KAPITEL

2 Das erste Programm: Little Crab[1]

Lernziele

Themen: Code schreiben: Objekte bewegen, drehen und auf den Bildschirmrand reagieren lassen

Konzepte: Quelltext, Methodenaufruf, Parameter, Folge, `if`-Anweisung

Im vorherigen Kapitel haben wir besprochen, wie bestehende Greenfoot-Szenarien verwendet werden: Wir haben Objekte erzeugt, Methoden aufgerufen und ein Spiel gespielt.

Jetzt wollen wir uns daran wagen, ein eigenes Spiel zu entwickeln.

2.1 Das little-crab-Szenario

Das Szenario, das wir für dieses Kapitel verwenden, heißt *little-crab*. Du findest es im Ordner *book-scenarios*.

Das Szenario sollte wie in Abbildung 2.1 aussehen.

Übung 2.1 Starte Greenfoot und öffne das Szenario *little-crab*. Platziere eine Krabbe in der Welt und führe das Programm aus (d.h., klicke auf den RUN-Button). Was kannst du beobachten? (Zur Erinnerung: Wenn die Symbole im Klassenmenü ganz rechts schraffiert angezeigt werden, musst du das Projekt zuerst übersetzen.)

Zur Rechten siehst du die Klassen dieses Szenarios (Abbildung 2.2). Wir stellen fest, dass es neben den bereits bekannten Greenfoot-Klassen **`Actor`** und **`World`**-Unterklassen namens **`CrabWorld`** und **`Crab`** (Krabbe) gibt.

1 „little crab" heißt auf Deutsch „kleine Krabbe".

Die Hierarchie (ausgedrückt durch die Pfeile) weist auf eine *Ist-eine*-Beziehung hin. Dies wird auch *Vererbung* genannt. Eine Krabbe *ist ein* Akteur und die Krabbenwelt *ist eine* Welt. Daraus folgt, dass eine Krabbe auch ein Akteur ist.

Zu Beginn werden wir nur mit der **Crab**-Klasse arbeiten. Später werden wir noch etwas näher auf die Klassen **`Actor`** und **`CrabWorld`** eingehen.

Wenn du gerade die Übung oben nachvollzogen hast, weißt du inzwischen die Antwort auf die Frage „Was kannst du beobachten?" Sie lautet: „Nichts".

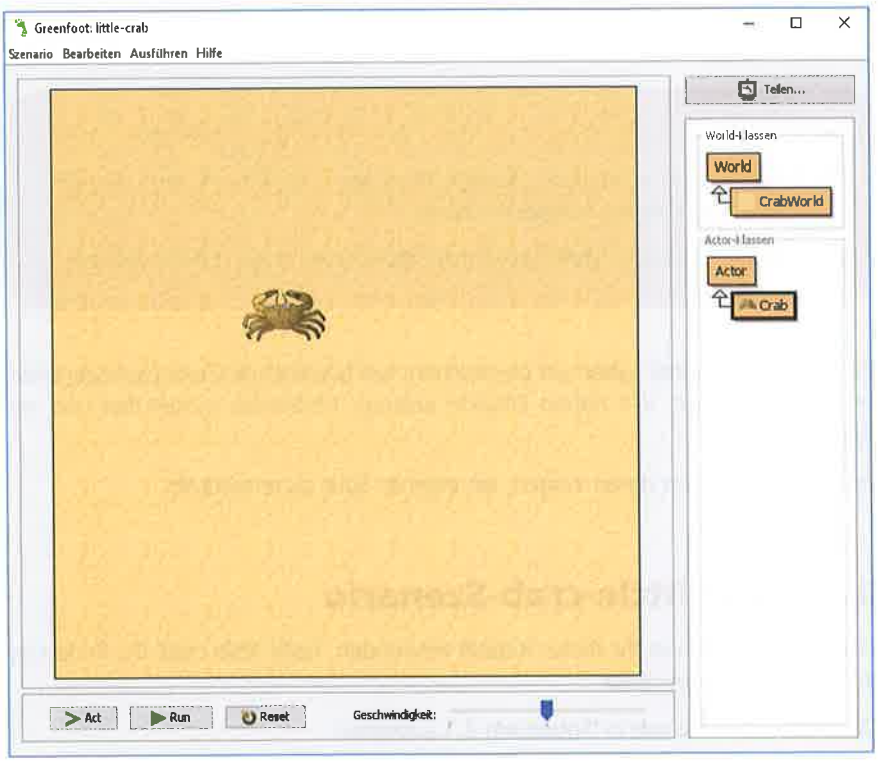

Abbildung 2.1
Das Szenario *Little Crab*.

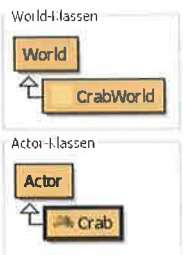

Abbildung 2.2
Die Klassen von *Little Crab*.

Die Krabbe macht rein gar nichts, wenn du Greenfoot ausführst. Das liegt daran, dass es in der Definition der **Crab**-Klasse keinen Quelltext gibt, der vorgibt, was die Krabbe tun soll.

In diesem Kapitel wollen wir uns der Aufgabe widmen, dies zu ändern. Das Erste, was wir machen werden, ist, die Krabbe in Bewegung zu setzen.

