KAPITEL

8 Interagierende Objekte: Newtons Labor

Lernziele

Themen: Mehr über interagierende Objekte, Verwendung von Hilfsklassen

Konzepte: Überladen, mehr Praxis mit Listen und der **for-each**-Schleife

In diesem Kapitel wollen wir die etwas anspruchsvolleren Interaktionen zwischen Objekten in der Welt untersuchen. Als Einstieg beschäftigen wir uns mit einer der universalsten Interaktionen zwischen Objekten überhaupt: der Gravitationskraft.

In diesem Szenario haben wir es mit Himmelskörpern (wie Sternen und Planeten) zu tun. Wir werden die Bewegung dieser Körper durch den Weltraum unter Zugrundelegung des Newton'schen Gravitationsgesetzes simulieren. (Wir wissen zwar inzwischen, dass die Formeln von Newton nicht ganz stimmen und dass Einsteins allgemeine Relativitätstheorie die Bewegungen der Planeten präziser beschreibt, aber für unsere einfache Simulation soll uns Newtons Gesetz reichen.

Solltest du dir Gedanken machen, dass deine Physik- und Mathematikkenntnisse nicht ausreichen, sei versichert, dass wir nicht allzu tief in die Materie einsteigen. Und die Formel, die wir verwenden, ist eigentlich recht einfach. Am Ende werden wir dieses Szenario in ein künstlerisches Experiment mit audio-visuellen Effekten umwandeln. Wenn du vornehmlich technisch interessiert bist, kannst du dich verstärkt auf den physikalischen Aspekt des Szenarios konzentrieren. Sollte dein Interesse dagegen mehr auf dem künstlerischen Gebiet liegen, steht es dir frei, dich stattdessen mehr mit diesem Aspekt zu beschäftigen.

8.1 Der Ausgangspunkt: Newtons Labor

Wir wollen dieses Projekt damit beginnen, dass wir eine teilweise implementierte Version dieses Szenarios untersuchen. Öffne das Szenario *Newtons-Lab-1* aus dem Ordner deiner Buchszenarien. Du wirst feststellen, dass es zu der Klasse **World** bereits eine Unterklasse (namens **Space**) gibt. Außerdem stehen uns die Klassen **SmoothMover**, **Body** und **Vector** zur Verfügung (Abbildung 8.1).

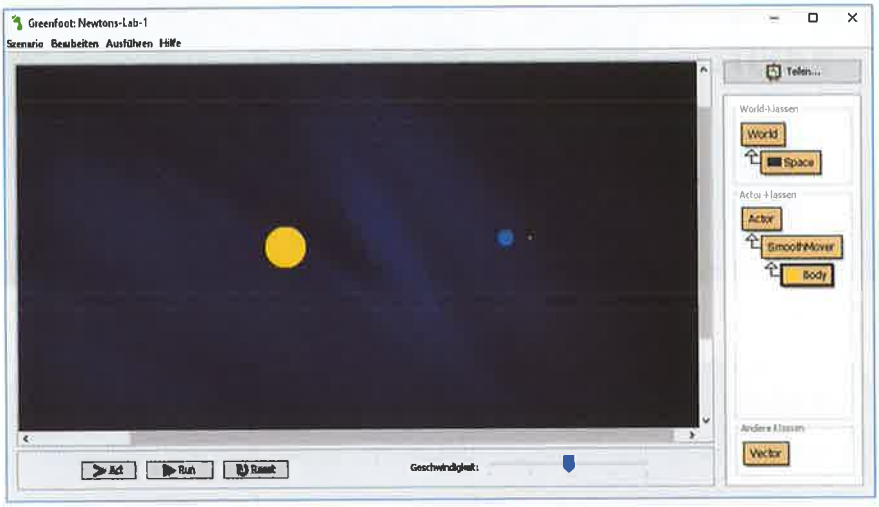


Abbildung 8.1
Das Szenario *Newtons-Lab-1*.

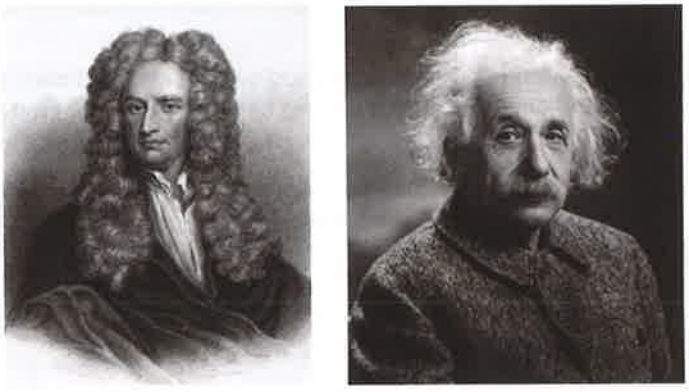


Abbildung 8.2
Isaac Newton und Albert Einstein.[1]

Übung 8.1 Öffne das Szenario *Newtons-Lab-1*. Spiele ein wenig damit herum (platziere einige Körper in dem Weltall). Was kannst du beobachten?

Wenn du versuchst, dieses Szenario auszuführen, wirst du feststellen, dass du zwar Objekte (vom Typ **Body**) in dem Weltall platzieren kannst, doch dass sich diese Körper nicht bewegen, d.h., sie machen noch gar nichts.

Bevor wir jedoch darangehen, die Implementierung zu vervollständigen, wollen wir das Szenario noch ein wenig näher betrachten.

Wenn du den Hintergrund der Welt mit der rechten Maustaste anklickst, kannst du die „öffentlichen" (**public**) Methoden der Klasse **Space** einsehen und aufrufen (Abbildung 8.3).

