

KAPITEL

3

Das Spiel „Little Crab“ ausbauen – fortgeschrittenere Programmieretechniken



Lernziele

Themen: Zufälliges Verhalten, Tastatursteuerung, Sound

Konzepte: Punktnotation, Zufallszahlen, Definition von Methoden, Kommentare

Im vorherigen Kapitel haben wir uns mit den Grundlagen der Programmierung beschäftigt und unser erstes eigenes Spiel programmiert. Dabei ist viel Neues auf uns eingeströmt. Jetzt werden wir weitere interessante Verhaltensweisen hinzufügen. Das Hinzufügen von Code wird uns von nun an, da wir viele grundlegende Konzepte bereits kennengelernt haben, allerdings etwas leichter fallen.

Als Erstes wollen wir zufälliges Verhalten programmieren.

3.1 Zufälliges Verhalten einbringen

In unserer bisherigen Implementierung kann die Krabbe über den Bildschirm laufen und am Rand der Welt wenden. Wenn die Krabbe jedoch läuft, läuft sie immer nur geradeaus. Dieses Verhalten wollen wir jetzt ändern. Krabben laufen nicht immer geradeaus, sodass wir eine Zufallskomponente einbauen wollen: Die Krabbe soll größtenteils geradeaus laufen und gelegentlich ein wenig vom Kurs abweichen.

In Greenfoot erreichen wir solche Verhaltensweisen mithilfe von Zufallszahlen: Die Greenfoot-Umgebung selbst stellt uns dazu eine Methode zur Verfügung, die eine Zufallszahl zurückgibt. Diese Methode namens **getRandomNumber** erwartet einen Parameter, der den Grenzwert für die zurückzuliefernde Zahl angibt. Als Ergebnis liefert die Methode dann eine Zufallszahl zwischen 0 (null) und diesem Grenzwert zurück. Zum Beispiel:

```
Greenfoot.getRandomNumber(20)
```

Diese Methode liefert eine Zufallszahl zwischen 0 und 20. Der Grenzwert 20 ist davon jedoch ausgenommen, sodass die Zahl tatsächlich im Bereich 0–19 liegt.

Konzept

Wenn wir eine Methode aufrufen wollen, die weder in unserer Klasse definiert ist noch geerbt wurde, müssen wir die Klasse oder das Objekt, in der sich die Methode befindet, getrennt durch einen Punkt vor dem Methodennamen angeben. Diese Schreibweise wird auch **Punktnotation** genannt.

Konzept

Methoden, die zu Klassen (und nicht zu Objekten) gehören, werden in ihrer Signatur durch das Schlüsselwort **static** gekennzeichnet. Sie werden auch als **Klassenmethoden** bezeichnet.

Die hier verwendete Notation wird auch *Punktnotation* genannt. Wenn wir Methoden aufrufen, die in unserer eigenen Klasse definiert sind oder geerbt wurden, reicht es, einfach den Methodennamen und die Parameterliste zu schreiben. Ist die Methode jedoch in einer anderen Klasse definiert, müssen wir die Klasse oder das Objekt angeben, in der sich die Methode befindet, gefolgt von einem Punkt, gefolgt von dem Methodennamen und eventueller Parameter. Da sich die Methode **getRandomNumber** weder in der Klasse **Crab** noch in der Klasse **Actor** befindet, sondern in einer Klasse namens **Greenfoot**, müssen wir den Methodenaufruf mit „**Greenfoot.**“ beginnen.

Hinweis: Statische Methoden

Methoden können zu Objekten oder Klassen gehören. Wenn Methoden zu Klassen gehören, schreiben wir:

```
Klassenname.methodenname (parameter);
```

um die Methode aufzurufen. Wenn eine Methode zu einem Objekt gehört, schreiben wir:

```
objekt.methodenname (parameter);
```

um die Methode aufzurufen.

Beide Arten von Methoden sind in einer Klasse definiert. Die Methodensignatur verrät uns, ob eine gegebene Methode zu den Objekten dieser Klasse oder der Klasse selbst gehört.

Methoden, die zu der Klasse selbst gehören, werden am Anfang der Methodensignatur durch das Schlüsselwort **static** gekennzeichnet. So lautet zum Beispiel die Signatur der **Greenfoot**-Methode **getRandomNumber**:

```
static int getRandomNumber(int limit)
```

Daran können wir ablesen, dass wir den Namen der Klasse (**Greenfoot**) vor dem Punkt in dem Methodenaufruf schreiben müssen.

In einem späteren Kapitel werden wir es noch mit Methoden zu tun bekommen, die anderen Objekten gehören.

Angenommen, wir wollen unsere Krabbe so programmieren, dass bei jedem Schritt eine zehnprozentige Chance besteht, dass die Krabbe ein wenig vom Kurs abweicht. Den größten Teil dieser Aufgabe können wir mit einer **if**-Anweisung erledigen:

```
if ( etwas-ist-wahr )  
{  
    turn(5);  
}
```

Jetzt müssen wir nur noch einen Ausdruck finden, den wir anstelle von **etwas-ist-wahr** einsetzen können und der in genau zehn Prozent aller Fälle **true** zurückliefert.

Hierzu können wir eine Zufallszahl (unter Verwendung der Methode **Greenfoot.getRandomNumber**) und den Kleiner-als-Operator verwenden. Der Kleiner-als-Operator vergleicht zwei Zahlen und liefert **true** zurück, wenn die erste kleiner ist als die zweite. „Kleiner als“ wird durch das Symbol „<“ ausgedrückt. Zum Beispiel ist

2 < 33

wahr (**true**), während

162 < 42

falsch (**false**) ist.

Übung 3.1 Bevor du weiterliest, versuche einmal auf Papier einen Ausdruck aufzuschreiben, der die Methode **getRandomNumber** und den Kleiner-als-Operator verwendet und bei dem die Ausführung in genau zehn Prozent aller Fälle wahr ist.

Übung 3.2 Schreibe einen weiteren Ausdruck nieder, der in genau sieben Prozent aller Fälle wahr ist.

Hinweis

Java verfügt über eine ganze Reihe von Operatoren, mit denen sich zwei Werte vergleichen lassen. Diese lauten:

- < Kleiner als
- > Größer als
- <= Kleiner gleich
- >= Größer gleich
- = Gleich
- != Nicht gleich

Wenn wir eine Chance in Prozent ausdrücken wollen, bedienen wir uns hierfür am besten der Zufallszahlen bis 100. Ein Ausdruck, der in zehn Prozent aller Fälle wahr ist, könnte zum Beispiel folgendermaßen lauten

Greenfoot.getRandomNumber(100) < 10

Da der Aufruf von **Greenfoot.getRandomNumber(100)** uns jedes Mal eine neue Zufallszahl zwischen 0 und 99 liefert und diese Zahlen gleichmäßig verteilt sind, sind sie in zehn Prozent aller Fälle unter 10.

Dies können wir uns jetzt zunutze machen, um unsere Krabbe in zehn Prozent aller Schritte eine kleine Drehung vollziehen zu lassen (Listing 3.1).

Listing 3.1:

Zufällige Kursänderungen – erster Versuch.

