

## KAPITEL

# 1

## Greenfoot kennenlernen



### Lernziele

**Themen:** Greenfoot-Schnittstelle, Interaktion mit Objekten, Aufruf von Methoden, Ausführen eines Szenarios

**Konzepte:** Objekt, Klasse, Methodenaufruf, Parameter, Rückgabewert

Dieses Buch soll dir zeigen, wie du mithilfe der Entwicklungsumgebung Greenfoot Computerspiele und Simulationen entwickelst. Den Einstieg bildet das vorliegende Kapitel, in dem wir Greenfoot vorstellen und uns anhand einiger bereits bestehender Programme ansehen, was Greenfoot alles kann und wie man es verwendet.

Anschließend, wenn wir uns mit Greenfoot etwas näher vertraut gemacht haben, steigen wir sofort in das Schreiben eigener Spiele ein.

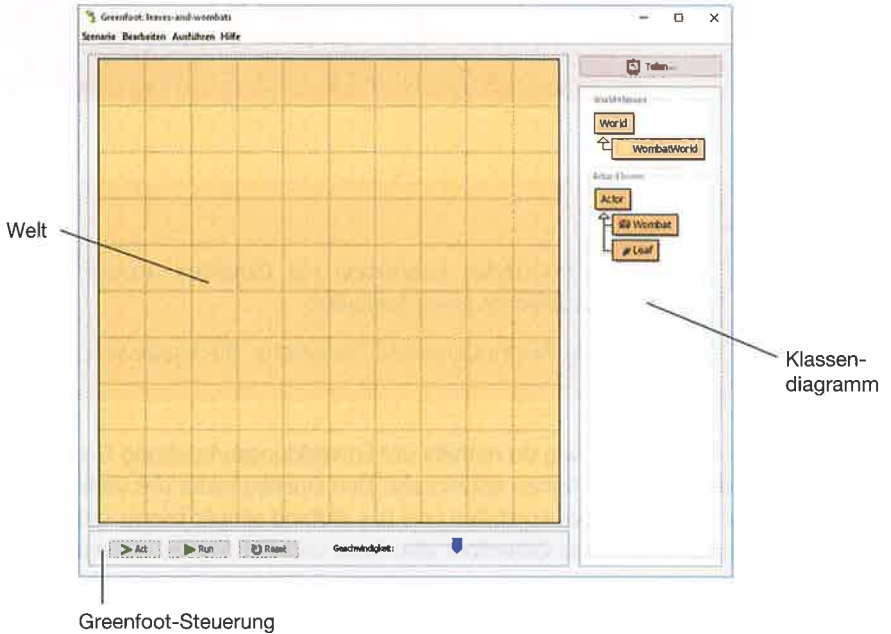
Am besten liest du dieses Kapitel (wenn nicht sogar das ganze Buch) direkt am Computer, wobei das Buch möglichst offen vor dir auf dem Schreibtisch liegen sollte, während Greenfoot auf dem Bildschirm angezeigt wird. Beim Lesen wirst du immer wieder aufgefordert, das gerade Erläuterte in Greenfoot nachzuvollziehen. Einige der Aufgaben kannst du überspringen, aber ein paar solltest du schon lösen, damit du in diesem Kapitel Fortschritte machst. Auf jeden Fall ist der Lerneffekt am höchsten, wenn du versuchst, alle Aufgaben zu erledigen.

Grundsätzlich gehen wir davon aus, dass du die Greenfoot-Software und die Buchszenarien bereits installiert hast (wie in **Anhang A** beschrieben). Wenn nicht, solltest du jetzt zuerst **Anhang A** lesen.

## 1.1 Die ersten Schritte

Starte Greenfoot und öffne das Szenario *leaves-and-wombats* aus dem Ordner *Greenfoot/Buch szenarien/Kapitel01*. Dazu wählst du *SZENARIO/ÖFFNEN* aus der Menüleiste aus.<sup>1</sup>

**Abbildung 1.1**  
Das Hauptfenster von Greenfoot.



Es erscheint das Hauptfenster von Greenfoot mit dem geöffneten Szenario, ähnlich wie in Abbildung 1.1.

Das Greenfoot-Fenster besteht im Wesentlichen aus drei Bereichen und einigen zusätzlichen Buttons. Diese drei Hauptbereiche sind:

- Die *Welt (world)*: Der größte Bereich wird Welt genannt. Dies ist der Bereich, in dem das Programm ausgeführt wird und in dem wir verfolgen können, was passiert. In diesem Szenario ist die Welt sandfarben und mit Gitterlinien durchzogen.
- Das *Klassendiagramm*: Der Bereich zur Rechten mit den hellbraunen Kästchen und den Pfeilen wird Klassendiagramm genannt. Was sich genau dahinter verbirgt, werden wir gleich ausführlich besprechen.
- Die *Greenfoot-Steuerung*: Die Buttons ACT, RUN und RESET und der Schieberegler für die Geschwindigkeit ganz unten dienen der Programmsteuerung. Auch darauf werden wir später noch zu sprechen kommen.

<sup>1</sup> Mit dieser Notation zeigen wir dir an, dass du Befehle aus einem Menü auswählen sollst. *SZENARIO/ÖFFNEN* steht also für den Befehl *ÖFFNEN* aus dem Menü *SZENARIO*.