1 Newton: von Georgios Kollidas/Shutterstock. Einstein: mit freundlicher Genehmigung von Library of Congress Prints and Photographs Division.

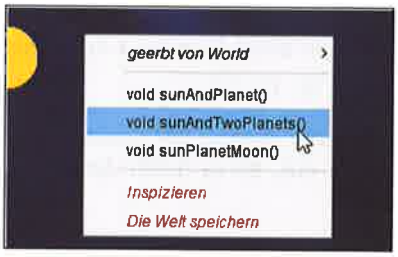


Abbildung 8.3
Die Methoden der **World**-Klasse **Space** in Newtons Labor.

Übung 8.2 Rufe die verschiedenen öffentlichen Methoden des **Space**-Objekts auf. Was kannst du beobachten?

Übung 8.3 Wenn du einen Stern oder Planeten in deiner Welt platziert hast, klicke ihn mit der rechten Maustaste an, um dich über seine öffentlichen Methoden zu informieren. Wie lauten sie?

Übung 8.4 Rufe die Methode **sunPlanetMoon** der öffentlichen Methoden von **Space** auf. Finde heraus, welche Masse die Sonne, der Planet und der Mond haben, und schreibe die Werte auf.

Übung 8.5 Wirf einen Blick auf den Quelltext der Klasse **Space** und zähle, wie viele Methoden darin implementiert sind.

8.2 Hilfsklassen: SmoothMover und Vector

In diesem Szenario verwenden wir zwei allgemein nützliche Hilfsklassen: **SmoothMover** und **Vector**. Hierbei handelt es sich um Klassen, die ein gegebenes Szenario um zusätzliche Funktionalität erweitern. Sie können in verschiedenen Szenarien für gleiche oder ähnliche Zwecke verwendet werden (so werden die beiden Klassen auch in etlichen der anderen Projekten verwendet).

Dank der Klasse **SmoothMover** wird die Bewegung der Akteure viel gleichmäßiger, da diese Klasse die Koordinaten des Akteurs als Dezimalzahlen (vom Typ **double**) und nicht als Integer (ganze Zahlen) speichert. Zustandsfelder vom Typ **double** können Zahlen mit Nachkommastellen (wie 2.4567)[2] speichern und sind deshalb präziser als Integer.

Für die Anzeige des Akteurs auf dem Bildschirm werden die Koordinaten trotzdem noch auf Integer gerundet, da die Positionsangabe zum Zeichnen auf den Bildschirm immer ein ganzes Pixel sein muss. Intern jedoch wird die Position als Dezimalzahl verwaltet.

Ein **SmoothMover**-Objekt kann zum Beispiel die *x*-Koordinate 12.3 haben. Wenn wir diesen Akteur jetzt entlang der *x*-Koordinate in Schritten von 0.6 bewegen, lauten seine nachfolgenden Positionen

12.3, 12.9, 13.5, 14.1, 14.7, 15.3, 15.9, 16.5, 17.1, ...

2 In Java müssen Kommazahlen angloamerikanisch, d.h. mit Punktnotation, geschrieben werden.

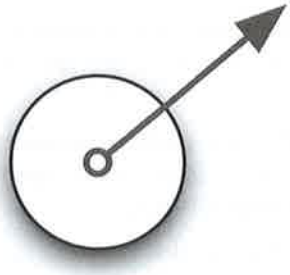
und so weiter. Auf dem Bildschirm aber sehen wir den Akteur an gerundeten *x*-Koordinaten, d.h., er wird an den folgenden *x*-Koordinaten gezeichnet:

12, 13, 14, 14, 15, 15, 16, 17, 17, ...

und so weiter. Auch wenn zum Zwecke der Anzeige auf Integer gerundet wird, sieht die Bewegung ruhiger aus, als es durch den Einsatz von Zustandsfeldern vom Typ **`int`** möglich wäre.

Die zweite Funktionalität, die wir **`SmoothMover`** verdanken, ist ein *Geschwindigkeitsvektor* – ein Vektor, der die aktuelle Richtung und Bewegungsgeschwindigkeit angibt. Wir können uns einen Vektor als einen (unsichtbaren) Pfeil mit einer gegebenen Richtung und Länge vorstellen (Abbildung 8.4).

Abbildung 8.4
Ein **`SmoothMover`**-Objekt mit einem Bewegungsvektor.



Die Klasse **`SmoothMover`** verfügt über eine **`move`**-Methode, die den Akteur gemäß seines aktuellen Vektors bewegt, und über Methoden, die den Geschwindigkeitsvektor – und damit die Bewegung des Objekts – ändern.

Randbemerkung: abstrakte Klassen

Wenn du die Klasse **`SmoothMover`** anklickst, wirst du feststellen, dass du von dieser Klasse keine Objekte erstellen kannst. Es wird kein Konstruktor angezeigt.

Wenn wir den Quelltext dieser Klasse untersuchen, finden wir im Klassenkopf das Schlüsselwort **`abstract`**. Durch die Deklaration einer Klasse als abstrakt können wir bei ihr die Erzeugung von Objekten verhindern. Abstrakte Klassen dienen nur als Oberklassen für andere Klassen, nicht um direkt Objekte davon zu erzeugen.

Übung 8.6 Platziere ein Objekt der Klasse **Body** in der Welt. Rufe das Kontextmenü dieses Objekts auf und finde heraus, welche Methoden dieses Objekt von der Klasse **`SmoothMover`** erbt. Schreibe sie nieder.

Übung 8.7 Welcher der Methodennamen taucht zweimal auf? Inwiefern unterscheiden sich die beiden Versionen?

