```

wobei **time** der Name deiner Zählervariable ist. (Beachte: diese Anweisung verwendet den Plus-Operator und einen Text-String, dies werden wir in **Kapitel 5** behandeln.)

Übungen mit springendem Ball

Übung 4.35 Erzeuge ein neues Szenario und in diesem eine Welt- und eine Akteur-Klasse mit Namen *Ball*. (Weise ihm ein Bild zu, das wie ein Ball aussieht.) Programmiere den Ball so, dass er sich mit konstanter Geschwindigkeit bewegt, und lasse ihn von den Rändern der Welt abprallen.

Übung 4.36 Programmiere deinen Ball so, dass er zählt, wie häufig er vom Rand abgeprallt ist. Führe dein Szenario aus, wenn der Objekt-Inspektor des Balls zum Testen geöffnet ist.

Übung 4.37 Programmiere dein Szenario so, dass am Start automatisch drei Bälle vorhanden sind.

Übung 4.38 Verändere deinen Initialisierungscode, sodass die drei Bälle an zufälligen Orten erscheinen.

Übung 4.39 Verändere den Code für das Abprallen vom Rand, sodass die Bälle in zufälligen Winkeln vom Rand abprallen.