## 1.2 Objekte und Klassen

Zuerst wollen wir uns dem Klassendiagramm zuwenden. In dem Klassendiagramm findest du alle Klassen, die für dieses Szenario definiert wurden. In unserem Fall lauten sie **World**, **WombatWorld**, **Actor**, **Wombat** und **Leaf**.

Für unsere Projekte werden wir die Programmiersprache Java verwenden. Java ist eine *objektorientierte* Sprache. Für die objektorientierte Programmierung sind die Konzepte von Klassen und Objekten von grundlegender Bedeutung.

Betrachten wir einmal die Klasse **Wombat**. Die Klasse **Wombat** steht für das allgemeine Konzept eines Wombats – sie beschreibt sozusagen alle Wombats. Sobald wir eine Klasse in Greenfoot angelegt haben, können wir davon *Objekte* erzeugen. (Objekte werden in der Programmierung oft als *Instanzen* bezeichnet – diese beiden Begriffe bedeuten das Gleiche.)

Ein Wombat ist übrigens ein australisches Beuteltier (Abbildung 1.2). Wenn du mehr über Wombats wissen möchtest, recherchiere ein wenig im Internet – dort wirst du eine Fülle an Informationen finden.

Wenn du mit der rechten Maustaste<sup>2</sup> auf die Klasse **Wombat** klickst, springt ein Kontextmenü zu dieser Klasse auf (Abbildung 1.3a). Die erste Option in diesem Menü (**new Wombat()**) erzeugt ein neues **Wombat**-Objekt. Probiere es einfach aus!

Es wird das Bild eines kleinen **Wombat**-Objekts eingeblendet, das du mit deiner Maus auf dem Bildschirm verschieben kannst (Abbildung 1.3b). Platziere den Wombat irgendwo in der Welt, indem du an eine beliebige Position klickst (Abbildung 1.3c).



### Konzept

Greenfoot-Szenarien bestehen aus einem Satz an **Klassen**.

### Konzept

Um zu betonen, dass Objekte immer konkrete Exemplare aus einer bestimmten Klasse sind, wird auch der Begriff *Instanz* benutzt. Objekt und Instanz bedeuten das Gleiche.

**Abbildung 1.2**  
Ein Wombat.<sup>3</sup>

<sup>2</sup> Wenn deine Maus, z.B. bei Mac OS-Rechnern, nur eine Taste hat, halte gleichzeitig mit der Maustaste die **Ctrl**-Taste gedrückt.

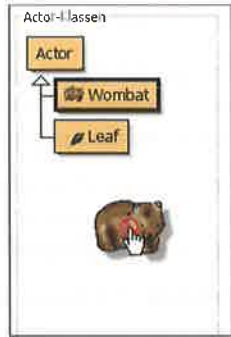
<sup>3</sup> Bildquelle: Marco Tomasini/Fotolia.

**Abbildung 1.3**

- a) Das Klassenmenü.
- b) Ein Objekt ziehen.
- c) Das Objekt platzieren.



a)



b)



c)

### Konzept

Von einer **Klasse** können viele **Objekte** erzeugt werden.

Sobald du in Greenfoot eine Klasse erstellt hast, kannst du davon beliebig viele Objekte erzeugen.

**Übung 1.1** Erzeuge weitere Wombats in der Welt. Erzeuge einige Blätter (*leaf*).

Im Moment interessieren uns nur die Klassen **Wombat** und **Leaf**. Die anderen Klassen werden wir weiter hinten besprechen.

## 1.3 Mit Objekten interagieren

Nachdem wir einige Objekte in der Welt platziert haben, können wir mit ihnen interagieren. Dazu müssen wir die Objekte mit der rechten Maustaste anklicken, um das *Objektmenü* (Abbildung 1.4) aufzurufen. In dem Objektmenü findest du alle Operationen, die mit diesem spezifischen Objekt ausgeführt werden können. So zeigt uns das Objektmenü von **wombat**, was dieser spezielle Wombat alles machen kann (sowie zwei weitere Funktionen, **INSPIZIEREN** und **ENTFERNEN**, auf die wir erst später eingehen wollen).

### Konzept

Objekte haben **Methoden**. Das Aufrufen dieser Methoden führt eine Aktion aus.

In Java werden diese Operationen *Methoden* genannt. Es kann nicht schaden, sich von Anfang an an die Standardterminologie zu gewöhnen, sodass auch wir ab jetzt von *Methoden* sprechen werden. Wir können eine Methode *aufrufen*, indem wir sie im Menü auswählen.

**Übung 1.2** Rufe für einen Wombat die Methode **move()** auf. Welche Operation ist damit verbunden? Wiederhole diesen Schritt mehrmals. Rufe dann die Methode **turnLeft()** auf. Platziere zwei Wombats in deiner Welt und Sorge dafür, dass sie sich anschauen.



**Abbildung 1.4**  
Das Objektmenü  
des Wombats.

Kurz gesagt, können wir also Dinge in Gang setzen, indem wir Objekte von einer der vorgegebenen Klassen erzeugen, und wir können diesen Objekten Befehle erteilen, indem wir deren Methoden aufrufen.