2.2 Die Krabbe in Bewegung setzen

Lass uns einmal einen Blick in den Quelltext der Klasse **Crab** werfen. Öffne den Editor, um den Code von **Crab** anzuzeigen. (Hierzu kannst du den Befehl EDITOR ÖFFNEN aus dem Kontextmenü der Klasse wählen oder einfach die Klasse doppelt anklicken.)

Der Quelltext, den du siehst, ist in Listing 2.1 wiedergegeben.

Listing 2.1
Der Originalcode der Klasse **Crab**.

```
import greenfoot.*;

/**
 * Diese Klasse definiert eine Krabbe. Krabben leben am Strand.
 */
public class Crab extends Actor
{
    public void act()
    {
        // Füge hier deinen Code ein.
    }
}
```

So sieht die Standarddefinition einer Klasse in Java aus. Das heißt, dieser Text definiert, was die Krabbe machen kann.

Dir sind sicherlich die unterschiedlichen Hintergrundfarben aufgefallen. Die gesamte Klassendefinition steht in einem grünen Kasten und darin wiederum befinden sich die einzelnen Methodendefinitionen in jeweils eigenem Kasten mit gelbem Hintergrund. (Es gibt außerdem ganz oben, vor der Klassendefinition, noch eine gesonderte Anweisung auf weißem Hintergrund.)

Wir werden gleich näher darauf eingehen. Zuerst einmal wollen wir dafür sorgen, dass sich die Krabbe bewegt.

In unserer Klassendefinition finden wir die Methode **act**. Sie sieht folgendermaßen aus:[2]

```
public void act()
{
    // Füge hier deinen Code ein.
}
```

Die erste Zeile ist die *Signatur* der Methode. Die letzten drei Zeilen – d.h. die beiden geschweiften Klammern und alles, was dazwischen steht – bilden den *Rumpf* der Methode. Hier können wir Code einfügen, der die Aktionen der Krabbe festlegt. Wir können den Kommentartext in der Mitte durch einen Befehl ersetzen. Ein solcher Befehl lautet zum Beispiel:

```
move(5);
```

2 Wenn wir Code innerhalb des Textes einfügen, werden wir die Hintergrundfarben nicht darstellen. Keine Sorge: die Farben verändern die Bedeutung des Codes nicht. Sie unterstützen dich nur dabei, deinen Code in Greenfoot zu lesen und zu schreiben.

Achte bitte darauf, dass der Befehl genau so geschrieben wird, einschließlich der Klammern und des Semikolons. Danach sollte die **act**-Methode wie folgt lauten:

```
public void act()
{
    move(5);
}
```

Übung 2.2 Ergänze die **act**-Methode in deiner Klasse **Crab** um den Befehl **move(5)**, wie oben gezeigt. Übersetze das Szenario (durch Klicken auf den Button ALLE ÜBERSETZEN)[3] und platziere eine Krabbe in der Welt. Klicke jetzt auf die Buttons ACT und RUN.

Übung 2.3 Ändere die Zahl 5 in eine andere Zahl. Probiere größere und kleinere Zahlen aus. Was, glaubst du, bedeuten die Zahlen?

Übung 2.4 Platziere mehrere Krabben in der Welt. Führe das Szenario aus. Was kannst du beobachten?

Du wirst feststellen, dass sich die Krabbe jetzt über den Bildschirm bewegen kann. Der Befehl **move(5)** veranlasst die Krabbe, sich ein Stück nach rechts zu bewegen.

Wenn wir auf den ACT-Button im Hauptfenster von Greenfoot klicken, wird die **act**-Methode nur einmal ausgeführt, d.h., der Befehl **move(5)**, den wir in die **act**-Methode eingefügt haben, wird ausgeführt. Die Zahl 5 im Befehl definiert, wie weit sich die Krabbe mit jedem Schritt fortbewegt: Bei jedem ACT-Schritt bewegt sich die Krabbe fünf Pixel nach rechts.

Das Anklicken des RUN-Buttons ist im Prinzip nichts anderes als eine sehr schnelle wiederholte Betätigung des ACT-Buttons. Die **act**-Methode wird so lange ausgeführt, bis wir auf PAUSE klicken.

Übung 2.5 Kannst du herausfinden, wie man die Krabbe rückwärts laufen lässt (nach links laufen lässt)?

Konzept

Ein **Methodenaufruf** ist ein Befehl, der ein Objekt anweist, eine Aktion auszuführen. Die Aktion wird durch eine Methode des Objekts definiert.

Terminologie

Der Befehl **move()** wird **Methodenaufruf** genannt. Eine **Methode** ist eine Aktion, die das Objekt (hier die Krabbe) ausführen kann, und ein **Methodenaufruf** ist ein Befehl, der die Krabbe anweist, eben diese Aktion auszuführen. Die Klammern und die Zahl darin sind Teil des Methodenaufrufs. Befehle wie diese werden mit einem Semikolon beendet.

3 Diesen Button gibt es in Version 3.0 nicht mehr, siehe Bemerkung dazu in Kapitel 1.

2.3 Drehen

Lass uns einmal schauen, ob es noch weitere Befehle gibt, die wir verwenden können. Die Krabbe versteht z.B. auch den Befehl **turn**, der folgendermaßen aussieht:

```
turn(3);
```

Die Zahl **3** in dem Befehl gibt an, um wie viel Grad die Krabbe sich drehen soll. Diesen Wert nennt man auch *Parameter*. (Die Zahl 5, die wir oben für den **move**-Aufruf verwendet haben, ist ebenfalls ein Parameter.)