Terminologie: Überladen

Java erlaubt die Definition zweier gleichnamiger Methoden in einer Klasse, solange diese sich in ihrer Parameterliste unterscheiden. Dies nennt man auch **Überladen**. (Der Name der Methode wird **überladen** – er bezieht sich auf mehr als eine Methode.)

Wenn wir eine überladene Methode aufrufen, stellt das Laufzeitsystem fest, welche der beiden Methoden wir meinen, indem es die mitgelieferten Parameter auswertet.

Wir sprechen auch davon, dass die beiden Methoden unterschiedliche Signaturen haben.

Konzept

Überladen bedeutet, den gleichen Methodennamen für zwei verschiedene Methoden oder Konstruktoren zu verwenden.

Die zweite Hilfsklasse, **Vector**, implementiert den Vektor und wird von der Klasse **SmoothMover** verwendet. Beachte, dass **Vector** nicht in der Gruppe der **Actor**-Klassen aufgeführt wird. Sie ist kein Akteur – sie wird nie für sich allein in der Welt erscheinen. Objekte dieser Klasse werden immer nur von anderen Akteur-Objekten erzeugt und verwendet.

Vektoren können auf zwei Arten dargestellt werden: entweder als ein Paar von Abständen in ihren *x*- und *y*-Koordinaten (*dx*, *dy*) oder als ein Wertepaar, das Richtung und Länge des Vektors angibt. Die Richtung wird dabei normalerweise als Winkel gemessen von der Horizontalen aus angegeben.

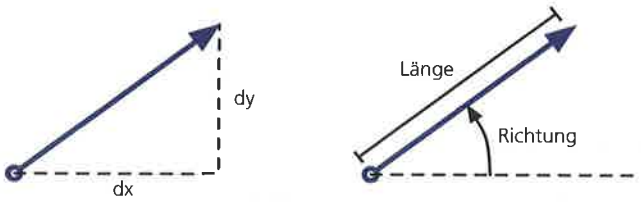


Abbildung 8.5
Zwei Möglichkeiten, einen Vektor anzugeben.

Die erste Darstellung, die die *x*- und *y*-Abstände verwendet, wird auch *kartesische* Darstellung genannt. Die zweite, die die Richtung und die Länge verwendet, heißt *polare* Darstellung. Du findest diese beiden Bezeichnungen in dem Quelltext zu der Klasse **Vector**.

Für unsere Zwecke ist manchmal die kartesische Darstellung günstiger und manchmal die polare. Deshalb ist unsere Vektorklasse für beide Darstellungen ausgelegt. Die dafür notwendigen Umwandlungen werden intern automatisch ausgeführt.

Übung 8.8 Mach dich mit den Methoden der Klassen **SmoothMover** und **Vector** vertraut, indem du den Editor öffnest und ihre Definition in der Ansicht DOKUMENTATION liest. (Zur Erinnerung: Du kannst über das Listenfeld oben rechts im Editor zwischen der Quelltext- und der Dokumentationsansicht wechseln.) Wenn du möchtest, kannst du auch den Quelltext lesen, aber das ist zu diesem Zeitpunkt noch nicht unbedingt nötig.

Übung 8.9 Platziere ein **Body**-Objekt in der Welt. Welche der Methoden, die von **SmoothMover** geerbt wurden, kannst du interaktiv aufrufen (über das Menü des Objekts)? Welche kannst du auf dem aktuellen Stand unseres Szenarios auf diesem Wege nicht aufrufen?

8.3 Die bestehende Klasse Body

Übung 8.10 Öffne den Quelltext für die Klasse **Body** und untersuche ihn.

Im Quelltext der Klasse **Body** gibt es zwei Aspekte, auf die wir näher eingehen sollten. So weist die Klasse zum einen zwei Konstruktoren auf (Listing 8.1). Dies ist ein weiteres Beispiel für Überladen: Es ist völlig in Ordnung, zwei Konstruktoren in einer Klasse zu definieren, sofern diese verschiedene Parameterlisten aufweisen.

In unserem Fall hat der eine Konstruktor überhaupt keine Parameter und der andere vier.

Listing 8.1: Konstruktoren der Klasse **Body**.

```
public class Body extends SmoothMover
{
    // etwas Code ausgelassen

    private double mass;

    /**
     * Erzeugt ein Body-Objekt mit Standardgröße, -masse,
     * -geschwindigkeit und -farbe.
     */
    public Body()
    {
        this (20, 300, new Vector(0, 1.0), defaultColor);
    }

    /**
     * Erzeugt ein Body-Objekt der angegebenen Größe, Masse, Geschwindigkeit
       und Farbe.
     */
    public Body(int size, double mass, Vector velocity, Color color)
    {
        this.mass = mass;
        addToVelocity(velocity);
        GreenfootImage image = new GreenfootImage (size, size);
        image.setColor (color);
        image.fillOval (0, 0, size-1, size-1);
        setImage (image);
    }
    // weiterer Code ausgelassen
}
```

Terminologie

Ein Konstruktor ohne Parameter wird auch **Standardkonstruktor** genannt.

Der Standardkonstruktor ermöglicht uns die interaktive Erzeugung von Körpern, ohne Details angeben zu müssen. Der zweite Konstruktor erlaubt uns, einen Körper mit den von uns angegebenen Werten für Größe, Masse, Geschwindigkeit und Farbe zu erzeugen. Dieser Konstruktor wird zum Beispiel in der Klasse **Space** verwendet, um die Sonne, den Planeten und den Mond zu erzeugen.