Lass uns das Objektmenü etwas genauer betrachten. Die Methoden **move** und **turnLeft** lauten vollständig:

```
void move()
void turnLeft()
```

Wie du siehst, wird im Menü nicht nur der Methodenname angegeben. Vor dem Namen steht noch das Wort **void** und an den Namen angehängt steht ein Klammernpaar. Diese beiden kryptischen Informationen verraten uns, welche Daten in den Methodenaufruf hineingereicht und welche Daten zurückgeliefert werden.

## 1.4 Rückgabetypen

Das Wort am Anfang wird auch als *Rückgabetyp* bezeichnet. Es teilt uns mit, was die Methode zurückliefert, wenn wir sie aufrufen. Das Wort **void** bedeutet in diesem Falle „nichts“: Methoden, deren Rückgabetyp **void** lautet, liefern keine Informationen zurück. Sie führen nur ihre Aktion aus und enden dann.

Wird anstelle von **void** irgendetwas anderes angegeben, verrät uns dies, dass die Methode bei Aufruf einen Wert zurückliefert, ja sogar von welchem Typ dieser Wert ist. So finden wir beispielsweise in dem Menü des Wombats (Abbildung 1.4) noch die Angaben **int** und **boolean** als Rückgabetypen. Die Angabe **int** ist eine Kurzform für „Integer“ und bezeichnet ganze Zahlen (d.h. Zahlen ohne Dezimalpunkt). Beispiele für Integer sind 3, 42, -3 und 12000000.

### Konzept

Der **Rückgabetyp** einer Methode gibt an, was diese Methode bei Aufruf zurückliefert.

**Konzept**

Eine Methode mit dem Rückgabotyp **void** liefert keinen Wert zurück.

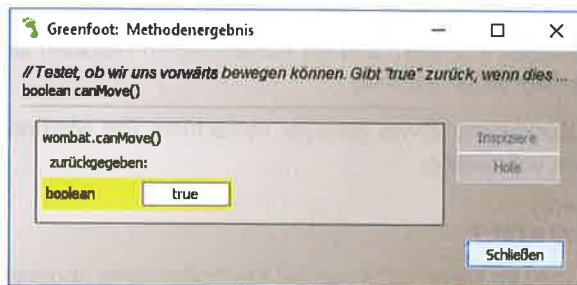
Der Rückgabotyp **boolean** kennt nur zwei mögliche Werte: **true** (wahr) und **false** (falsch). Eine Methode mit einem booleschen Rückgabotyp liefert entweder den Wert **true** oder den Wert **false** zurück.

Methoden mit dem Rückgabotyp **void** sind für unseren Wombat quasi Befehle. Wenn wir die Methode **turnLeft** aufrufen, gehorcht der Wombat und wendet sich nach links. Methoden, deren Rückgabotyp nicht **void** lautet, sind eher als Fragen zu verstehen. Betrachten wir dazu die Methode **canMove**:

**boolean canMove()**

Wenn wir diese Methode aufrufen, erhalten wir als Ergebnis das Dialogfeld aus Abbildung 1.5. Die Information, auf die es hier ankommt, ist das Wort **true**, das von diesem Methodenaufruf zurückgeliefert wurde. Genau genommen haben wir den Wombat gerade gefragt, ob er sich bewegen kann, und der Wombat hat „Ja!“ (**true**) geantwortet.

**Abbildung 1.5**  
Ein Methodenergebnis.



**Übung 1.3** Rufe die Methode **canMove()** für deinen Wombat auf. Liefert sie immer **true** zurück? Oder gibt es auch Situationen, in denen sie **false** zurückliefert?

Versuche, eine andere Methode aufzurufen, die ebenfalls einen Wert zurückliefert:

**int getLeavesEaten()**

Mithilfe dieser Methode erhalten wir die Information, wie viele Blätter unser Wombat gefressen hat.

**Übung 1.4** Für einen neu erzeugten Wombat liefert die Methode **getLeavesEaten()** immer null zurück. Kannst du eine Situation schaffen, in der das Ergebnis dieser Methode nicht null ist? (Mit anderen Worten: Kannst du deinen Wombat veranlassen, einige Blätter zu fressen?)

Methoden, deren Rückgabebetyp nicht **void** ist, teilen uns in der Regel etwas über das Objekt mit (*Kann es sich bewegen? Wie viele Blätter hat es gefressen?*), aber ändern das Objekt nicht. Methoden, deren Rückgabebetyp **void** lautet, sind normalerweise Befehle an das Objekt, die bewirken, dass das Objekt etwas macht.

## 1.5 Parameter

Eine weitere Besonderheit des Methodenmenüs, auf die wir noch nicht eingegangen sind, sind die Klammern nach dem Methodennamen.

Rückgabewert

Parameter

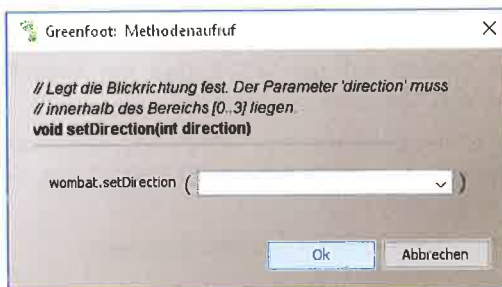
```
int getLeavesEaten()
void setDirection(int direction)
```

In den Klammern, die auf den Methodennamen folgen, steht die *Parameterliste*. Diese teilt uns mit, ob die Methode noch weitere Informationen benötigt, um ausgeführt zu werden, und wenn ja, welcher Art diese Informationen sind.