Konzept

Einigen Methoden kannst du in Klammern zusätzliche Informationen übergeben. Der übergebene Wert wird **Parameter** genannt.

Wir können hierfür auch andere Werte angeben, zum Beispiel:

```
turn(23);
```

Die Gradangaben sollten 360 Grad nicht überschreiten, d.h., es kann ein Wert zwischen 0 und 359 Grad verwendet werden. (Würden wir uns um 360 Grad drehen, hätten wir uns einmal komplett gedreht, was einer Drehung von 0 Grad bzw. keiner Drehung entspräche.)

Wenn wir uns lieber drehen wollen, anstatt uns vorwärts zu bewegen, brauchen wir nur den **move(5)**-Befehl durch einen **turn(3)**-Befehl zu ersetzen. (Die Parameterwerte, in diesem Fall 5 und 3, sind mehr oder weniger willkürlich gewählt; du kannst ebenso gut andere Werte nehmen.) Die **act**-Methode sähe dann folgendermaßen aus:

```
public void act()
{
    turn(3);
}
```

Übung 2.6 Ersetze in deinem Szenario den Befehl **move(5)** durch den Befehl **turn(3)**. Führe das Szenario aus. Gib probeweise auch andere Werte statt 3 ein und beobachte, was passiert. Denke daran: Jedes Mal, wenn du deinen Quelltext geändert hast, musst du ihn erneut übersetzen.

Übung 2.7 Wie kannst du erreichen, dass sich die Krabbe links herum dreht?

Als Nächstes probieren wir, die Krabbe gleichzeitig zu drehen und vorwärts zu bewegen. Da in der **act**-Methode mehr als ein Befehl stehen kann, schreiben wir einfach mehrere Befehle untereinander.

Konzept

Mehrere Befehle werden **nacheinander** ausgeführt, und zwar in der Reihenfolge, in der sie im Code erscheinen.

Listing 2.2 zeigt, wie der vollständige Code der **Crab**-Klasse aussieht, wenn die Krabbe gedreht und vorwärts bewegt wird. In unserem Fall bewegt sich die Krabbe bei jedem Anklicken des ACT-Buttons zuerst vorwärts und dreht sich dann (dabei laufen die beiden Aktionen so schnell hintereinander ab, dass der Eindruck vermittelt wird, es würde gleichzeitig passieren).

Listing 2.2: Die Krabbe drehen und vorwärts bewegen.

```
import greenfoot.*;

/**
 * Diese Klasse definiert eine Krabbe. Krabben leben am Strand.
 */
public class Crab extends Actor
{
    public void act()
    {
        move(5);
        turn(3);
    }
}
```

Übung 2.8 Probiere es aus: Verwende in der **act**-Methode deiner Krabbe sowohl einen Befehl **move(N)** als auch einen Befehl **turn(M)**. Gib verschiedene Werte für **N** und **M** ein.

Terminologie

Die Zahl in den Klammern des **turn**-Befehls – d.h. die **3** in **turn(3)** – wird **Parameter** genannt. Ein Parameter ist eine zusätzliche Information, die wir mitliefern müssen, wenn wir bestimmte Methoden aufrufen.

Einige Methoden erwarten keine Parameter. Wir schreiben hier nur den Methodennamen und die Klammern, aber nichts in die Klammern, zum Beispiel **stop()**. Andere Methoden wie **turn** und **move** benötigen weitere Informationen: *Um wie viel soll gedreht werden? Wie weit soll die Krabbe laufen?* In einem solchen Fall müssen wir die Information in Form eines Parameterwertes innerhalb der Klammern angeben, z.B. **turn(17)**.

Konzept

Wenn eine Klasse übersetzt wird, prüft der Compiler, ob irgendwelche Fehler vorliegen. Wird ein solcher Fehler gefunden, wird eine **Fehlermeldung** angezeigt.

Randbemerkung: Fehler

Wenn wir Quelltext schreiben, sollten wir größte Sorgfalt walten lassen: jedes einzelne Zeichen zählt. Wird auch nur ein kleiner Fehler gemacht, funktioniert unser Programm nicht mehr. In der Regel lässt es sich nicht mehr übersetzen.

Das wird uns immer wieder passieren: Wenn wir Programme schreiben, machen wir unweigerlich Fehler, die wir dann korrigieren müssen. Das wollen wir einmal ausprobieren.

Wenn wir zum Beispiel das Semikolon hinter dem Befehl **move(5)** vergessen, werden wir bei dem Versuch, unseren Code zu übersetzen, darauf hingewiesen.

Übung 2.9 Öffne den Editor, um den Quelltext der Krabbe anzuzeigen, und entferne das Semikolon hinter `move(5)`. Übersetze den Code. Experimentiere auch mit anderen Fehlern herum, wie eine falsche Schreibweise von `move` oder anderen Zufallsänderungen am Code. Achte jedoch darauf, dass nach der Übung alle Änderungen wieder zurückgenommen werden.

Übung 2.10 Nimm verschiedene Änderungen vor, um unterschiedliche Fehlermeldungen zu erhalten. Finde mindestens fünf verschiedene Fehlermeldungen. Notiere dir jede Fehlermeldung und durch welchen Fehler sie ausgelöst wurde.