Der zweite Konstruktor initialisiert den Zustand des Akteurs mit allen Parameterwerten, die ihm übergeben wurden. Der erste Konstruktor sieht etwas mysteriöser aus. Er besteht aus nur einer Zeile:

```
this (20, 300, new Vector(90, 1.0), defaultColor);
```

Diese Zeile sieht fast aus wie ein Methodenaufruf, sieht man einmal davon ab, dass hier das Schlüsselwort **this** anstelle eines Methodennamens verwendet wird. Mit diesem Aufruf führt der Konstruktor den *anderen* Konstruktor aus (den mit den vier Parametern) und übergibt Standardwerte für alle vier Parameter. Diese Verwendung des Schlüsselwortes **this** (als Methodenname) ist nur in Konstruktoren möglich, um einen anderen Konstruktor als Teil der Initialisierung aufzurufen.

Konzept

Das Schlüsselwort **this** wird verwendet, um einen Konstruktor von einem anderen Konstruktor aus aufzurufen oder um sich auf das aktuelle Objekt zu beziehen.

Es gibt noch eine andere Einsatzmöglichkeit für das Schlüsselwort **this**:

```
this.mass = mass;
```

Hier haben wir es mit einem weiteren Beispiel für Überladen zu tun: Der gleiche Name wird für zwei Variablen (einen Parameter und ein Zustandsfeld) verwendet. Wenn wir diesen Variablen Werte zuweisen, müssen wir angeben, welche der beiden Variablen namens **mass** wir auf jeder Seite der Zuweisung meinen.

Wenn wir **mass** schreiben, ohne es weiter zu qualifizieren, wird die nächstliegende Definition einer Variablen dieses Namens verwendet – in diesem Fall der Parameter. Wenn wir **this.mass** schreiben, geben wir an, dass wir das **mass**-Zustandsfeld des aktuellen Objekts meinen. Daraus folgt, dass die obige Codezeile den Parameter **mass** dem Zustandsfeld namens **mass** zuweist.

Übung 8.11 Entferne in der obigen Codezeile den Code „**this.**" vor **mass**, sodass du die Zeile

```
mass = mass;
```

erhältst. Lässt sich dieser Code kompilieren? Lässt er sich ausführen? Was glaubst du, macht dieser Code? Welchen Effekt hat er? (Erzeuge ein Objekt und verwende seine INSPIZIEREN-Funktion, um das **mass**-Zustandsfeld zu beobachten. Wenn du damit fertig bist, nimm deine Änderungen am Code wieder zurück.)

Der zweite Aspekt, den wir ein wenig näher beleuchten wollen, betrifft die beiden Zeilen ganz oben in der Klasse (Listing 8.2).

Listing 8.2: Deklaration von Konstanten.

```
public class Body extends SmoothMover
{
    private static final double GRAVITY = 5.8;
    private static final Color defaultColor = new Color(255, 216, 0);

    // Rest des Codes ausgelassen
}
```

Diese beiden Deklarationen gleichen den Deklarationen von Zustandsfeldern, mit der Ausnahme, dass hinter dem Schlüsselwort **`private`** die beiden Schlüsselwörter **`static final`** eingefügt sind.

Konzept

Eine **Konstante** ist ein benannter Wert, der ähnlich wie eine Variable verwendet werden kann, sich jedoch niemals ändert.

Dies bezeichnen wir als *Konstante*. Eine Konstante hat insofern Ähnlichkeiten mit einem Zustandsfeld, als wir den Namen in unserem Code verwenden können, um auf seinen Wert Bezug zu nehmen. Dieser Wert kann jedoch von Objekten nie geändert werden (es ist eine *Konstante*). Die Deklarationen von Konstanten können wir an dem Schlüsselwert **`final`** erkennen.

Das Schlüsselwort **`static`** hat den Effekt, dass diese Konstante von allen Akteuren dieser Klasse genutzt werden kann – schließlich benötigen wir nicht einzelne Kopien davon in jedem Objekt. Uns ist das Schlüsselwort **`static`** bereits zuvor (in ***Kapitel 3***) im Zusammenhang mit den Klassenmethoden begegnet. Und genauso wie statische Methoden zu der Klasse selbst gehören (aber von Objekten dieser Klasse aufgerufen werden können), so gehören auch statische Felder zu der Klasse und erlauben den Zugriff aus den Objekten heraus.

In diesem Fall beschreiben die deklarierten Konstanten einen Wert für die Gravitationskraft[3] (der später Verwendung finden soll) und eine Standardfarbe für die Körper. Hierbei handelt es sich um ein Objekt vom Typ **`Color`**, auf das wir weiter hinten in diesem Kapitel noch näher eingehen werden.

Es hat sich bewährt, Zustandsfelder, die sich in einem Programm nicht ändern, als konstant zu deklarieren. Damit wird verhindert, dass im Code aus Versehen Änderungen an dem Wert vorgenommen werden.

8.4 Erste Erweiterung: Bewegung erzeugen

Okay, den bestehenden Code haben wir nun lange genug betrachtet. Es ist Zeit, eigenen Code zu schreiben, damit sich in dem Szenario endlich etwas tut.

Unsere erste Aufgabe soll es sein, die Körper in Bewegung zu setzen. Wir haben bereits erwähnt, dass die Klasse **`SmoothMover`** über eine Methode **`move()`** verfügt, und da **`Body`** von **`SmoothMover`** geerbt hat, hat sie auch Zugriff auf diese Methode.

3 Unser Wert für die Gravitationskraft hat keine direkte Beziehung zu irgendeiner bestimmten Einheit in der Natur. Es ist ein Wert, der willkürlich für dieses Szenario gewählt wurde. Wenn wir dazu übergehen, Gravitationskraft für unsere Körper zu implementieren, kannst du durch Ändern dieses Wertes mit verschiedenen Beträgen für die Gravitationskraft herumexperimentieren.