Wenn wir es nur mit einem leeren Paar Klammern zu tun haben (wie es bei allen bisherigen Methoden der Fall war), dann hat die Methode eine *leere Parameterliste*. Mit anderen Worten, die Methode erwartet keine Parameter – wenn wir die Methode aufrufen, wird sie einfach ausgeführt. Steht irgendetwas in diesen Klammern, dann erwartet die Methode einen oder mehrere Parameter, d.h. zusätzliche Informationen, die wir bereitstellen müssen.

Nehmen wir z.B. die Methode **setDirection**. Wie wir sehen, enthält die Parameterliste die Angabe **int direction**. Wenn wir die Methode aufrufen, erscheint ein Dialogfeld ähnlich dem in Abbildung 1.6.

Die Angabe **int direction** verrät uns, dass diese Methode einen Parameter vom Typ **int** erwartet, der eine *Richtung* angibt. Ein Parameter ist eine zusätzliche Information, die wir dieser Methode mitgeben müssen, damit sie sich ausführen lässt. Jeder Parameter besteht aus zwei Teilen: zum einen dem Parametertyp (hier: **int**) und zum anderen dem Namen, der uns einen Hinweis darauf gibt, wofür dieser Parameter verwendet wird. Wenn eine Methode einen Parameter besitzt, müssen wir diese zusätzliche Information mitliefern, wenn wir die Methode aufrufen.



In diesem Fall verrät uns der Typ **int**, dass wir eine ganze Zahl übergeben sollen, und der Name lässt vermuten, dass diese Zahl irgendetwas mit der Richtung zu tun hat, in die gedreht werden soll.

### Konzept

Methoden mit dem Rückgabebetyp **void** stellen **Befehle** dar, Methoden mit einem Rückgabebetyp ungleich **void** **Fragen**.

### Konzept

Ein **Parameter** ist ein Mechanismus, um einer Methode zusätzliche Daten zu übergeben.

### Konzept

Parameter und Rückgabewerte haben **Typen**. Beispiele für Typen sind **int** für Zahlen und **boolean** für wahr/falsch-Werte.

**Abbildung 1.6**

Das Dialogfeld für den Methodenaufruf.

Oben im Dialogfeld steht ein Kommentar, der uns genauere Auskunft über die Eingabe gibt: der **direction**-Parameter sollte zwischen 0 und 3 liegen.

**Übung 1.5** Rufe die Methode **setDirection(int direction)** auf. Gib eine Zahl für den Parameterwert ein und beobachte, was passiert. Welche Zahl entspricht welcher Richtung? Halte deine Beobachtungen schriftlich fest. Was passiert, wenn du eine Zahl größer 3 eingibst? Was passiert, wenn deine Eingabe keine ganze Zahl ist (z.B. eine Dezimalzahl 2.5)<sup>4</sup> oder ein Wort (drei)?

### Konzept

Die Spezifikation einer Methode, die Auskunft über ihren Rückgabetyt, Namen und ihre Parameter gibt, wird **Signatur** genannt.

Die Methode **setDirection** erwartet nur einen einzigen Parameter. Später werden wir es mit Methoden zu tun haben, die mehr als einen Parameter verlangen. In einem solchen Fall wird die Methode alle Parameter, die sie erwartet, innerhalb der Klammern auflisten.


Die formale Beschreibung einer Methode, wie sie im Objektmenü zu sehen ist (also Rückgabetyt, Methodename und Parameterliste), bezeichnen wir als *Methodensignatur*.

Damit haben wir jetzt einen Punkt erreicht, an dem du die wichtigsten Interaktionen mit Greenfoot-Objekten ausführen kannst. Du kannst Objekte von Klassen erzeugen, die Methodensignaturen interpretieren und Methoden (mit und ohne Parameter) aufrufen.

## 1.6 Die Ausführung in Greenfoot

Es gibt noch eine weitere Möglichkeit, mit Greenfoot-Objekten zu interagieren: Die Greenfoot-Steuerung.

### Tipp!

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

**Übung 1.6** Platziere einen Wombat und eine Reihe von Blättern in deiner Welt und rufe dann mehrmals die Wombat-Methode **act()** auf. Was macht diese Methode? Inwiefern unterscheidet sie sich von der **move**-Methode? Achte darauf, dass du verschiedene Situationen ausprobierst, z.B. wenn der Wombat das Ende der Welt erreicht hat oder auf einem Blatt sitzt.

**Übung 1.7** Platziere noch einmal einen Wombat und einige Blätter in der Welt und klicke diesmal auf den ACT-Button der Greenfoot-Steuerung unten im Fenster. Was kannst du beobachten?

**Übung 1.8** Was ist der Unterschied zwischen dem Anklicken des ACT-Buttons und dem Aufrufen der **act()**-Methode? (Versuche es einmal mit mehreren Wombats in der Welt.)

**Übung 1.9** Klicke auf den RUN-Button. Was geschieht?

<sup>4</sup> Java, wie auch die meisten anderen Programmiersprachen, verwenden für Dezimalzahlen die angloamerikanische Notation. Um Verwirrung zu vermeiden, benutzen wir daher auch im Text den Dezimalpunkt.