Diese Übung demonstriert beispielhaft, wie Greenfoot bereits bei dem kleinsten Fehler den Editor öffnet, eine Zeile hervorhebt und unten im Editorfenster eine Meldung anzeigt. Diese Meldung versucht den Fehler zu erläutern. Die Meldungen sind jedoch hinsichtlich ihrer Genauigkeit und Nützlichkeit höchst unterschiedlich. Manchmal verraten sie uns ziemlich genau, wo das Problem liegt, aber in anderen Fällen sind sie nur kryptisch und schwer zu verstehen. Die hervorgehobene Zeile ist oft die Zeile mit dem Problem, doch manchmal tritt das Problem auch in der Zeile davor auf. Wenn du zum Beispiel die Meldung „`; expected`" erhältst, kann es vorkommen, dass das Semikolon in Wahrheit in der Zeile über der hervorgehobenen Zeile fehlt.

Nach und nach werden wir lernen, diese Meldungen besser zu verstehen. Im Moment solltest du jedenfalls den Code sehr sorgfältig durchgehen und prüfen, ob du alles richtig eingegeben hast, wenn du eine Meldung erhältst und nicht sicher bist, was damit gemeint ist.

Tipp!

Wenn eine Fehlermeldung unten im Editorfenster angezeigt wird, erscheint rechts davon ein Button mit einem *Fragezeichen*. Durch Anklicken dieses Buttons erhältst du zusätzliche Informationen zur Fehlermeldung.

2.4 Bildschirmränder

Als wir in den vorherigen Abschnitten die Krabben zum Laufen und Drehen gebracht haben, blieben die Krabben am Rand des Bildschirms hängen. (Greenfoot ist so konzipiert, dass die Akteure die Welt nicht verlassen und an ihrem Rand herunterfallen können.)

Dieses Verhalten wollen wir jetzt verbessern, sodass die Krabbe merkt, wenn sie das Ende der Welt erreicht hat, und wendet. Die Frage ist nur, wie wir dies hinbekommen.

Oben haben wir die Methoden `move` und `turn` verwendet, sodass es vielleicht auch für dieses Problem eine passende Methode gibt (was der Fall ist). Doch wie finden wir heraus, welche Methoden uns zur Verfügung stehen?

Die Methoden `move` und `turn`, die wir bisher verwendet haben, stammen beide von der Klasse `Actor` ab. Eine Krabbe ist ein Akteur (im Klassendiagramm durch einen Pfeil von `Crab` zu `Actor` ausgedrückt) und kann deshalb alles machen, was ein Akteur machen kann. Unsere Klasse `Actor` weiß, wie Bewegungen und Drehungen ausgeführt werden, und deshalb weiß das unsere Krabbe auch.

Konzept

Eine Unterklasse **erbt** alle Methoden ihrer Oberklasse. Das bedeutet, dass die Unterklasse alle von der Oberklasse definierten Methoden besitzt und nutzen kann.

Dies nennt man *Vererbung*. Die Klasse **Crab** erbt alle Fähigkeiten (Methoden) von der Klasse **Actor**.

Die Frage bleibt, was unsere Akteure sonst noch alles können?

Um dies herauszufinden, können wir die **Actor**-Klasse öffnen. Wenn du (mit einem Doppelklick) die **Actor**-Klasse öffnest, wird dir auffallen, dass sie sich nicht in einem Text-Editor wie die **Crab**-Klasse öffnet, sondern stattdessen eine Art Dokumentation in einem Webbrowser angezeigt wird (Abbildung 2.3). Dies liegt daran, dass die **Actor**-Klasse ein fester Bestandteil von Greenfoot ist: sie kann nicht bearbeitet werden. Aber wir können trotzdem die Methoden von **Actor** benutzen und sie aufrufen. Diese Dokumentation sagt uns, welche Methoden es gibt, welche Parameter sie haben und was sie tun. (Wir können uns auch die Dokumentationen der anderen Klassen ansehen, indem wir im Editor von QUELLTEXT zu DOKUMENTATION wechseln, über das Auswahl-Listenfeld oben rechts vom Editor-Fenster. Für **Actor** gibt es aber nur die Ansicht DOKUMENTATION.)