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

/**
 * Diese Klasse definiert eine Krabbe. Krabben leben am Strand.
 */
public class Crab extends Actor
{
    public void act()
    {
        if ( isAtEdge() )
        {
            turn(17);
        }

        if ( Greenfoot.getRandomNumber(100) < 10 )
        {
            turn(4);
        }
        move(5);
    }
}
```

Übung 3.3 Implementiere diese zufälligen Kursänderungen in deiner eigenen Version. Experimentiere mit verschiedenen Wahrscheinlichkeiten für die Krabbe, einen anderen Kurs einzuschlagen.

Der Ansatz war schon recht vielversprechend, aber es gibt noch einiges zu verbessern. Zum einen dreht sich die Krabbe, wenn sie sich dreht, immer um den gleichen Winkel (4 Grad), und zweitens dreht sie sich immer nach rechts und nie nach links. Was wir jedoch gerne hätten, ist eine Krabbe, die sich um einen kleinen, aber zufälligen Winkel nach links oder nach rechts dreht. (Wie dies zu lösen ist, werden wir gleich besprechen. Wenn du es dir schon zutraust, kannst du ja versuchen, eine eigene Version zu implementieren, bevor du weiterliest.)

Der einfache Trick zur Behebung des ersten Problems – der immer gleiche Drehwinkel (in unserem Fall 4 Grad) – besteht darin, die fest vorgegebene Zahl 4 in unserem Code ebenfalls durch eine Zufallszahl zu ersetzen:

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

In diesem Beispiel dreht sich die Krabbe immer noch in zehn Prozent aller ihrer Schritte und wenn sie sich dreht, dreht sie sich um einen zufälligen Winkel, der zwischen 0 und 45 Grad liegt.

Übung 3.4 Gib den oben stehenden Code in dein Beispiel ein. Was kannst du beobachten? Dreht sich die Krabbe um verschiedene Winkel, wenn sie sich dreht?

Übung 3.5 Bleibt noch das Problem, dass sich die Krabbe immer nur nach rechts dreht. Dies entspricht nicht dem normalen Verhalten einer Krabbe, sodass hier noch Handlungsbedarf besteht. Ändere deinen Code so, dass sich deine Krabbe bei jeder Drehung entweder nach rechts oder nach links um bis zu 45 Grad dreht.

Übung 3.6 Führe verschiedene Szenarien mit mehreren Krabben in der Welt aus. Drehen sie sich alle zur selben Zeit oder unabhängig voneinander? Warum?

Übung 3.5 ist anfangs nicht einfach, wenn du so etwas bisher noch nie gemacht hast, daher wollen wir dazu noch ein paar Hinweise geben. Am Anfang ist die Versuchung groß, eine **if**-Anweisung zu benutzen, um beide Richtungen abzudecken, doch es gibt tatsächlich eine viel einfachere Lösung.

Denk einmal in folgende Richtung: **Greenfoot.getRandomNumber** gibt uns nur Zahlen mit null als unterer Grenze. Wir benötigen aber Zahlen von -45 bis 45 . Das heißt, wir möchten (rund) 90 unterschiedliche Zahlen haben. Wir können also einen Bereich von 90 Zahlen erhalten, indem wir **Greenfoot.getRandomNumber(90)** verwenden, doch damit bekommen wir die Zahlen $0-90$. Wie gelangen wir von $0-90$ zu $-45-45$?

Die Antwort lautet: Wir können einfach eine Zufallszahl aus dem Bereich bis 90 nehmen und dann 45 subtrahieren. Abbildung 3.1 illustriert dies. (Beachte, dass man sich damit eigentlich um 1 verrechnet: der Bereich der zufälligen Zahlen liegt tatsächlich bei $0-89$, nach der Subtraktion von 45 sind wir letztendlich bei $-45-44$. Doch wir kümmern uns bei mit unseren zufälligen Drehungen nicht so sehr um Exaktheit, daher ist das in Ordnung.)

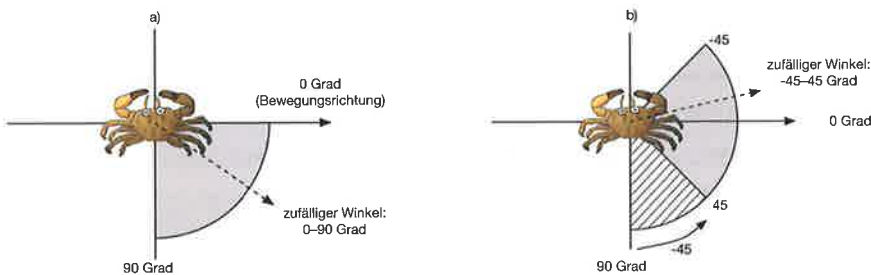


Abbildung 3.1

a) Zufällige Winkelgröße von $0-90$ Grad.
b) Zufällige Winkelgröße von $-45-45$ Grad.

Versuche, diese Übung allein zu lösen. Das Projekt *little-crab-2* (das Teil der Buchszenarien ist) beinhaltet eine Implementierung des bisher Besprochenen, einschließlich der letzten Übungen.

3.2 Würmer hinzufügen

Um unsere Welt ein wenig interessanter zu gestalten, wollen wir sie um eine weitere Tierart bereichern.

Krabben lieben Würmer. (Das mag zwar nicht für alle Krabben in der realen Welt zutreffen, doch bei manchen ist das der Fall. Deshalb wollen wir davon ausgehen, dass unsere Krabbe eine von denen ist, die Würmer mögen.) Aus diesem Grund wollen wir jetzt eine Klasse für Würmer implementieren.

Wir können einem Greenfoot-Szenario eine neue Akteur-Klasse hinzufügen, indem wir aus dem Kontextmenü einer der bestehenden Akteur-Klassen den Befehl **NEUE UNTERKLASSE** wählen (Abbildung 3.2). In diesem Fall ist unsere neue Klasse **Worm** eine spezielle Art von Tier, sodass sie eine Unterklasse von **Actor** sein sollte. (Denke daran, dass eine Unterklasse in einer *Ist-eine*-Beziehung zu ihrer Oberklasse steht: Ein Wurm *ist ein* Akteur.)

Wenn wir eine neue Unterklasse erzeugen, werden wir aufgefordert, einen Namen für die Klasse einzugeben und ein Bild auszuwählen (Abbildung 3.3).

In unserem Fall nennen wir die Klasse „Worm“. Per Konvention beginnen Klassennamen in Java immer mit einem Großbuchstaben. Außerdem sollte der Name beschreiben, welche Art von Objekt er repräsentiert, sodass „Worm“ der naheliegende Name für unseren Zweck ist.

Dann sollten wir der Klasse ein Bild zuweisen. Es gibt bereits einige Bilder, die mit dem Szenario verbunden sind, sowie eine ganze Bibliothek von zur Auswahl stehenden generischen Bildern. Hier haben wir bereits das Bild eines Wurmes vorgegeben und in **SCENARIOBILDER** verfügbar gemacht, sodass wir einfach nur das Bild namens *worm.png* auswählen müssen.

Abbildung 3.2
Neue Unterklassen erzeugen.

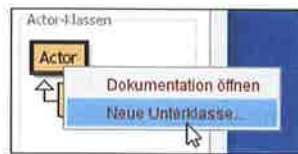
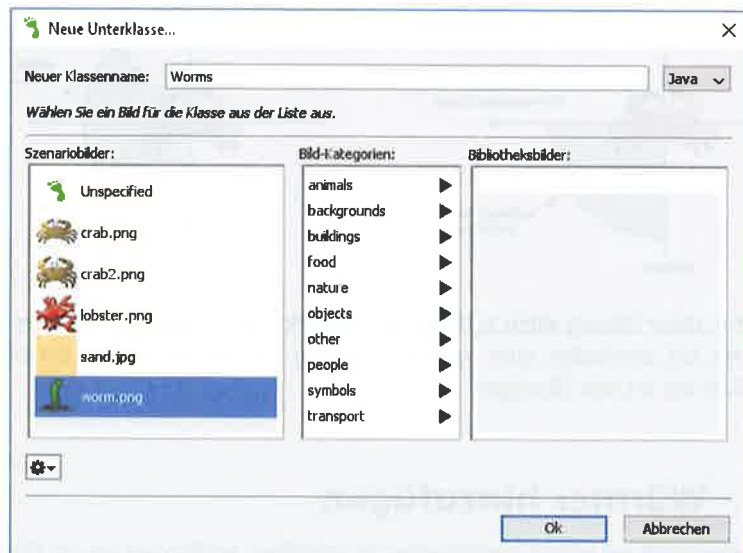


Abbildung 3.3
Eine neue Klasse erzeugen.



Haben wir das erledigt, können wir auf OK klicken. Damit wird die Klasse unserem Szenario hinzugefügt und wir können unseren Quelltext neu übersetzen und Würmer in unsere Welt aufnehmen.

Übung 3.7 Füge einige Würmer in deine Welt ein. Füge außerdem einige Krabben ein. Führe das Szenario aus. Was kannst du beobachten? Was machen die Würmer? Was passiert, wenn eine Krabbe auf einen Wurm trifft?

Inzwischen wissen wir, wie wir neue Klassen in unser Szenario aufnehmen. Die nächste Aufgabe besteht darin, dafür zu sorgen, dass diese Klassen interagieren.

3.3 Würmer fressen

Jetzt wollen wir für die Krabbe ein neues Verhalten definieren: Wenn eine Krabbe auf einen Wurm trifft, frisst sie ihn. Auch hier prüfen wir zuerst, welche Methoden wir bereits von der Klasse **Actor** geerbt haben. Dazu öffnen wir erneut die Dokumentation für die Klasse **Actor** und stellen fest, dass es die beiden folgenden Methoden gibt:

boolean isTouching (java.lang.Class class)

Prüft, ob dieser Akteur irgendein anderes Objekt der gegebenen Klasse berührt.

void removeTouching (java.lang.Class class)


Entfernt ein Objekt der gegebenen Klasse, das dieser Akteur gerade berührt (falls es eins gibt).

Mithilfe dieser Methoden können wir dieses Verhalten implementieren. Die erste Methode prüft, ob die Krabbe einen Wurm berührt. Diese Methode liefert einen booleschen Wert – **true** oder **false** – zurück, den wir dann in einer **if**-Anweisung verwenden können.

Die zweite Methode entfernt einen Wurm. Beide Methoden erwarten einen Parameter vom Typ **java.lang.Class**. Das bedeutet, dass wir eine unserer Klassen aus unserem Szenario angeben müssen. Ein Codebeispiel sähe folgendermaßen aus:

```
if ( isTouching(Worm.class) )
{
    removeTouching(Worm.class);
}
```

In diesem Fall geben wir bei beiden Methodenaufrufen (der Methode **isTouching** und der Methode **removeTouching**) als Parameter **Worm.class** an. Damit deklarieren wir, welche Art von Objekt wir suchen und welche Art von Objekt wir entfernen wollen. Unsere vollständige **act**-Methode zu diesem Zeitpunkt findest du in Listing 3.2.

Dieses Beispiel solltest du nachvollziehen. Platziere eine Reihe von Würmern in der Welt (zur Erinnerung: Wenn du bei gedrückt gehaltener -Taste in die Welt klickst, kannst du schnell mehrere Akteure nacheinander platzieren), füge einige Krabben hinzu, führe das Szenario aus und schau, was passiert.

Listing 3.2:

Erste Version eines Programms, in dem die Krabben Würmer fressen.

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

    if ( Greenfoot.getRandomNumber(100) < 10 )
    {
        turn( Greenfoot.getRandomNumber(90)-45 );
    }
    move(5);

    if ( isTouching(Worm.class) )
    {
        removeTouching(Worm.class);
    }
}
```

Hinweis für Fortgeschrittene: Pakete

(Die „Hinweise für Fortgeschrittene“ sind für die besonders interessierten Leser unter euch gedacht. Sie müssen zu diesem Zeitpunkt eigentlich noch nicht verstanden werden und können ruhig übersprungen werden.)

In der Definition der Methoden **isToching** und **removeTouching** ist uns ein Parametertyp mit dem Namen **java.lang.Class** begegnet. Was hat es damit auf sich?

Viele Typen werden durch Klassen definiert. Viele dieser Klassen sind Teil der Java-Klassenbibliotheken. Du kannst die Dokumentation der Java-Klassenbibliotheken einsehen, wenn du im HILFE-Menü von Greenfoot den Befehl JAVA KLASSENBIBLIOTHEKEN wählst.

Die Java-Klassenbibliotheken enthalten Tausende von Klassen. Um die Arbeit mit diesen Klassen zu erleichtern, wurden sie in *Pakete* (logisch verwandte Gruppen von Klassen) zusammengefasst. Wenn ein Klassenname Punkte beinhaltet, wie **java.lang.Class**, ist nur der letzte Teil der Name der Klasse und die Teile davor bilden den Namen des Pakets. Für unseren Fall hier würde das bedeuten, dass wir nach der Klasse „**Class**“ aus dem Paket „**java.lang**“ Ausschau halten.

Versuche einmal, diese Klasse in den Java-Klassenbibliotheken zu finden.

Konzept

Eine **Methodendefinition** definiert eine neue Aktion für Objekte dieser Klasse. Die Aktion wird nicht direkt ausgeführt, aber die Methode kann vermittels eines **Methodenaufrufs** später ausgeführt werden.

3.4 Neue Methoden erzeugen

In den vorherigen Abschnitten haben wir für die Krabbe ein neues Verhalten definiert – am Rand der Welt wenden, gelegentlich beliebig große Drehbewegungen machen und Würmer fressen. Wenn wir so weitermachen, wird die **act**-Methode länger und länger, bis sie kaum noch verständlich ist. Dies lässt sich beheben, indem wir die Methode in kleinere Einheiten zerlegen.

Wir können in der Klasse **Crab** für unsere Zwecke auch eigene Methoden erzeugen. Anstatt zum Beispiel den Code zum Suchen und Entfernen eines Wurms in die **act**-Methode zu schreiben, können wir hierfür eine neue Methode einfügen.

Dazu müssen wir zuerst einen Namen für diese Methode festlegen; nennen wir sie **LookForWorm**. Anschließend können wir eine neue Methode erzeugen, indem wir den folgenden Code hinzufügen:

```
/**
 * Prüft, ob wir auf einen Wurm gestoßen sind.
 * Wenn ja, wird er gefressen. Wenn nein, passiert nichts.
 */
public void LookForWorm()
{
    if ( isTouching(Worm.class) )
    {
        removeTouching(Worm.class);
    }
}
```

Die ersten vier Zeilen bilden einen *Kommentar*. Ein Kommentar wird vom Computer ignoriert – er wird ausschließlich als Hilfe für Leute geschrieben, die unseren Code lesen. Hier verwenden wir den Kommentar, um den Lesern unseres Codes den Zweck dieser Methode zu erklären.

Die neu definierte Methode wird nicht sofort ausgeführt. Grundsätzlich führt die Definition einer Methode nämlich noch nicht zu ihrer Ausführung. Sie definiert lediglich eine neue potenzielle Aktion („Suche nach einem Wurm“), die später ausgeführt werden kann. Sie wird jedoch nur ausgeführt, wenn die Methode aufgerufen wird. Wir können in die **act**-Methode einen Aufruf unserer Methode einfügen:

```
LookForWorm();
```

Beachte, dass zu dem Aufruf auch die Klammern für die (leere) Parameterliste gehören. Den vollständigen Quelltext nach dieser Umstrukturierung zeigt Listing 3.3.