Die **act**-Methode gehört zu den wichtigsten Methoden der Greenfoot-Objekte. Wir werden auch in den nachfolgenden Kapiteln immer wieder mit ihr zu tun haben. Alle Objekte in einer Greenfoot-Welt verfügen über eine **act**-Methode. Durch den Aufruf von **act** instruieren wir das Objekt mehr oder weniger, „das zu machen, was ein solches Objekt in diesem Moment so macht“. Wenn du die Wombat-Methode **act** für unseren Wombat ausprobiert hast, wirst du festgestellt haben, dass sie in etwa Folgendes macht:

- Wenn wir auf einem Blatt sitzen, fressen wir das Blatt.
- Andernfalls, sofern wir uns vorwärts bewegen können, bewegen wir uns vorwärts.
- Andernfalls wenden wir uns nach links.

Wenn du die Übungen oben gemacht hast, solltest du festgestellt haben, dass der ACT-Button der Greenfoot-Steuerung einfach die **act**-Methode der Akteure in der Welt aufruft. Der einzige Unterschied zum Aufruf der Methode über das Objektmenü ist, dass der ACT-Button die **act**-Methode auf alle Objekte in der Welt aufruft, während der Weg über das Objektmenü nur das ausgewählte Objekt betrifft.

Der Button RUN ruft die Methode **act** einfach wieder und wieder auf, bis du auf PAUSE drückst.

Nun wollen wir das, was wir bisher besprochen haben, an einem weiteren Szenario ausprobieren.

### Konzept

Objekte, die in der Welt abgelegt werden können, werden auch als **Akteure** (*actors*) bezeichnet.

## 1.7 Ein zweites Beispiel

Hierzu öffnen wir ein neues Szenario namens *asteroids1* aus dem Ordner *Kapitel01* der Buchszenarien. Es sollte in etwa aussehen wie in Abbildung 1.7 (außer dass noch keine Rakete oder Asteroiden auf dem Bildschirm zu sehen sind).

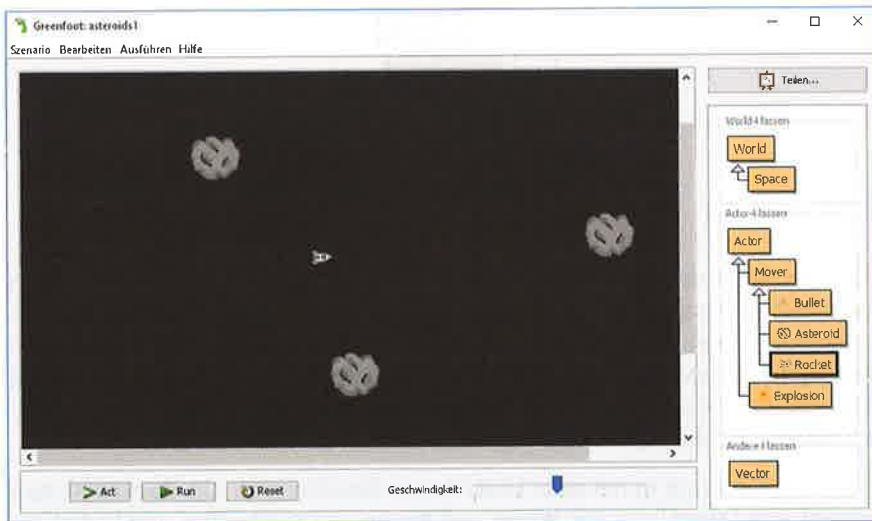


Abbildung 1.7  
Das Szenario  
*asteroids1*.

## 1.8 Das Klassendiagramm verstehen

Bevor wir weitermachen, wollen wir noch einen gründlicheren Blick auf das Klassendiagramm werfen (Abbildung 1.8). Ganz oben stehen die beiden Klassen **World** und **Space**, die durch einen Pfeil verbunden sind.

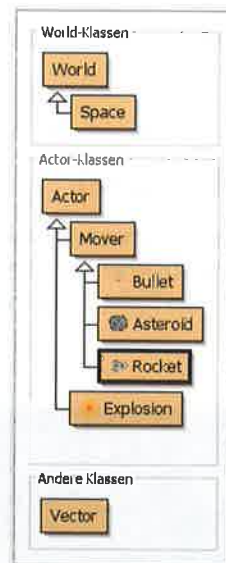
Die Klasse **World** gibt es in allen Greenfoot-Szenarien – sie ist sozusagen in Greenfoot „eingebaut“. Die Klasse darunter, **Space**, repräsentiert die spezielle Welt für dieses bestimmte Szenario. Ihr Name kann bei jedem Szenario anders lauten, aber jedes Szenario hat an dieser Stelle seine eigene spezielle Welt.

Der Pfeil steht für eine *Ist-eine*-Beziehung: **Space ist eine World** (im Sinne der Greenfoot-Welten ist **Space** hier eine spezielle Greenfoot-Welt). Wir sprechen auch davon, dass **Space** eine *Unterklasse* von **World** ist.