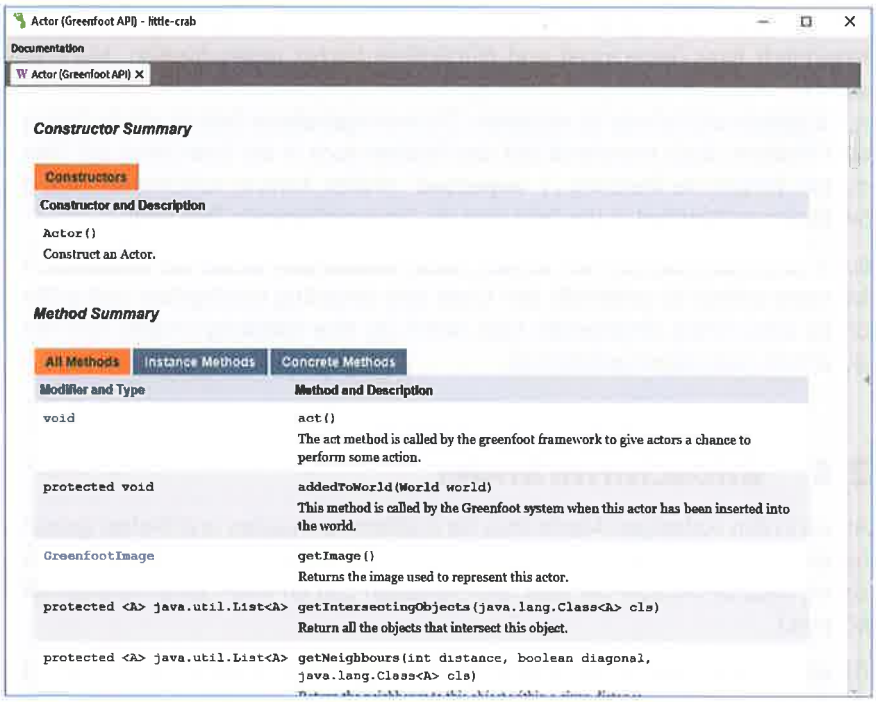

Abbildung 2.3
Dokumentation der Klasse **Actor**.

Übung 2.11 Öffne die Dokumentation für die Klasse **Actor**. Scrolle zur Liste der Methoden für diese Klasse (die „Method Summary"). Wie viele Methoden hat diese Klasse?

Übung 2.12 Gehe die Liste der verfügbaren Methoden durch. Kannst du eine finden, die sich so anhört, als könnte sie uns dabei helfen zu prüfen, ob wir den Rand der Welt erreicht haben?

In der Auflistung der Methoden (*method summary*) finden wir alle Methoden, die die Klasse **Actor** bereitstellt. Drei davon sind für uns im Moment besonders interessant:

boolean isAtEdge()
Stellt fest, ob der Akteur den Rand der Welt erreicht hat.

void move(int distance)
Rückt diesen Akteur um die angegebene Strecke in die Richtung vor, in die er aktuell blickt.

void turn(int amount)
Dreht diesen Akteur um die angegebene Zahl (in Grad).

Was wir hier sehen, sind die Signaturen für drei Methoden, wie wir sie bereits aus ***Kapitel 1*** kennen. Jede Signatur beginnt mit einem Rückgabetyp, gefolgt von dem Methodennamen und der Parameterliste. Darunter schließt sich ein Kommentar an, der beschreibt, was die Methode macht. Die drei Methodennamen lauten **isAtEdge**, **move** und **turn**.

Die beiden Methoden **move** und **turn** haben wir bereits in den vorherigen Abschnitten benutzt. Wenn wir ihre Parameterlisten betrachten, stellen wir fest, dass sich nichts geändert hat: beide erwarten einen Parameter vom Typ **int** (eine ganze Zahl). Für die Methode move wird damit die zu gehende Strecke angegeben, für die Methode turn ist es die Gradzahl, um die sich der Akteur dreht. (Wiederhole ***Kapitel 1.5***, wenn du nicht mehr genau weißt, was Parameterlisten sind.)

Außerdem stellen wir fest, dass die Methoden **move** und **turn** als Rückgabetyp **void** aufweisen. Das bedeutet, dass keine dieser Methoden einen Wert zurückliefert. Wir befehlen dem Objekt mehr oder weniger, sich zu bewegen oder zu drehen. Die Krabbe gehorcht einfach dem Befehl und gibt uns keine Antwort.

Die Signatur von **isAtEdge** weicht hiervon etwas ab. Sie lautet:

```
boolean isAtEdge()
```

Diese Methode hat keine Parameter (es steht nichts in den Klammern), dafür aber einen Rückgabewert: **boolean**. Wir sind bereits in ***Kapitel 1.4*** kurz auf den Typ **boolean** eingegangen. Dieser Typ zeichnet sich dadurch aus, dass er zwei Werte halten kann: **true** (wahr) oder **false** (falsch).

Durch Aufrufen von Methoden, die einen Rückgabewert aufweisen (d.h., bei denen der Rückgabetyp nicht **void** ist), wird kein Befehl ausgegeben, sondern eher eine Frage gestellt. Wenn wir die Methode **isAtEdge()** verwenden, antwortet die Methode entweder mit **true** (Ja!) oder **false** (Nein!). Somit eignet sich diese Methode gut, um zu prüfen, ob wir den Rand der Welt erreicht haben.

Konzept

Durch Aufrufen einer Methode mit dem **Rückgabetyp void** geben wir einen Befehl aus. Durch Aufrufen einer Methode mit einem **Rückgabetyp ungleich void** stellen wir eine Frage.

Übung 2.13 Erzeuge eine Krabbe. Klicke sie mit der rechten Maustaste an und suche die Methode **isAtEdge()** (sie verbirgt sich im Untermenü GEERBT VON ACTOR, da die Krabbe diese Methode von der Klasse **Actor** geerbt hat). Rufe diese Methode auf. Was liefert sie zurück?

Übung 2.14 Lass die Krabbe an den Rand des Bildschirms laufen (oder bewege sie selbst dorthin) und rufe dann erneut die Methode **isAtEdge()** auf. Was liefert sie jetzt zurück?

Wir können jetzt diese Methode mit einer *if-Anweisung* kombinieren, um den Code aus Listing 2.3 zu schreiben.