Übung 8.12 Füge in die **act**-Methode von **Body** einen Aufruf der **move()**-Methode ein. (Stelle sicher, dass du die **move()**-Methode aufrufst, die keine Paramter hat, und nicht die Methode mit dem **int**-Parameter, die von **Actor** geerbt wurde.) Teste deinen Code. Was ist die standardmäßige Bewegungsrichtung? Wie hoch ist die Standardgeschwindigkeit?

Übung 8.13 Erzeuge mehrere **Body**-Objekte. Wie verhalten sie sich?

Übung 8.14 Rufe die öffentlichen **Space**-Methoden (**sunAndPlanet()** usw.) auf und führe das Szenario aus. Wie bewegen sich diese Objekte? Wo sind deren anfängliche Bewegungsrichtung und Geschwindigkeit definiert?

Übung 8.15 Ändere die Standardrichtung eines Körpers in „links". Das heißt, wenn ein Körper mithilfe des Standardkonstruktors erzeugt wird und seine **move()**-Methode ausgeführt wird, soll sich der Körper nach links bewegen.

Wie wir bei diesen Experimenten feststellen, reicht es, den Körpern mitzuteilen, sich zu bewegen, damit sie dies auch tun. Sie werden sich jedoch in einer geraden Linie bewegen. Das liegt daran, dass die Bewegung der Körper (Geschwindigkeit und Richtung) durch ihren Geschwindigkeitsvektor vorgegeben ist und nichts diesen Vektor ändert. Deshalb ist die Bewegung konstant.

8.5 Die Klasse Color

Beim Lesen des Codes oben sind wir sowohl in der Klasse **Body** als auch in der Klasse **Space** auf die Klasse **Color** gestoßen. Der zweite Konstruktor der Klasse **Body** erwartet einen Parameter vom Typ **Color** und der Code in der Klasse **Space** erzeugt **Color**-Objekte mit Ausdrücken wie:

```
new Color(248, 160, 86)
```

Wir sind dieser Klasse im vorigen Kapitel schon kurz begegnet, als wir die Java-Klassenbibliotheken durchgelesen haben.

Übung 8.16 Öffne noch einmal die Dokumentation der Klasse **Color**.

Die drei Parameter des **Color**-Konstruktors sind die roten, grünen und blauen Anteile einer bestimmten Farbe. Jede Farbe auf dem Bildschirm kann als eine Zusammensetzung aus diesen drei Basisfarben beschrieben werden. (Wir werden in ***Kapitel 10*** noch etwas näher auf das Thema Farben eingehen. Dort findest du in ***Abschnitt 10.9*** im Übrigen auch eine Tabelle der RGB-Farbwerte. Und wenn du über ein passendes Grafikprogramm verfügst, nutze doch die Gelegenheit, um selbst ein wenig mit den RGB-Werten zu experimentieren.)

Übung 8.17 Wirf einen Blick auf die Beschreibung der Klasse **Color**. Wie viele Konstruktoren hat sie?

Übung 8.18 Suche nach der Beschreibung für den von uns verwendeten Konstruktor (der mit den drei Integerwerten als Parameter). Wie lautet der gültige Wertebereich für diese Integer?

Wie wir wissen, müssen wir eine Klasse aus der Java-Bibliothek importieren, wenn wir sie benutzen möchten (anders als bei den Klassen des **java.lang**-Pakets). Für die Klasse **Color** lautet die Importanweisung:

```
import java.awt.Color;
```

Übung 8.19 Wo in der Klasse **Body** findest die wichtige Anweisung für die Klasse **Color**?

Übung 8.20 Welche Fehlermeldung erhältst du, wenn du versuchst, die Klasse **Color** zu benutzen, aber diese wichtige Anweisung vergessen hast? (Probiere es aus.)

Die Klasse **Color** ist hilfreich in Greenfoot, wenn wir etwas in Bilder hineinzeichnen möchten. In diesem Fall zeichnen wir einen runden Kreis auf das Bild des Akteurs. Wir werden dieser Klasse in späteren Kapiteln erneut begegnen.

Denn als Nächstes wollen wir unser Szenario um die Gravitationskraft ergänzen. Das heißt, wenn wir mehr als einen Körper in unserem Weltraum haben, sollen die Gravitationskräfte zwischen diesen Körpern Einfluss auf die Bewegung der Körper nehmen.

8.6 Gravitationskraft hinzufügen

Zuerst wollen wir uns die aktuelle **act**-Methode in unserer **Body**-Klasse (Listing 8.3) anschauen. (Wenn du Übung 6.12 noch nicht nachvollzogen hast, fehlt hier der Aufruf der **move**-Methode – den du jedoch nachträglich einfügen kannst.)

Listing 8.3: Die aktuelle **act**-Methode.

```
/**
 * Agiert. Das heißt: Wendet die Gravitationskräfte von allen
 * anderen vorhandenen Körpern an und bewegt sich anschließend.
 */
public void act()
{
    move();
}
```

Während der Code zurzeit nur den Aufruf von **move** enthält, beschreibt der Kommentar bereits korrekt, was wir zu tun beabsichtigen: Bevor wir uns bewegen, soll-

ten wir die Kräfte anwenden, die von den Gravitationskräften aller anderen Objekte im Weltraum ausgeübt werden.

In Pseudocode lässt sich die Aufgabe wie folgt umreißen:

```
Wende die Kräfte der anderen Körpern an:
    ermittle alle anderen Körper im Weltraum;
    for jeden dieser Körper:
    {
        wende die Gravitationskraft dieses Körpers auf den
        eigenen an;
    }
```

Hier haben wir wieder den Fall, dass wir eine Liste von Objekten und eine **for-each**-Schleife verwenden wollen, wie wir es im vorigen Kapitel geübt haben.