Wir müssen in der Regel keine Objekte von Welt-Klassen erzeugen – dies erledigt Greenfoot für uns. Wenn wir ein Szenario öffnen, erzeugt Greenfoot automatisch ein Objekt der **World**-Unterklasse. Das Objekt wird dann im Hauptteil des Bildschirms angezeigt. (Der große schwarze Weltraum ist beispielsweise ein Objekt der Klasse **Space**.)

### Konzept

Eine **Unterklasse** ist eine Klasse, die eine Spezialisierung einer anderen Klasse ist. In Greenfoot wird dies durch einen Pfeil im Klassendiagramm symbolisiert.



**Abbildung 1.8**  
Ein Klassendiagramm.

Darunter siehst du eine weitere Gruppe von sechs Klassen, die ebenfalls durch Pfeile verbunden sind. Jede Klasse repräsentiert dabei ihre eigenen Objekte. Von unten nach oben gelesen stoßen wir zuerst auf die Klassen für die Asteroiden (**Asteroid**), Raketen (**Rocket**) und Geschosse (**Bullet**), deren Objekte sich alle viel bewegen (*move*) und die deshalb auch von der Klasse **Mover** abgeleitet wurden. Die **Mover**- und die **Explosion**-Objekte wiederum sind Akteure (*actor*).

Auch hier haben wir es mit einer Unterklassen-Beziehung zu tun. **Rocket** ist zum Beispiel eine Unterklasse von **Mover** und **Mover** und **Explosion** sind Unterklas-


sen von **Actor**. (Wir könnten aber genauso gut sagen, dass **Mover** eine Oberklasse von **Rocket** ist und **Actor** eine Oberklasse von **Explosion**.)

Unterklassen-Beziehungen können sich über mehrere Ebenen erstrecken: **Rocket** ist zum Beispiel eine Unterklasse von **Actor** (weil es eine Unterklasse von **Mover** ist, die eine Unterklasse von **Actor** ist). Später werden wir noch einmal auf die Bedeutung von Unter- und Oberklassen eingehen.

Ganz unten im Diagramm im Bereich *Andere Klasse* findest du die Klasse **Vector**. Hierbei handelt es sich um eine Hilfsklasse, die von den anderen Klassen verwendet wird. Von ihr können wir keine Objekte in der Welt ablegen.

## 1.9 Mit Asteroiden spielen

Um mit diesem Szenario zu spielen, müssen wir zuerst einige Akteur-Objekte (Objekte der Unterklassen von **Actor**) erzeugen und in der Welt ablegen. In diesem Fall erzeugen wir nur Objekte von Klassen, die keine weiteren Unterklassen aufweisen: **Rocket**, **Bullet**, **Asteroid** und **Explosion**.

Lass uns also zunächst einmal eine Rakete und zwei Asteroiden im Weltraum ablegen. (Denke daran: Du kannst Objekte durch Klicken mit der rechten Maustaste erzeugen oder durch Auswählen der Klasse und Klick mit gedrückter -Taste in die Welt.)

Wenn du deine Objekte platziert hast, klicke auf den RUN-Button. Anschließend kannst du das Raumschiff mit den Pfeiltasten auf deiner Tastatur steuern und mit der Leertaste Schüsse abfeuern. Versuche, die Asteroiden aus dem Weg zu räumen, bevor du mit ihnen kollidierst.

**Übung 1.10** Wenn du dieses Spiel eine Weile gespielt hast, dürfte dir nicht entgangen sein, dass deine Schüsse erst mit einiger Verzögerung abgefeuert werden. Um dieses Manko zu beheben, wollen wir unseren Code zum Feuern aus dem Raumschiff ein wenig optimieren. (Dann hast du eine bessere Chance, die Asteroiden aus dem Weg zu räumen!) Platziere eine Rakete in der Welt, rufe dann (über das Objektmenü) die dazugehörige Methode **setGunReloadTime** auf und setze die Zeit zum erneuten Laden auf 5. Spiele dann noch einmal (mit mindestens zwei Asteroiden), um die Änderungen auszuprobieren.

**Übung 1.11** Nachdem es dir gelungen ist, alle Asteroiden zu zerstören, (oder an einem anderen beliebigen Zeitpunkt im Spiel), halte die Ausführung an (durch Drücken von PAUSE) und finde heraus, wie viele Schüsse du abgegeben hast. Auch hierzu steht dir eine Methode aus dem Objektmenü der Rakete zur Verfügung. (Versuche einmal, zwei Asteroiden mit so wenigen Schüssen wie möglich zu zerstören.)

**Übung 1.12** Sicher ist dir aufgefallen, dass die Rakete sich ein wenig bewegt, wenn du sie in der Welt ablegst. Wie hoch ist ihre Anfangsgeschwindigkeit?

**Übung 1.13** Asteroiden haben eine inhärente *Stabilität*. Jedes Mal, wenn sie von einer Kugel getroffen werden, nimmt ihre Stabilität ab. Wenn die Stabilität null erreicht, sind die Asteroiden zerstört. Wie hoch ist der Stabilitätswert der Asteroiden bei ihrer Erzeugung? Um wie viel nimmt die Stabilität ab, wenn der Asteroid durch eine Kugel getroffen wird? (Tipp: Schieße auf einen Asteroiden nur einmal und prüfe dann erneut seine Stabilität. Ein weiterer Tipp: Um auf den Asteroiden zu schießen, musst du das Spiel mit RUN ausführen. Um das Objektmenü aufzurufen, musst du das Spiel zuerst mit PAUSE anhalten.)