Listing 2.3: Wenden, wenn der Rand der Welt erreicht wird.

```
import greenfoot.*;

/**
 * Diese Klasse definiert eine Krabbe. Krabben leben am Strand.
 */
public class Crab extends Actor
{
    public void act()
    {
        if (isAtEdge() )
        {
            turn(17);
        }
        move(5);
    }
}
```

Konzept

Eine **if-Anweisung** eignet sich für Befehle, die nur ausgeführt werden sollen, wenn eine bestimmte Bedingung wahr ist.

Die **if**-Anweisung ist ein Konstrukt der Programmiersprache Java, das es dir ermöglicht, Befehle nur dann auszuführen, wenn eine bestimmte Bedingung erfüllt ist. So wollen wir in unserem Beispiel nur dann wenden, wenn unsere Krabbe den Rand der Welt erreicht hat. Dafür haben wir folgenden Code geschrieben:

```
if ( isAtEdge() )
{
    turn(17);
}
move(5);
```

Die allgemeine Form einer **if**-Anweisung sieht folgendermaßen aus:

```
if ( Bedingung )
{
    Befehl;
    Befehl;
    ...
}
```

Anstelle von *Bedingung* kann jeder beliebige Ausdruck stehen, der entweder (wie unser Methodenaufruf **isAtEdge()**) als wahr oder als falsch ausgewertet werden kann, und die *Befehle* werden nur dann ausgeführt, wenn die Bedingung wahr ist. Es ist möglich, mehr als einen Befehl anzugeben.

Ist die Bedingung falsch, werden die Befehle einfach übersprungen und die Ausführung fährt mit dem Code fort, der auf die schließende geschweifte Klammer der **if**-Anweisung folgt.

Beachte, dass der Aufruf unserer **move(5)**-Methode außerhalb der **if**-Anweisung erfolgt, sodass diese Methode auf alle Fälle ausgeführt wird. Mit anderen Worten: Wenn wir den Rand der Welt erreicht haben, wenden wir und bewegen uns. Wenn wir nicht am Rand der Welt sind, bewegen wir uns einfach.

Übung 2.15 Starte einen Versuch! Tippe den oben angegebenen Code in deinen Editor ein und schau, ob sich deine Krabben bei Erreichen des Weltrandes umdrehen. Achte besonders auf die öffnenden und schließenden Klammern – schnell hat man eine zu viel oder zu wenig eingetippt.

Übung 2.16 Gib für den Parameter der `turn`-Methode verschiedene Werte ein, bis dir das Ergebnis gefällt.

Übung 2.17 Platziere die `move(5)`-Anweisung in die `if`-Anweisung statt dahinter. Teste dies – welche Auswirkungen das hat? Erkläre das zu beobachtende Verhalten. (Anschließend behebe diesen Fehler und mache alles wieder rückgängig.)

Tipp!

Wenn du im Greenfoot-Editor den Cursor hinter einer öffnenden oder schließenden Klammer setzt, markiert Greenfoot das dazugehörige Gegenstück (d.h. die schließende bzw. öffnende Klammer). Damit kannst du prüfen, ob alle deine Klammernpaare wie erforderlich vollständig sind.

Randbemerkung: Farbige Sichtbarkeitsbereiche und Einrückung

Wenn du dir den Quelltext in Greenfoot oder in den Codebeispielen in diesem Buch anschaust (z.B. Listing 2.3), dann wirst du die farbigen Kästen bemerken, die für den Hintergrund verwendet werden. Diese werden *Gültigkeitsbereiche* (*scope*) genannt. Ein Gültigkeitsbereich ist die Ausdehnung eines gegebenen Java-Konstrukts. In Greenfoot wurden den unterschiedlichen Arten von Konstrukten verschiedene Farben zugewiesen: eine Klasse hat zum Beispiel einen grünen Hintergrund, ein Methode eine gelben und eine `if`-Anweisung wird auf einem violetten Grau dargestellt. Du siehst, dass diese Gültigkeitsbereiche *verschachtelt* sein können: eine `if`-Anweisung ist innerhalb einer Methode, eine Methode befindet sich in einer Klasse.

Wenn du diese farbigen Gültigkeitsbereiche beachtest, wird sich das im Laufe der Zeit auszahlen – du kannst damit einige häufig vorkommende Fehler vermeiden. Gültigkeitsbereiche sind in der Regel in deinem Code durch ein Paar geschweifter Klammern definiert (üblicherweise mit einer Kopfzeile oberhalb der öffnenden Klammer, in der angegeben wird, um welche Art von Gültigkeitsbereich es sich hier handelt). Es kann sehr leicht passieren, dass die Anzahl der geschweiften Klammern aus dem Gleichgewicht kommt – das heißt, es gibt mehr öffnende als schließende Klammern oder umgekehrt. Wenn dies passiert, wird sich dein Programm nicht übersetzen lassen.

Die Farben der Gültigkeitsbereiche helfen dir, ein solches Problem zu entdecken. Du wirst dich sehr schnell daran gewöhnen, wie die Gültigkeitsbereiche aussehen sollten, und dann wird dir auffallen, dass es irgendwie seltsam aussieht, wenn eine Klammer zu viel ist oder fehlt.

Hand in Hand mit der Farbe von Gültigkeitsbereichen geht das Prinzip der Einrückung.