Dies ist nicht ganz einfach, sodass unser erster Schritt darin bestehen soll, eine separate Methode für diese Aufgabe zu definieren (Listing 8.4). Die Erzeugung einer neuen (anfänglich leeren) Methode mag auf den ersten Blick trivial erscheinen und uns nicht viel einbringen. Eine solche Methode hilft jedoch, unser Problem in kleinere Teilprobleme zu zerlegen und unsere Gedanken zu strukturieren.

Listing 8.4
Vorbereitungen, um die Gravitationskräfte zu berücksichtigen.

```
/**
 * Agiert. Das heißt: Wendet die Gravitationskräfte von allen
 * anderen vorhandenen Körpern an und bewegt sich anschließend.
 */
public void act()
{
    applyForces();
    move();
}
```

```
/**
 * Wendet die Gravitationskräfte aller anderen Himmelskörper
 * in diesem Universum an.
 */
private void applyForces()
{
    // noch zu erledigen
}
```

Übung 8.21 Wenn du dich sicher mit dem Umgang der Methode **getObjects** aus der **List**-Klasse und der **for-each**-Schleife fühlst, versuch dieses jetzt in deiner eigenen **Body**-Klasse zu implementieren. (Wenn du nicht so sicher bist, lies einfach weiter. Wir werden dies nun Schritt für Schritt durchgehen.)

Unser obiger Pseudocode gibt uns das Grundgerüst unserer **applyForces()**-Methode vor. Wir können sehen, dass wir als Erstes Zugriff auf alle anderen Körper im Weltraum benötigen. Dazu verwenden wir dieselbe **getObjects**-Methode wie im vorigen Kapitel.

Übung 8.22 Schreibe den Aufruf auf, mit dem du Zugriff auf alle Objekte vom Typ **Body** im Weltraum erlangst. Die Objekte werden als Liste zurückgegeben. Deklariere eine Variable für diese Liste und weise ihr die Liste der Körper zu, die du erhältst.

Übung 8.23 Schreibe, zuerst auf Papier, die **for-each**-Schleife auf, die diese Liste durchläuft.

Der Code, den wir hier schreiben müssen, ähnelt sehr dem Code, den wir für das Ändern der Bilder aller Blätter im vorigen Kapitel geschrieben haben. Sieh dir diesen Code noch einmal an, wenn du dir nicht ganz sicher bist.

Übung 8.24 Sobald du mit der Papierversion deiner Schleife zufrieden bist, schreibe sie in die **applyForces()**-Methode deiner **Body**-Klasse. Denk daran, die Importanweisung für **java.util.List** hinzuzufügen. Kompiliere deinen Code, um zu prüfen, ob die Syntax korrekt ist.

Nachdem wir nun herausgefunden haben, wie wir unsere Schleife schreiben, müssen wir uns entscheiden, was in den Schleifenrumpf kommt. Unser Pseudocode sagt uns, dass wir die Gravitationskraft des anderen Körpers auf unseren eigenen anwenden wollen. Wie immer, wenn wir auf eine etwas knifflige Aufgabe treffen, bei der wir nicht sofort wissen, wie wir sie lösen sollen, fügen wir einfach einen Methodenaufruf für eine neue Methode mit entsprechendem Namen ein – und kümmern uns später um die Implementierung dieser Methode.

Übung 8.25 Erstelle einen Platzhalter (eine leere Methode) für eine private Methode namens **applyGravity(Body body)**. Rufe diese Methode vom Inneren der Schleife in deiner **applyForces()**-Methode aus auf. Übergib dabei das aktuelle Listenelement als Parameter.

Wenn du die Übungen bis hierher geschafft hast, bist du fast fertig. Die vollständige Implementierung der Methode **applyForces()** siehst du in Listing 8.5 Wenn du diesen Code gut durchliest, wird dir auffallen, dass wir noch ein weiteres Konstrukt eingefügt haben: die Schleife enthält nun eine **if**-Anweisung:

```
if (body != this)
{
    ...
}
```

Übung 8.26 Vergleiche deinen eigenen Code mit unserem. Falls nötig, vervollständige deine **applyForces()**-Methode noch.

Der Grund dafür ist, dass **bodies** eine Liste aller Körper im Weltraum ist, die auch das aktuelle Objekt umfasst (auf das wir die Gravitationskräfte anwenden wollen). Wir müssen allerdings nicht die Gravitationskraft eines Objekts auf sich selbst anwenden, sodass wir eine **if**-Anweisung hinzufügen, die **applyGravity** nur aufruft, wenn das Element der Liste nicht das aktuelle Objekt selbst ist. Beachte, wie das Schlüsselwort **this** hier verwendet wird, um auf das aktuelle Objekt zu verweisen.

```
private void applyForces()
{
    List<Body> bodies = getWorld().getObjects(Body.class);

    for (Body body : bodies)
    {
        if (body != this)
        {
            applyGravity(body);
        }
    }
}

/**
 *  Wendet die Gravitationskraft eines gegebenen Körpers auf diesen an.
 */
private void applyGravity(Body other)
{
    //noch zu erledigen
}
```

Listing 8.5
Die Gravitationskräfte von allen anderen Körpern im Weltraum anwenden.

8.7 Gravitationskraft anwenden

In Listing 8.5 haben wir die Aufgabe gelöst, wie auf jedes Objekt im Weltraum zugegriffen wird, stehen aber immer noch vor dem Problem, die Gravitationskräfte tatsächlich anzuwenden. Die Methode **applyGravity** muss noch geschrieben werden.