**Übung 1.14** Erstelle einen sehr großen Asteroiden.

## 1.10 Quelltext

### Konzept

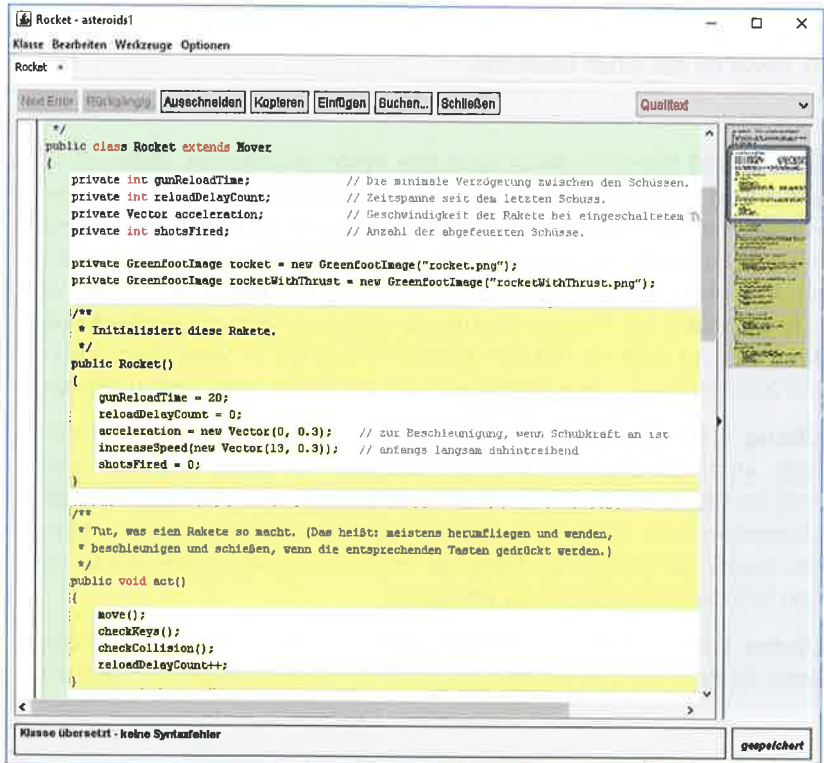
Jede Klasse wird durch **Quelltext** definiert. Dieser Code definiert, was die Objekte dieser Klasse machen können. Wir können den Quelltext einsehen, indem wir den Editor für die Klasse öffnen.

### Tipp!

Du kannst eine Klasse im Editor öffnen, indem du die Klasse im Klassendiagramm doppelt anklickst.

Das Verhalten eines jeden Objekts wird durch seine Klasse definiert. Wir können dieses Verhalten vorgeben, indem wir *Quelltext* schreiben. Dazu benutzen wir die Programmiersprache Java. Der Quelltext einer Klasse ist der Code, der alle Details über die Klasse und ihre Objekte enthält. Um den Quelltext einzusehen, müssen wir aus dem Kontextmenü der Klasse den Befehl EDITOR ÖFFNEN wählen, der ein Editorfenster einblendet (Abbildung 1.9).

Im Moment ist es nur wichtig zu wissen, dass wir auf das Verhalten der Objekte Einfluss nehmen können, indem wir den Quelltext der Klasse ändern. Das wollen wir im Folgenden versuchen.



**Abbildung 1.9**  
Die Klasse **Rocket**  
im Editorfenster.

Bereits zuvor haben wir gesehen, dass die Feuergeschwindigkeit der Rakete ziemlich langsam war. Wir könnten das für jede Rakete einzeln ändern, indem wir auf jeder neuen Rakete eine Methode ausführen; doch müssten wir diesen Schritt bei jedem neuen Spiel wiederholen. Wir können stattdessen aber auch den Code der Rakete ändern, sodass ihre Anfangsfeuergeschwindigkeit (sagen wir auf 5) geändert wird. Anschließend starten alle Raketen mit diesem verbesserten Verhalten.

Öffne den Editor für die Klasse **Rocket**. Ungefähr in der 25. Zeilen von oben solltest du eine Zeile mit folgendem Wortlaut finden:

```
gunReloadTime = 20;
```

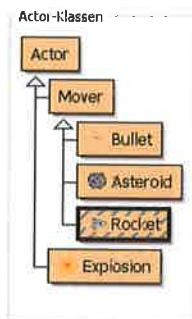
Genau in dieser Zeile wird die Zeit für das erneute Laden zum Schießen gesetzt. Ändere diese Zeile in

```
gunReloadTime = 5;
```

Achte darauf, nichts sonst zu ändern. Du wirst bald merken, dass Programmiersysteme ziemlich pingelig sind. Bereits ein falsches oder fehlendes Zeichen kann zu Fehlern führen. Wenn du zum Beispiel das Semikolon am Ende der Zeile entfernst, wirst du schnell einen Fehler ernten.