In allen Codebeispielen, die du bisher gesehen hast (z.B. Listing 2.3), sind dir vielleicht die sorgfältigen Einrückungen aufgefallen. Jedes Mal nach einer geöffneten geschweiften Klammer wird der Code in den nachfolgenden Zeilen um eine Ebene eingerückt. Bei der schließenden geschweiften Klammer

wird die Einrückung wieder zurückgenommen, sodass die schließende geschweifte Klammer direkt unter der dazugehörigen öffnenden geschweiften Klammer steht. Dadurch ist es leichter, zu jeder Klammer das dazugehörige Gegenstück zu finden.

Wir verwenden für eine Einrückungsebene vier Leerzeichen. Du kannst diese Leerzeichen für eine Einrückungsebene auch auf einmal mit der Tabulatortaste einfügen. Greenfoot kann dir auch helfen, wenn deine Einrückungen zu unordentlich werden: Der Editor hat eine Funktion AUTO-LAYOUT im BEARBEITEN-Menü, das versucht, deine Einrückung für die ganze Klasse zu verbessern.

Es ist sehr wichtig, dass du in deinem Code auf die Einrückungen achtest. Wenn du deinen Code nicht sorgfältig einrückst, wird die Färbung der Sichtbarkeitsbereiche chaotisch und damit nutzlos, außerdem sind einige Fehler (vor allem falsch gesetzte oder fehlende geschweifte Klammern) nur schwer zu finden. Ordentliche Einrückungen tragen erheblich zur besseren Lesbarkeit des Codes bei und helfen so, Fehler zu vermeiden.

Übung 2.18 Öffne den Quelltext deiner **Crab**-Klasse. Entferne verschiedene öffnende oder schließende Klammern und beobachte die Veränderung bei den Farben der Gültigkeitsbereiche. Kannst du für jeden Fall die Veränderungen bei der Farbe erklären? Experimentiere auch damit, die Einrückungen der Klammern und anderem Code zu verändern, und beobachte, wie sich das auf die Optik auswirkt. Bringe zum Schluss die Klammern und Einrückungen wieder in Ordnung, sodass der Code wieder gut aussieht.

Zusammenfassung der Programmiertechniken

Unsere Diskussion der Programmierung in diesem Buch ist stark beispielorientiert. Allgemeine Programmiertechniken werden dann eingeführt, wenn wir sie zur Verbesserung unserer Szenarien benötigen. Von nun an fassen wir am Ende eines jeden Kapitels die wichtigen Programmiertechniken noch einmal zusammen. Damit betonen wir, was du dir wirklich merken musst, um gute Fortschritte machen zu können.

In diesem Kapitel haben wir gelernt, wie Methoden (z.B. `move(5)` oder `isAtEdge()`) mit und ohne Parameter aufgerufen werden. Dies wird die Basis aller weiteren Java-Programmierung bilden. Außerdem haben wir gelernt, wo sich der Rumpf der **act**-Methode verbirgt – denn an diesem Ort fügen wir unsere Befehle ein.

Außerdem hast du bereits die Bekanntschaft mit einigen Fehlermeldungen gemacht. Diese Meldungen werden dir im Laufe deiner Programmiererkarriere immer wieder begegnen. Wir alle machen Fehler und wir alle erhalten Fehlermeldungen. Das bedeutet jedoch nicht, dass du ein schlechter Programmierer bist – es gehört vielmehr zum Alltag eines Programmierers.

Wir haben einen kurzen Blick auf Vererbung geworfen: Klassen erben die Methoden ihrer Oberklassen. Die Dokumentation einer Klasse gibt dir einen Überblick, welche Methoden jeweils zur Verfügung stehen.

Und – was äußerst wichtig ist – wir haben gesehen, wie Entscheidungen gefällt werden: Wir haben eine `if`-Anweisung für eine bedingte Ausführung verwendet. Dies ging einher mit der Einführung des Typs **`boolean`** – ein Wert, der entweder **`true`** oder **`false`** sein kann.

Zusammenfassung der Konzepte

- Ein **Methodenaufruf** ist ein Befehl, der ein Objekt anweist, eine Aktion auszuführen. Die Aktion wird durch eine Methode des Objekts definiert.
- Einigen Methoden kannst du in Klammern zusätzliche Informationen übergeben. Der übergebene Wert wird **Parameter** genannt.
- Mehrere Befehle werden **nacheinander** ausgeführt, und zwar in der Reihenfolge, in der sie im Code erscheinen.
- Wenn eine Klasse übersetzt wird, prüft der Compiler, ob irgendwelche Fehler vorliegen. Wird ein solcher Fehler gefunden, wird eine **Fehlermeldung** angezeigt.
- Eine Unterklasse **erbt** alle Methoden ihrer Oberklasse. Das bedeutet, dass die Unterklasse alle von der Oberklasse definierten Methoden besitzt und nutzen kann.
- Durch Aufrufen einer Methode mit dem **Rückgabetyp `void`** geben wir einen Befehl aus. Durch Aufrufen einer Methode mit einem **Rückgabetyp ungleich `void`** stellen wir eine Frage.
- Eine **`if`-Anweisung** eignet sich für Befehle, die nur ausgeführt werden sollen, wenn eine bestimmte Bedingung wahr ist.

Vertiefende Aufgaben

Bei einigen Kapiteln gibt es am Ende einen Abschnitt mit dem Titel *Vertiefende Aufgaben*. An dieser Stelle werden keine neuen Inhalte vorgestellt, sondern es soll dir die Möglichkeit gegeben werden, ein wichtiges Konzept, das in diesem Kapitel eingeführt wurde, in einem anderen Zusammenhang einzuüben und dein Verständnis zu vertiefen.