Dies dürfte uns mittlerweile aber etwas leichter fallen, da diese Methode jetzt nur noch mit zwei Objekten zu tun hat: das aktuelle Objekt und ein anderes Objekt, das als Parameter übergeben wird. Wir wollen jetzt die Gravitationskraft des anderen Objekts auf unser Objekt anwenden. Das ist der Punkt, an dem das Newton'sche Gesetz ins Spiel kommt.

Newtons Formel der Gravitationskraft lautet folgendermaßen:

$$\text{Kraft} = \frac{\text{Masse}_1 \cdot \text{Masse}_2}{\text{Entfernung}^2} G$$

In anderen Worten, um die Kraft zu berechnen, die auf das aktuelle Objekt anzuwenden ist, müssen wir die Masse dieses Objekts mit der Masse des anderen Objekts multiplizieren und dann durch das Quadrat der Entfernung der beiden Objekte teilen. Zum Schluss wird der erhaltene Wert mit der Konstanten **G** – der sogenannten *Gravitationskonstanten* – multipliziert. (Vielleicht erinnerst du dich, dass wir bereits eine Konstante für diesen Wert (namens **GRAVITY**) in unserer Klasse definiert haben.)

Wenn du dich sicher genug fühlst oder Pioniergeist zeigen möchtest, kannst du dich an der Implementierung von **applyGravity** erst einmal selbst versuchen. Du musst einen Vektor vom aktuellen Körper zum anderen Körper erzeugen mit einer Länge, die von dieser Formel vorgegeben wird. Die restlichen Leser können sich stattdessen mit der fertigen Implementierung dieser Methode beschäftigen (Listing 8.6).

Listing 8.6: Die Gravitationskraft eines anderen Körpers berechnen und anwenden.

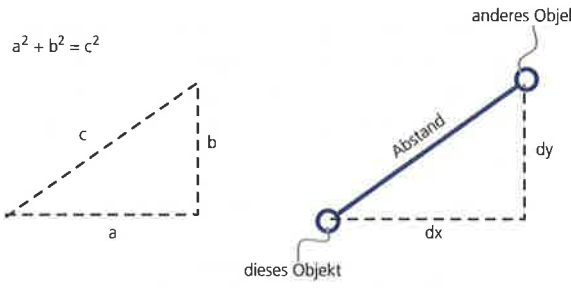
```
/**
 *  Wendet die Gravitationskraft eines gegebenen Körpers auf diesen an.
 */
private void applyGravity(Body other)
{
    double dx = other.getExactX() - this.getExactX();
    double dy = other.getExactY() - this.getExactY();
    Vector dv = new Vector (dx, dy); //Richtung bestimmen
    double distance = Math.sqrt (dx*dx + dy*dy);
    double force = GRAVITY * this.mass * other.mass / (distance * distance);
    double acceleration = force / this.mass;
    dv.setLength (acceleration);
    addToVelocity (dv);
}
```

Diese Methode ist nicht so kompliziert, wie sie aussieht. Zuerst berechnen wir die Entfernungen zwischen unserem Objekt und dem anderen Objekt in *x*- und *y*-Richtung (*dx* und *dy*). Dann erzeugen wir einen neuen Vektor unter Zuhilfenahme dieser Werte. Dieser Vektor hat bereits die richtige Richtung, aber nicht die korrekte Länge.

Als Nächstes berechnen wir die Entfernung zwischen den beiden Objekten mithilfe des Satzes des Pythagoras ($a^2 + b^2 = c^2$) für ein rechtwinkliges Dreieck (Abbildung 8.6).

Die Abbildung verrät uns, dass die Entfernung die Quadratwurzel von *dx* zum Quadrat plus *dy* zum Quadrat ist. In unserem Code (Listing 8.6) verwenden wir eine Methode namens **sqrt** der Klasse **Math**, um die Quadratwurzel zu berechnen. (**Math** ist eine Klasse in **java.lang** und wird deshalb automatisch importiert.)

Abbildung 8.6 Die Entfernung, zerlegt in *dx* und *dy*.



Übung 8.27 Suche die Klasse **Math** in der Java-Dokumentation. Wie viele Parameter hat die Methode **sqrt**? Von welchem Typ sind sie? Welchen Typ liefert diese Methode zurück?

Übung 8.28 Suche in der Klasse **Math** nach der Methode, mit der man das Maximum zweier Integer ermitteln kann. Wie heißt sie?

Die nächste Codezeile berechnet die Stärke der Anziehungskraft mithilfe des oben angegebenen Newton'schen Gravitationsgesetzes.

Zum Schluss müssen wir noch die Beschleunigung berechnen, da die eigentliche Änderung der Geschwindigkeit für unser Objekt nicht nur durch die Gravitationskraft bestimmt wird, sondern auch durch die Masse unseres Objekts: Je schwerer das Objekt, desto langsamer wird es beschleunigen. Die Beschleunigung wird mit folgender Formel berechnet:

$$\text{Beschleunigung} = \frac{\text{Kraft}}{\text{Masse}}$$

Sobald wir die Beschleunigung berechnet haben, können wir unseren neuen Kraftvektor auf die korrekte Länge setzen und diesen Vektor zu der Geschwindigkeit unseres Körpers hinzuaddieren. Dies ist ganz einfach; du musst lediglich die Methode **`addToVelocity`** verwenden, die von der Klasse **`SmoothMover`** bereitgestellt wird.