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

    if ( Greenfoot.getRandomNumber(100) < 10 )
    {
        turn( Greenfoot.getRandomNumber(90)-45 );
    }
    move(5);
    LookForWorm;
}
```

```
/**
 * Prüft, ob wir auf einen Wurm gestoßen sind.
 * Wenn ja, wird er gefressen. Wenn nein, passiert nichts.
 */
public void LookForWorm()
{
    if ( isTouching(Worm.class) )
    {
        removeTouching(Worm.class);
    }
}
```

Konzept

Kommentare sind an Programmierer gerichtete Erläuterungen, die in den Quelltext geschrieben werden; vom Computer werden sie ignoriert.

Listing 3.3

Quelltext in mehrere Methoden aufteilen.

Beachte, dass diese Änderungen am Code keinen Einfluss auf das Verhalten unserer Krabbe haben. Sie machen aber langfristig den Code besser lesbar. Je mehr Code wir der Klasse hinzufügen, umso länger werden die Methoden. Und lange Methoden sind nur sehr schwer zu verstehen. Indem wir unseren Code in mehrere kürzere Methoden zerlegen, erhöhen wir die Lesbarkeit unseres Codes.

Übung 3.8 Erzeuge eine weitere neue Methode namens **randomTurn** (diese Methode übernimmt keine Parameter und liefert auch nichts zurück). Markiere den Code für die zufälligen Drehbewegungen und verschiebe ihn aus der **act**-Methode in die neue Methode **randomTurn**. Rufe dann diese neue **randomTurn**-Methode aus deiner **act**-Methode heraus auf. Denke daran, einen Kommentar für diese Methode zu schreiben.

Erzeuge noch eine neue Methode namens **turnAtEdge** (sie übernimmt ebenfalls keine Parameter und liefert nichts zurück). Verschiebe den Code, der prüft, ob wir uns am Rand der Welt befinden (und wenn ja eine Drehung auslöst), in die Methode **turnAtEdge**. Rufe die Methode **turnAtEdge** aus deiner **act**-Methode heraus auf. Deine **act**-Methode sollte jetzt wie die Version in Listing 3.4 aussehen.

Listing 3.4:

Die neue **act**-Methode nach der Zerlegung in Methoden für die Teilaufgaben.

```
public void act()
{
    turnAtEdge();
    randomTurn();
    move(5);
    lookForWorm();
}
```

Methodennamen beginnen in Java standardmäßig mit einem Kleinbuchstaben. Sie dürfen keine Leerzeichen enthalten und viele andere Satzzeichen sind ebenfalls tabu. Besteht der Methodename logisch aus mehreren Wörtern, verwenden wir Großbuchstaben in der Mitte des Namens, um den Anfang eines jeden Wortes zu kennzeichnen.

3.5 Einen Hummer hinzufügen

Wir haben eine Stufe erreicht, auf der sich unsere Krabbe mehr oder weniger willkürlich durch die Welt bewegt und Würmer frisst, wenn sie zufällig auf welche trifft.

Um die Sache noch einmal etwas interessanter zu machen, wollen wir noch ein weiteres Tier aufnehmen: einen Hummer (Abbildung 3.4).

Abbildung 3.4

Einen Feind hinzunehmen: einen Hummer.



Die Hummer in unserem Szenario lieben es, Krabben zu jagen.

Übung 3.9 Füge deinem Szenario eine neue Klasse hinzu. Die Klasse sollte eine Unterklasse von **Actor** sein und den Namen **Lobster** (Hummer) mit einem großgeschriebenen **L** erhalten. Außerdem solltest du die Klasse mit dem vorgegebenen Bild *lobster.png* verbinden.

Übung 3.10 Was glaubst du, machen Hummer, wenn du sie so, wie sie sind, in der Welt platzierst? Kompiliere dein Szenario und probiere es aus.

Wir wollen jetzt unser Programm so ändern, dass die neuen Hummer die Krabben fressen. Das ist eigentlich ganz leicht, da sich ihr Verhalten kaum von dem Verhalten der Krabben unterscheidet. Der einzige Unterschied ist, dass Hummer nach Krabben Ausschau halten, während Krabben Würmer suchen.

Übung 3.11 Kopiere die komplette **act**-Methode aus der Klasse **Crab** in die Klasse **Lobster**. Kopiere ebenfalls die Methoden **lookForWorm**, **turnAtEdge** und **randomTurn**.

Übung 3.12 Ändere den Code von **Lobster** so, dass der Hummer nach Krabben und nicht nach Würmern Ausschau hält. Dazu musst du lediglich jedes Vorkommen von „Worm“ im Quelltext in „Crab“ ändern. Dort, wo zum Beispiel **worm.class** auftaucht, ändere den Code in **Crab.class**. Ändere auch den Namen der Methode **lookForWorm** in **lookForCrab**. Denke auch daran, deine Kommentare anzupassen.

Übung 3.13 Platziere eine Krabbe, drei Hummer und viele Würmer in der Welt. Führe das Szenario aus. Schafft es die Krabbe, alle Würmer zu fressen, bevor sie von einem Hummer erwischt wird?

Du solltest inzwischen eine Szenario-Version vorliegen haben, in der sich Krabben und Hummer nach dem Zufallsprinzip bewegen und nach Würmern bzw. Krabben suchen.

Dieses Programm wollen wir jetzt zu einem Spiel ausbauen.

3.6 Tastatursteuerung

Für ein spieleähnliches Verhalten müssen wir einen Spieler integrieren. Der Spieler (du!) sollte in der Lage sein, die Krabbe über die Tastatur zu steuern, während die Hummer weiterhin nach dem Zufallsprinzip durch die Welt laufen.

In der Greenfoot-Umgebung gibt es eine Methode, die es uns ermöglicht zu prüfen, ob eine Taste auf der Tastatur gedrückt wurde. Sie heißt **isKeyDown** und ist wie die Methode **getRandomNumber** aus Abschnitt *Zufälliges Verhalten einbringen* eine Methode der Klasse **Greenfoot**. Ihre Signatur lautet:

```
static boolean isKeyDown(String key)
```

Wie wir sehen, ist die Methode statisch (d.h., es ist eine Klassenmethode) und der Rückgabebetyp ist **boolean**. Das bedeutet, dass die Methode entweder **true** oder **false** zurückliefert und als Bedingung in einer **if**-Anweisung verwendet werden kann.

Wir sehen außerdem, dass die Methode einen Parameter vom Typ **String** erwartet. Ein **String** ist ein Text (z.B. ein Wort oder ein Satz), der in doppelten Anführungszeichen steht. Die folgenden Beispiele sind alles **String**-Beispiele:

```
"Dies ist ein String"
```

```
"Name"
```

```
"A"
```

In unserem Fall ist der erwartete **String** der Name der Taste, die wir testen wollen. Jede Taste der Tastatur hat einen Namen. Bei Tasten, die sichtbare Zeichen erzeugen, ist dieses Zeichen der Name. So heißt zum Beispiel die A-Taste „A“. Andere Tasten haben ebenfalls Namen. Die linke Cursortaste wird beispielsweise „left“ (links) genannt. Wenn wir also testen wollen, ob die linke Cursortaste gedrückt wurde, können wir Folgendes schreiben:

```
if (Greenfoot.isKeyDown("left"))
{
    ...// tue etwas
}
```

Beachte, dass wir „**Greenfoot.**“ vor dem Aufruf von **isKeyDown** schreiben müssen, da die Methode **isKeyDown** in der Klasse **Greenfoot** definiert ist.

Wenn wir beispielsweise erreichen wollen, dass sich unsere Krabbe bei jedem Drücken der linken Cursortaste um vier Grad nach links dreht, können wir schreiben:

```
if (Greenfoot.isKeyDown("left"))
{
    turn(-4);
}
```

Tipp!

Greenfoot speichert Klassen und Szenarien automatisch, wenn ihre Fenster geschlossen werden. Wenn du Kopien von Zwischenstadien der Szenarien behalten möchtest, wähle den Befehl **KOPIE SPEICHERN ALS** aus dem Menü **SZENARIO**.

Unser Plan ist es jetzt, die Codeabschnitte der Krabbe zu entfernen, die für die zufälligen Drehbewegungen sowie für das automatische Wenden der Krabbe am Rand der Welt zuständig sind, und ihn durch Code zu ersetzen, der es uns ermöglicht, das Drehen der Krabbe von der Tastatur aus zu steuern.

Übung 3.14 Entferne den Code für die zufälligen Drehbewegungen aus der Krabbe.

Übung 3.15 Entferne den Code, der dafür zuständig ist, dass die Krabbe am Rand der Welt wendet.

Übung 3.16 Füge Code in die **act**-Methode der Krabbe ein, der dafür sorgt, dass sich die Krabbe immer nach links dreht, wenn die linke Cursortaste gedrückt wird. Teste den Code.

Übung 3.17 Füge weiteren – ähnlichen – Code in die **act**-Methode der Krabbe ein, der dafür sorgt, dass sich die Krabbe immer nach rechts dreht, wenn die rechte Cursortaste gedrückt wird.

Übung 3.18 Sofern du es noch nicht gemacht hast, Sorge dafür, dass der Code, der das Drücken der Tasten prüft und ein Drehen veranlasst, nicht direkt in die **act**-Methode eingefügt wird, sondern in eine eigene Methode ausgelagert wird, die z.B. **checkKeypress** heißen könnte. Diese Methode sollte von der **act**-Methode aus aufgerufen werden.

Übung 3.19 Bisher hast du einfach den Krabben-Code in die **Lobster**-Klasse kopiert, beide bewegen sich daher mit derselben Geschwindigkeit. Du kannst das ändern und sie unterschiedlich schnell gehen lassen, indem du den Parameter im Methodenaufruf von **move(5)** in einer der beiden Klasse veränderst. Versuche, die Hummer schneller oder langsamer gehen zu lassen. Probiere aus, wie dir das Spiel gefällt, wenn du das machst. Wähle eine Geschwindigkeit für die Krabbe und den Hummer, die dir zusagt.

Versuche die Aufgaben erst einmal allein zu lösen. Wenn du hängen bleibst, lies einfach weiter. In Listing 3.5 findest du den vollständigen Code der Methoden **act** und **checkKeypress**, wie er nach der Überarbeitung aussieht. Die Lösung ist auch in den Buchszenarien verfügbar, siehe *little-crab-3*. Diese Version enthält alle Änderungen, die wir bisher besprochen haben.

```
/**
 * Tut, was auch immer Krabben gerne tun.
 */
public void act()
{
    checkKeypress();
    move(5);
    lookForWorm();
}
```

```
/**
 * Prüft, ob eine Steuertaste auf der Tastatur gedrückt wurde.
 * Wenn ja, reagiert die Methode entsprechend.
 */
public void checkKeypress()
{
    if (Greenfoot.isKeyDown("left"))
    {
        turn(-4);
    }
    if (Greenfoot.isKeyDown("right"))
    {
        turn(4);
    }
}
```

Listing 3.5

Die **act**-Methode zur Steuerung der Krabbe über die Tastatur.

Jetzt bist du so weit, dass du zum ersten Mal dein Spiel ausprobieren kannst! Platziere eine Krabbe, einige Würmer und ein paar Hummer in der Welt und finde heraus, ob du alle Würmer fressen kannst, bevor die Hummer dich fressen. (Offensichtlich wird es mit jedem weiteren Hummer immer schwerer ...)

3.7 Das Spiel beenden

Eine einfache Verbesserung, die wir noch vornehmen können, ist, das Spiel zu beenden, wenn die Krabbe von einem Hummer gefangen wurde. Hierfür gibt es bei Greenfoot eine Methode – wir müssen nur herausfinden, wie sie heißt.

Um festzustellen, wie die in Greenfoot verfügbaren Methoden heißen, können wir die Dokumentation der Greenfoot-Klassen aufrufen.

Wähle dazu in Greenfoot aus dem HILFE-Menü den Befehl GREENFOOT KLASSEN-DOKUMENTATION. Damit rufst du die Dokumentation aller Greenfoot-Klassen in einem Webbrowser auf (Abbildung 3.5).

Diese Dokumentation heißt auch *Greenfoot API* (Abkürzung für *Application Programmers' Interface*¹). Die API zeigt alle verfügbaren Klassen und beschreibt für jede Klasse die darin zur Verfügung stehenden Methoden. Wie du siehst, gibt es in Greenfoot sieben Klassen: **Actor**, **Greenfoot**, **GreenfootImage**, **GreenfootSond**, **MouseInfo**, **UserInfo** und **World**.

Konzept

Die **API-Dokumentation** bietet eine Übersicht über alle Klassen und Methoden, die in Greenfoot zur Verfügung stehen. Wir müssen in ihr oft nach Methoden suchen.

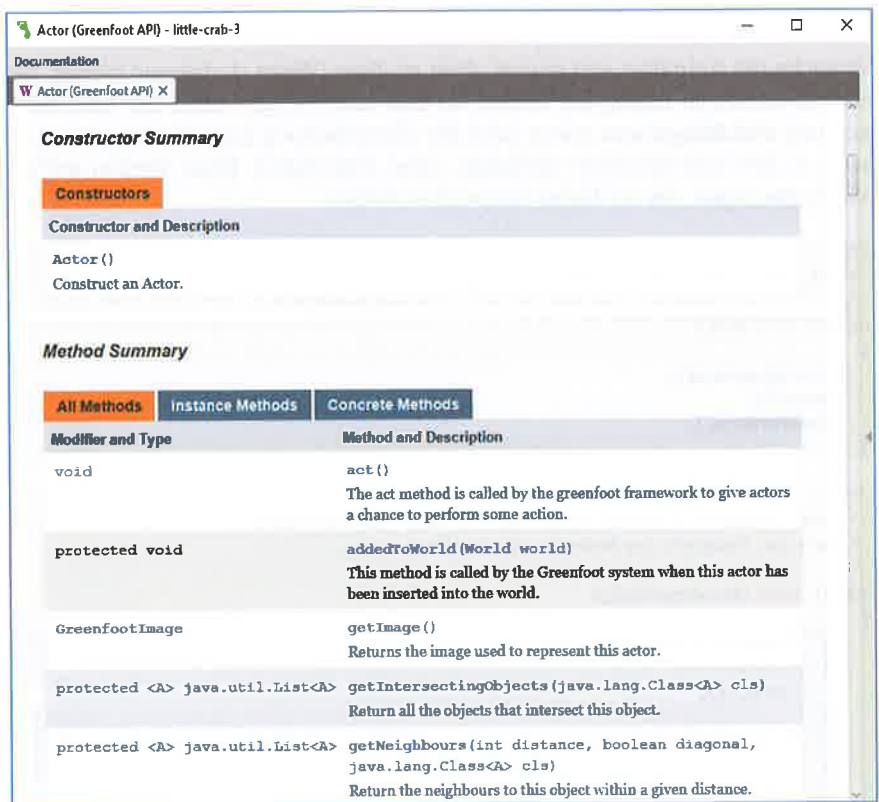


Abbildung 3.5
Die Greenfoot API in einem Browserfenster.²

Die Methode, nach der wir suchen, ist Teil der Klasse **Greenfoot**.

1 Schnittstelle für Anwendungsprogrammierer.
2 Du kannst unter <http://www.greenfoot.org/doc/translations.html> nach einer deutschen Übersetzung der Greenfoot API schauen.

Übung 3.20 Öffne die Greenfoot API in deinem Browser. Wähle die Klasse **Greenfoot**. In ihrer Dokumentation findest du einen Abschnitt mit der Überschrift „Method Summary“. Versuche, in diesem Abschnitt eine Methode zu finden, die die Ausführung eines laufenden Szenarios stoppt. Wie wird diese Methode genannt?

Übung 3.21 Erwartet diese Methode irgendwelche Parameter? Wie lautet ihr Rückgabebetyp?

Um die Dokumentation zu einer bestimmten Greenfoot-Klasse einzusehen, brauchen wir nur die entsprechende Klasse links in der Liste der Klassen auszuwählen. Für die ausgewählte Klasse werden dann im Hauptbereich des Browsers eine allgemeine Erläuterung, Einzelheiten zu ihren Konstruktoren und eine Liste ihrer Methoden angezeigt. (Konstruktoren sind Thema eines späteren Kapitels.)

Wenn wir die Liste der verfügbaren Methoden in der Klasse **Greenfoot** durchforsten, stoßen wir irgendwann auf eine Methode namens **stop**. Dies ist die Methode, mit der wir die Ausführung anhalten können, wenn die Krabbe gefangen wird.

Wir können diese Methode benutzen, indem wir