Die zwei wichtigsten Konstrukte, denen wir in diesem Kapitel begegnet sind, sind Methodenaufrufe und **`if`**-Anweisungen. Hier gibt es einige weitere Übungen mit diesen beiden Konstrukten (Abbildung 2.4).

Abbildung 2.4
FatCat.

Methodensignaturen

Übung 2.19 Betrachte die folgenden Methodensignaturen:

```
public void play();
public void addAmount(int amount);
public boolean hasWings();
public void compare(int x, int y, int z);
public boolean isGreater (int number);
```

Beantworte für jede dieser Signaturen (schriftlich) die folgenden Fragen:

a. Wie lautet der Name der Methode?
b. Gibt die Methode einen Wert zurück? Falls ja, von welchem Typ ist der Rückgabewert?
c. Wie viele Parameter hat die Methode?

Übung 2.20 Schreibe eine Methodensignatur für eine Methode namens **`go`**. Die Methode hat keine Parameter und gibt keinen Wert zurück.

Übung 2.21 Schreibe eine Methodensignatur für eine Methode namens **`process`**. Die Methode hat einen Parameter vom Typ **`int`**, der **`number`** heißt, und gibt einen Wert vom Typ **`int`** zurück.

Übung 2.22 Schreibe eine Methodensignatur für eine Methode namens **`isOpen`**. Diese Methode hat keine Parameter und gibt einen Wert vom Typ **`boolean`** zurück.

Übung 2.23 Schreibe auf einem Blatt einen Methodenaufruf (Hinweis: dies ist ein Methodenaufruf, keine Signatur) für die **`play`**-Methode aus Übung 2.19. Schreibe einen weiteren Methodenaufruf für die Methode **`addAmount`** aus Übung 2.19. Und schreibe schließlich einen Methodenaufruf für die **`compare`**-Methode aus derselben Übung.

Die folgenden Übungen sollen in dem Greenfoot-Szenario *fatcat* implementiert werden. Öffne erst das Szenario in Greenfoot, bevor du weitermachst.

Lesen von Dokumentationen

Übung 2.24 Öffne den Editor für die Klasse **Cat**. Ändere die Sicht des Editors von QUELLTEXT auf DOKUMENTATION mithilfe des Auswahl-Listenfelds oben rechts im Editor-Fenster. Wie viele Methoden hat die Klasse **Cat**?

Übung 2.25 Wie viele Methoden von **Cat** haben einen Rückgabewert?

Übung 2.26 Wie viele Parameter hat die Methode **sleep**?

Methodenaufrufe schreiben (mit und ohne Parameter)

Übung 2.27 Versuche, einige deiner **Cat**-Methoden interaktiv mithilfe des Kontextmenüs von **Cat** aufzurufen. Die interessantesten Methoden sind alle „geerbt von **Cat**".

Übung 2.28 Ist der Katze langweilig (*bored*)? Was kannst du machen, damit ihr nicht mehr langweilig ist?

Übung 2.29 Öffne den Editor für die Klasse **MyCat**. (In diese Klasse soll der Code für die folgenden Übungen geschrieben werden.)

Übung 2.30 Lass die Katze etwas essen, wenn sie aktiv ist. (Das heißt, schreibe in der **act**-Methode einen Aufruf der **eat**-Methode.) Übersetze den Code. Teste den Code, indem du den ACT-Button in der Greenfoot-Steuerung anklickst.

Übung 2.31 Bringe die Katze zum Tanzen. (Mache das nicht interaktiv – schreibe dazu Code in der **act**-Methode. Klicke danach auf den ACT-Button in der Greenfoot-Steuerung.)

Übung 2.32 Lass die Katze schlafen.

Übung 2.33 Bringe deine Katze dazu, eine Folge von Tätigkeiten deiner Wahl auszuführen, die aus mehreren verfügbaren Aktionen hintereinander besteht.

If-Anweisungen

Übung 2.34 Verändere die **act**-Methode deiner Katze so, dass bei Anklicken von ACT Folgendes passiert: Falls die Katze müde ist, dann schläft sie ein bisschen, und wenn sie nicht müde ist, dann tut sie nichts.

Übung 2.35 Verändere die **act**-Methode deiner Katze so, dass sie tanzt, wenn ihr langweilig ist. (Aber nur, wenn ihr langweilig ist.)

Übung 2.36 Verändere die **act**-Methode deiner Katze so, dass sie isst, wenn sie hungrig ist.

Übung 2.37 Verändere die **act**-Methode deiner Katze folgendermaßen: Wenn die Katze müde ist, so schläft sie ein bisschen, danach ruft sie „Hurra". Wenn sie nicht müde ist, ruft sie einfach so „Hurra". (Zum Testen lass die Katze müde werden, indem einige Methoden interaktiv aufgerufen werden). Wie kannst du die Katze müde machen?

Übung 2.38 Schreibe Code in der **act**-Methode, der Folgendes macht: Wenn deine Katz alleine ist, lass sie schlafen; falls sie nicht alleine ist, ruft sie „Hurra". Teste deinen Code, indem du eine zweite Katze in der Welt platzierst, bevor du ACT klickst.