Schließe jetzt das Editorfenster (unsere Änderungen sind abgeschlossen) und betrachte noch einmal das Klassendiagramm. Es hat sich verändert: die **Rocket**-Klasse erscheint nun schraffiert (Abbildung 1.10). Die Schraffierung ist ein Hinweis darauf, dass die Klasse bearbeitet wurde und jetzt neu *übersetzt* (oder fachsprachlich: *kompiliert*) werden muss.<sup>5</sup> Die Kompilierung ist ein Übersetzungsprozess: Der Quelltext der Klasse wird in Maschinencode übersetzt, den dein Computer ausführen kann.

Die Klassen müssen nach jeder Änderung am Quelltext kompiliert werden, bevor neue Objekte dieser Klasse erzeugt werden können. (Manchmal müssen mehrere Klassen neu kompiliert werden, auch wenn wir nur eine Klasse geändert haben. Dies könnte der Fall sein, wenn Klassen voneinander abhängen. Wenn sich eine Klasse ändert, müssen eventuell mehrere neu übersetzt werden.)



### Konzept

Computer verstehen keinen Quelltext. Er muss erst in Maschinencode übersetzt werden, bevor er ausgeführt werden kann. Diesen Vorgang nennt man **Übersetzung** oder **Kompilierung**.

**Abbildung 1.10**

Die Klassen nach der Bearbeitung.

Du kannst die Klassen übersetzen, indem du auf den Button ALLE ÜBERSETZEN unten rechts im Hauptfenster von Greenfoot klickst.<sup>6</sup> Sobald die Klassen übersetzt sind, verschwinden die Streifen und wir können wieder Objekte erstellen.

<sup>5</sup> In der Greenfoot-Version 3.0 muss der Quelltext nicht mehr eigens kompiliert werden, dies geschieht automatisch bei der Speicherung.

<sup>6</sup> Diesen Button gibt es in der Version 3.0 nicht mehr, siehe oben.

**Übung 1.15** Ändere den Quelltext der Klasse **Rocket** wie oben beschrieben. Schließe den Editor und übersetze die Klassen. Starte jetzt einen neuen Versuch: Die Raketen sollten jetzt gleich von Beginn an viel schneller feuern.

In **Kapitel 9** werden wir noch einmal zu diesem Asteroiden-Spiel zurückkehren und uns anschauen, wie der Quelltext für dieses Spiel geschrieben wird.

## Zusammenfassung der Programmier Techniken



In diesem Kapitel haben wir gesehen, wie Greenfoot-Szenarien aussehen können und wie wir mit ihnen interagieren. Wir haben gelernt, wie wir Objekte erzeugen und wie wir mit diesen Objekten durch Aufruf ihrer Methoden kommunizieren. Einige Methoden waren Befehle an das Objekt, andere wiederum lieferten Informationen über das Objekt zurück. Parameter dienen dazu, zusätzliche Informationen in die Methoden hineinzureichen, während Rückgabewerte Informationen zurück an den Aufrufer liefern.

Von den Klassen wurden Objekte erzeugt. Der Quelltext steuert die Definition der Klasse (und darüber das Verhalten und die Merkmale aller Objekte dieser Klasse). Mithilfe eines Editors können wir den Quelltext ändern. Nach der Bearbeitung des Quelltexts müssen die Klassen neu übersetzt werden.

Im Rest des Buches werden wir größtenteils darüber sprechen, wie du in Java Quelltext für interessante, attraktive und kurzweilige Szenarien schreibst.

## Zusammenfassung der Konzepte



- Greenfoot-Szenarien bestehen aus einem Satz an **Klassen**.
- Um zu betonen, dass Objekte immer konkrete Exemplare aus einer bestimmten Klasse sind, wird auch der Begriff **Instanz** benutzt. Objekt und Instanz bedeuten das Gleiche.
- Von einer **Klasse** können viele **Objekte** erzeugt werden.
- Objekte haben **Methoden**. Das Aufrufen dieser Methoden führt eine Aktion aus.
- Der **Rückgabotyp** einer Methode gibt an, was diese Methode bei Aufruf zurückliefert.
- Eine Methode mit dem Rückgabotyp **void** liefert keinen Wert zurück.
- Methoden mit dem Rückgabotyp **void** stellen **Befehle** dar, Methoden mit einem Rückgabotyp ungleich **void** **Fragen**.
- Ein **Parameter** ist ein Mechanismus, um einer Methode zusätzliche Daten zu übergeben.
- Parameter und Rückgabewerte haben **Typen**. Beispiele für Typen sind **int** für Zahlen und **boolean** für wahr/falsch-Werte.
- Die Spezifikation einer Methode, die Auskunft über ihren Rückgabotyp, Namen und ihre Parameter gibt, wird **Signatur** genannt.
- Objekte, die in der Welt abgelegt werden können, werden als **Akteure** (*actor*) bezeichnet.
- Eine **Unterklasse** ist eine Klasse, die eine Spezialisierung einer anderen Klasse ist. In Greenfoot wird dies durch einen Pfeil im Klassendiagramm symbolisiert.
- Jede Klasse wird durch **Quelltext** definiert. Dieser Code definiert, was die Objekte dieser Klasse machen können. Wir können den Quelltext einsehen, indem wir den Editor für die Klasse öffnen.
- Computer verstehen keinen Quelltext. Er muss erst in Maschinencode übersetzt werden, bevor er ausgeführt werden kann. Diesen Vorgang nennt man **Übersetzung** oder **Kompilierung**.