```
Greenfoot.stop();
```

in unseren Quelltext schreiben.

Übung 3.22 Füge deinem Szenario Code hinzu, der das Spiel beendet, wenn ein Hummer die Krabbe fängt. Dazu musst du entscheiden, wo dieser Code einzufügen ist. Suche nach der Codestelle, die ausgeführt wird, wenn ein Hummer eine Krabbe fängt, und füge hier die obige Codezeile ein.

Wir werden in Zukunft häufiger auf die Klassendokumentation zurückgreifen, um Einzelheiten zu den Methoden nachzulesen, die wir benutzen müssen. Nach einer Weile werden wir sicherlich einige Methoden auswendig kennen, doch es gibt immer Methoden, die wir nachschlagen müssen.

3.8 Sound hinzufügen

Wenn du dein Spiel noch weiter verbessern willst, könntest du zum Beispiel Sounds hinzufügen. Auch hierfür stellt uns die Klasse **Greenfoot** eine Methode zur Verfügung.

Übung 3.23 Öffne die GREENFOOT KLASSENDOKUMENTATION (über das HILFE-Menü), um herauszufinden, wie man Sounds abspielt. Es gibt eine Greenfoot-Klasse **GreenfootSound** für vollständige Soundsteuerung und außerdem eine einfachere Methode in der Klasse **Greenfoot**, um schnell und leicht einen Sound zu erzeugen. Suche nach den Details der Methode in der **Greenfoot**-Klasse. Wie lautet ihr Name? Welche Parameter erwartet sie?

Die Dokumentation verrät uns, dass es in der Klasse **Greenfoot** eine Methode namens **playSound** gibt. Diese Methode erwartet als Parameter den Namen einer Sounddatei (einen String) und liefert nichts zurück.

Hinweis

Vielleicht interessiert es dich, wie sich unsere Greenfoot-Szenarien im Dateisystem darstellen. In dem Ordner, der die Buchszenarien enthält, findest du, nach Kapiteln geordnet, einen Ordner für jedes Greenfoot-Szenario. So gibt es zum Beispiel für das Krabben-Beispiel mehrere verschiedene Versionen (*little-crab*, *little-crab-2*, *little-crab-3* usw.). In den jeweiligen Szenarien werden zu jeder Klasse mehrere Dateien sowie weitere Hilfsdateien gespeichert. Darüber hinaus gibt es zwei Ordner für Mediendateien: *images* enthält die Bilddateien des Szenarios und *sounds* speichert die Sounddateien.

Du kannst die verfügbaren Sounds einsehen, wenn du dir diesen Ordner anschaust, und du kannst weiteren Sound bereitstellen, indem du ihn hier speicherst.

In unserem Krabben-Szenario sind bereits zwei Sounddateien vorhanden. Sie heißen *slurp.wav* und *au.wav*.

Jetzt ist es nicht weiter schwer, einen dieser Sounds mithilfe des folgenden Methodenaufrufs abzuspielen:

```
Greenfoot.playSound("slurp.wav");
```

Versuche es einmal selber!

Übung 3.24 Ergänze dein Szenario um das Abspielen von Sounddateien: Wenn eine Krabbe einen Wurm frisst, spiele den Sound „slurp.wav“, und wenn ein Hummer die Krabbe frisst, spiele den Sound „au.wav“. Das Wichtigste dabei ist es, die richtige Position im Code zu finden, wo dies passieren soll.

In der Version *little-crab-4* dieses Szenarios findest du die Lösung hierzu. In dieser Projektversion findest du all die Funktionalität, die wir bisher besprochen haben: Würmer, Hummer, Tastatursteuerung und Sound (Abbildung 3.6).

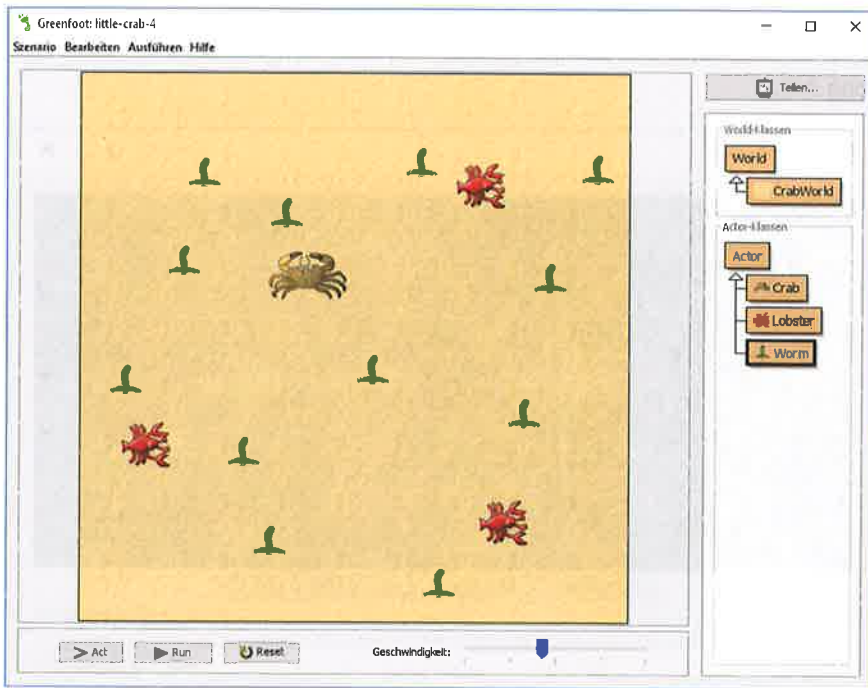


Abbildung 3.6
Das Krabben-Spiel
mit Würmern und
Hummern.

3.9 Eigenen Sound herstellen

Es gibt unterschiedliche Möglichkeiten, deinen eigenen Sound zu einem Greenfoot-Szenario hinzuzufügen. Du kannst zum Beispiel in vielen Sound-Bibliotheken im Internet Soundeffekte finden oder deinen eigenen Sound mit Programmen zur Soundaufnahme und -bearbeitung erzeugen.

Der Einsatz von Sounds ist manchmal ein wenig knifflig, weil Sounds auf einem Computer in vielen unterschiedlichen Dateiformaten gespeichert werden können. Greenfoot kann einige davon abspielen, jedoch nicht alle. Im Allgemeinen kann Greenfoot Sounddateien im MP3-, AIFF-, AU- und WAV-Format abspielen (allerdings nur bestimmte WAV-Dateien – es wird kompliziert).

Wir werden das Thema Sounds ausführlicher in **Kapitel 10** besprechen. Hier wollen wir die einfachste Methode verwenden, um unseren eigenen Sound zu unserem Szenario hinzuzufügen: wir nehmen ihn direkt in Greenfoot auf.

Beide Soundeffekte, die wir im vorigen Abschnitt verwendet haben, wurden aufgezeichnet, indem wir einfach in das Mikrofon gesprochen haben. Greenfoot hat ein eingebautes Aufnahmegerät, den Klang-Rekorder, mit dem du dasselbe machen kannst.

Um ihn zu starten, wähle die Funktion KLANG-REKORDER ANZEIGEN aus dem AUSFÜHREN-Menü. Es wird sich dann ein neues Fenster mit dem Klang-Rekorder öffnen (Abbildung 3.7).

Abbildung 3.7
Der Klang-Rekorder
von Greenfoot.



Mithilfe des Klang-Rekorders kannst du nun deine eigenen Sounds aufnehmen. Dazu klickst du auf den AUFNAHME-Button und sprichst in das Mikrofon.³ Drücke AUFNAHME STOPPEN, wenn du fertig bist. Du kannst den ABSPIELEN-Button verwenden, wenn du kontrollieren möchtest, wie sich deine Aufnahme anhört.

Es gibt nur eine Möglichkeit der Bearbeitung: AUF AUSWAHL KÜRZEN. Diese Funktion dient dazu, unerwünschte Teile der Aufnahme am Anfang und am Ende herauszuschneiden. Häufig hat man ein paar Geräusche oder Stille am Anfang der Aufnahme, und dies hört sich in deinem Programm nicht gut an: durch Stille am Anfang wirkt der Sound verzögert.

Um die unerwünschten Teile zu entfernen, markierst du mit deiner Maus den Teil des Sounds, den du behalten möchtest (Abbildung 3.8), und drückst dann AUF AUSWAHL KÜRZEN.

³ Offensichtlich funktioniert das nur, wenn dein Computer ein Mikrofon besitzt. Bei den meisten modernen Laptops ist ein Mikrofon integriert. Bei einigen Desktoprechnern wirst du jedoch ein externes Mikrofon anschließen müssen. Wenn du kein Mikrofon hast, überspringe diesen Abschnitt einfach.

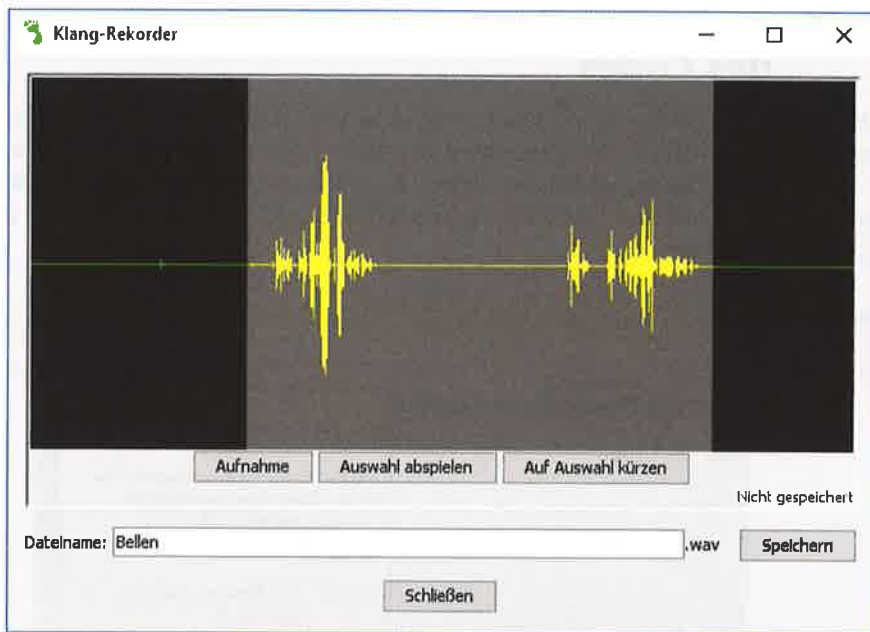


Abbildung 3.8
Auswahl des guten
Teils der Aufnahme.

Zum Schluss wählst du einen Namen für deinen Sound: tippe ihn in das Feld für den Dateinamen und klicke auf **SPEICHERN**. Der Sound wird im WAV-Format gespeichert und erhält automatisch das Suffix *wav*. Falls du also beispielsweise deine Sounddatei „bellen“ nennst, dann wird sie als *bellen.wav* gespeichert. Greenfoot speichert die Datei automatisch an der richtigen Stelle (im Ordner *sounds* deines Szenario-Ordners).

Du kannst den Sound dann in deinem Code mithilfe der Methode verwenden, die wir vorher kennengelernt haben:

```
Greenfoot.playSound("bellen.wav");
```

Nun ist es Zeit, dass du es selbst ausprobierst.

Übung 3.25 Wenn dir bei deinem Computer ein Mikrofon zur Verfügung steht, erstelle deine eigenen Sounds, um das Fressen der Würmer oder der Krabbe zu untermalen. Nimm die Sounds auf und verwende sie in deinem Code.

3.10 Automatische Vervollständigung des Codes

Tipp!

Verwende **Code-Vervollständigung**, um die Eingabe von Methodenaufrufen zu vereinfachen.

Um das Arbeiten effizienter zu gestalten, ist es hilfreich, *Code-Vervollständigung* zu verwenden, um deine Methodenaufrufe einzugeben (Abbildung 3.9). Du kannst Code-Vervollständigung verwenden, wenn du gerade den Namen einer Methode eintippst, die du aufrufen möchtest. Die Funktion wird durch Drücken von `[Strg]` + `[Leertaste]` aktiviert.

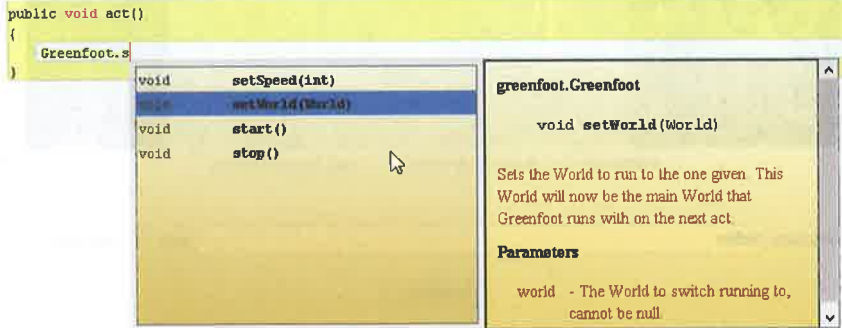


Abbildung 3.9
Verwendung von Code-Vervollständigung (`[Strg]` + `[Leertaste]`).

Du hast zum Beispiel

Greenfoot.

eingetragen und der Cursor befindet sich hinter dem Punkt. Wenn du jetzt `[Strg]` + `[Leertaste]` drückst, dann öffnet sich ein Dialogfenster, in dem alle Methoden aufgeführt sind, die du hier aufrufen kannst (in diesem Fall sind es alle Methoden der **Greenfoot**-Klasse). Tippst du nun den Anfang eines Methodennamens ein (in Abbildung 3.9 haben wir zum Beispiel den Buchstaben „s“ eingegeben), dann verkürzt sich die Liste der Methoden auf diejenigen, deren Namen mit den eingegebenen Zeichen beginnen.

Du kannst eine Methode aus der Liste mithilfe deiner Cursortasten oder mit der Maus auswählen, und du fügst sie mit der `[↵]`-Taste in deinen Code ein.

Die Code-Vervollständigung ist nützlich, wenn du herausfinden möchtest, welche Methoden es gibt; um eine Methode zu finden, von der du weißt, dass sie existiert, aber du dich nicht an ihren genauen Namen erinnern kannst; um Parameter oder die Dokumentation nachzulesen; oder einfach um dir ein wenig Tipparbeit zu sparen und die Eingabe deines Codes zu beschleunigen.

Zusammenfassung der Programmieretechniken

In diesem Kapitel haben wir weitere Anwendungsbeispiele zur **if**-Anweisung gesehen – beispielsweise bei den Zufallsdrehungen und der Reaktion auf das Drücken von Tasten. Wir haben auch gesehen, wie man Methoden aus einer anderen Klasse aufruft, speziell an den Methoden **getRandomNumber**, **isKeyDown** und **playSound** der Klasse **Greenfoot**. Hierzu haben wir die Punktnotation verwendet, bei der der Klassenname vor dem Punkt aufgeführt wird.

Insgesamt haben wir jetzt drei verschiedene Codestellen kennengelernt, von denen aus Methoden aufgerufen werden können. Wir können Methoden aufrufen, die in der aktuellen Klasse selbst definiert sind (sogenannte *lokale Methoden*), Methoden, die in einer Oberklasse definiert wurden (*geerbte Methoden*) und statische Methoden von anderen Klassen. Für den letzteren Aufruf verwenden wir die Punktnotation. (Es gibt noch eine weitere Version eines Methodenaufrufs: Methoden von anderen Objekten aufrufen – hierauf werden wir später noch eingehen.)

Ein weiterer wichtiger Punkt, den wir angesprochen haben, war die API-Dokumentation der bereits vorhandenen Klassen. Wir haben gesehen, wie man herausfindet, welche Methoden es zu den einzelnen Klassen gibt und wie sie aufgerufen werden.

Zu guter Letzt haben wir ein sehr wichtiges Konzept kennengelernt: die Fähigkeit, unsere eigenen Methoden zu definieren. Wir haben gesehen, wie Methoden für klar unterscheidbare Unteraufgaben definiert werden und wie sie von anderen Methoden aufgerufen werden.



Zusammenfassung der Konzepte

- Wenn wir eine Methode aufrufen wollen, die weder in unserer Klasse definiert ist noch geerbt wurde, müssen wir die Klasse oder das Objekt, in der sich die Methode befindet, getrennt durch einen Punkt vor dem Methodennamen angeben. Diese Schreibweise wird auch **Punktnotation** genannt.
- Methoden, die zu Klassen (und nicht zu Objekten) gehören, werden in ihrer Signatur durch das Schlüsselwort **static** gekennzeichnet. Sie werden auch als **Klassenmethoden** bezeichnet.
- Eine **Methodendefinition** definiert eine neue Aktion für Objekte dieser Klasse. Die Aktion wird nicht direkt ausgeführt, aber die Methode kann mittels eines Methodenaufrufs später ausgeführt werden.
- **Kommentare** sind an Programmierer gerichtete Erläuterungen, die in den Quelltext geschrieben werden; vom Computer werden sie ignoriert.
- Die **API-Dokumentation** bietet eine Übersicht über alle Klassen und Methoden, die in Greenfoot zur Verfügung stehen. Wir müssen in ihr oft nach Methoden suchen.



Vertiefende Aufgaben

Die Konzepte, die wir hier vertiefen wollen, sind (noch einmal) **if**-Anweisungen, das Lesen von API-Dokumentationen, das Aufrufen einer Methode von anderen Klassen und das Erzeugen unserer eigenen Methoden.

Dazu wollen wir ein weiteres Szenario verwenden: *stickman*. Suche es in den Buchszenarien und öffne es.

Lesen von API-Dokumentationen

Übung 3.26 Die **Greenfoot**-Klasse besitzt eine Methode, mit der man den Geräuschpegel vom Mikrofon des Computers erhält. Finde diese Methode in der API-Dokumentation. Wie viele Parameter hat sie?

Übung 3.27 Was ist der Rückgabotyp dieser Methode? Was sagt uns dies?

Übung 3.28 Welches sind die möglichen Werte, die von dieser Methode zurückgegeben werden?

Übung 3.29 Ist diese Methode **static** oder nicht? Was sagt uns das?

Übung 3.30 Wie können wir diese Methode aufrufen? Schreibe einen korrekten Methodenaufruf für diese Methode.

Aufrufen von Klassenmethoden / if-Anweisungen

Übung 3.31 Bewege das Strichmännchen in deinem *stickman*-Szenario nach rechts, sodass es bei Ausführung deines Szenarios über die rechte Seite des Bildschirms wandert.

Übung 3.32 Lasse das Strichmännchen nur dann nach rechts gehen, wenn du ein Geräusch machst. Verwende dazu eine **if**-Anweisung und die Eingabemethode über das Mikrofon, die du in der Übung 3.26 gefunden hast. Experimentiere mit unterschiedlichen Werten für den Geräuschpegel. Dieser ist von deinem Mikrofon und deiner Umgebung abhängig. Beginne damit, das Strichmännchen loszuschicken, wenn der Pegel größer als 3 ist. Teste deinen Code.

Übung 3.33 Lasse das Strichmännchen nach links laufen, wenn du irgendein Geräusch machst, aber es soll sich ständig nach rechts bewegen, wenn es kein Geräusch gibt. Teste den Code. Versuche, das Strichmännchen durch Rufen in der Nähe der Bildschirmmitte zu halten.

Methoden definieren

Übung 3.34 Verschiebe den Code, der für die Linksbewegung bei Rufen zuständig ist, in eine eigene Methode. Nenne diese Methode **MoveLeftIfNoise**. Teste den Code. Stelle sicher, dass alles wie zuvor funktioniert.

Übung 3.35 Füge eine weitere Methode zu deiner Klasse hinzu, die **GameOver** heißt. Diese Methode sollte aufgerufen werden, wenn das Strichmännchen das Ende des Bildschirms erreicht. Wenn sie aufgerufen wird, spielt die Methode einen „Spielende“-Sound ab und beendet das Szenario.

Übung 3.36 Verschiebe die Überprüfung der Bedingung für das Spielende ebenfalls in eine separate Methode, die **checkGameOver** heißt. Die **act**-Methode sollte **checkGameOver** aufrufen, welche wiederum gegebenenfalls **gameOver** aufruft.

Übung 3.37 Lasse das Strichmännchen bei einem Geräusch nach oben schweben. Die Höhe, bis zu der es schwebt, sollte dem Geräuschpegel entsprechen. Das Strichmännchen kommt wieder herunter, wenn es kein Geräusch mehr gibt. Beachte: die Forderung, dass die Höhe der Lautstärke entspricht, ist ziemlich schwierig – du musst einige Dinge nachlesen, die wir noch nicht besprochen haben. Wenn du es als zu schwer empfindest, lasse diese Übung aus.

Übung 3.38 Führe einen weiteren Akteur ein (vielleicht ein Tier), der am linken Bildschirmrand startet und sich seitwärts bewegt. Wenn er den rechten Bildschirmrand erreicht, wird er wieder zum linken Rand zurückgesetzt. Auf seinem Pfad über den Bildschirm sollte er das Strichmännchen berühren, wenn dieser sich nicht bewegt. Du kannst dann das Strichmännchen über den Akteur hüpfen lassen, indem du irgendein Geräusch machst.

Übung 3.39 Beende das Szenario (mit Spielende-Sound), wenn das Strichmännchen das Tier berührt.