Übung 8.29 Ordne die Variablen aus Listing 8.6 der oben abgedruckten Newton'schen Formel, dem Satz des Pythagoras und der Beschleunigungsformel zu. Welche Variable entspricht welchem Teil welcher Formel?

Damit ist unsere Aufgabe beendet. (Eine Implementierung des bisher beschriebenen Codes findest du in den Buchszenarien unter *Newtons-Lab-2*.)

In dieser Aufgabe war eindeutig mehr mathematisches und physikalisches Hintergrundwissen gefragt als in den anderen, die wir bisher kennengelernt haben. Es mag auf den ersten Blick kompliziert aussehen und du hast vielleicht das Gefühl, dass du dies nicht alleine hättest schreiben können. Keine Sorge – das ist normal. Hier geht es darum, dass du den Code untersuchst und verstehst. Wenn du ihn verstehst und erklären kannst, dann hast du dieses Ziel erreicht. Wir erwarten an dieser Stelle nicht, dass du in deinem Code Konzepte verwendest, denen du gerade zum ersten Mal begegnest. Doch sie zu analysieren wird dir helfen, später ähnlichen Code zu schreiben, wenn du auf eine verwandte Fragestellung triffst.

Wenn Mathematik und Physik nicht gerade deine Stärken sind, mach dir keine Sorgen, wir werden uns bald weniger mathematischen Projekten zuwenden. Denke immer daran: Programmierung kann alles, was du magst. Du kannst den Schwerpunkt auf die Mathematik legen, aber genauso gut auch auf Kreativität und künstlerisches Design.

8.8 Ausprobieren

Nachdem wir unsere Implementierung der Gravitationskräfte abgeschlossen haben, wollen wir sie auch testen. Für den Anfang verwenden wir erst einmal die drei vordefinierten Szenarien der Klasse **Space**.

Übung 8.30 Jetzt, wo der Code zur Anwendung der Gravitationskräfte fertig ist, testen wir erneut die drei Initialisierungsmethoden des **Space**-Objekts (**`sunAndPlanet()`**, **`sunAndTwoPlanets()`** und **`sunPlanetMoon()`**). Was kannst du beobachten?

Übung 8.31 Experimentiere mit unterschiedlichen Werten für die Gravitationskraft (die Konstante **`GRAVITY`** ganz oben in der Klasse **Body**).

Übung 8.32 Experimentiere mit unterschiedlichen Werten für die Masse und/oder Anfangsbewegung der Körper (definiert in der Klasse **Space**).

Übung 8.33 Erzeuge einige neue Sterne-Planeten-Konstellationen und beobachte, wie sie sich verhalten. Findest du ein System, das stabil ist?

Fallstricke

Sei vorsichtig bei der Verwendung des Konstruktors der Klasse **`Vector`**. Der Konstruktor ist überladen: Eine Version erwartet als Parameter zwei **`double`**-Werte, die andere Version einen Parameter vom Typ **`int`** und einen anderen vom Typ **`double`**. Das heißt

```
new Vector(32, 12.0)
```

ruft den einen Konstruktor auf, während

```
new Vector (32.0, 12.0)
```

den anderen Konstruktor aufruft (und einen völlig anderen Vektor erzeugt).

Du wirst schnell feststellen, dass es sehr schwierig ist, die Parameter so einzustellen, dass das System für eine längere Zeit stabil bleibt. Die Kombination von Masse und Gravitationskraft führt oft dazu, dass die Objekte kollidieren oder den Weltraum verlassen. (Da wir das „miteinander kollidieren" noch nicht implementiert haben, können Objekte mehr oder weniger durch andere Objekte hindurchfliegen. Wenn sie sich jedoch sehr nahe kommen, werden ihre Kräfte sehr groß und sie katapultieren sich oft auf ziemlich seltsame Flugbahnen.)

Einige dieser Effekte kannst du in ähnlicher Form auch in der Natur beobachten, auch wenn unsere Simulation aufgrund einiger Vereinfachungen ein wenig ungenau ist. So hat beispielsweise die Tatsache, dass alle Objekte nacheinander agieren statt gleichzeitig, einen Effekt auf das Verhalten und ist keine realistische Darstellung. Dies führt zu kleinen Fehlern, die sich aufsummieren und im Laufe der Zeit einen Einfluss bekommen. Damit die Simulation realistischer ist, müssten wir zuerst alle Kräfte berechnen (ohne Bewegungen) und dann alle Bewegungen gemäß den Rechenergebnissen ausführen. Außerdem bildet unsere Simulation die Kräfte nicht akkurat ab, wenn sich zwei Körper sehr nahe kommen, was zu weiteren unrealistischen Effekten führt.

Wir können die Frage nach der Stabilität auch für unser eigenes Sonnensystem stellen. Während die Umlaufbahnen der Planeten unseres Sonnensystems relativ stabil sind, ist es schwierig, weit im Voraus genaue Angaben zu ihrer Bewegung zu machen. Wir sind ziemlich sicher, dass in den nächsten paar Milliarden Jahren keiner der Planeten in die Sonne stürzen wird. Aber kleine Änderungen an der Umlaufbahn können schon auftreten. Simulationen wie unsere (nur wesentlich genauer und detaillierter, aber ähnlich vom Prinzip her) wurden herangezogen, um zukünftige Umlaufbahnen vorherzusagen. Wir haben jedoch gesehen, dass dies nur sehr schwer genau zu simulieren ist. Simulationen können zeigen, dass winzige Unterschiede in den Anfangsbedingungen nach Milliarden von Jahren einen riesigen Unterschied machen können.[4]

Nachdem wir wissen, wie schwer es ist, Parameter zu finden, die – und sei es auch nur kurzfristig – ein stabiles System erzeugen, ist es umso überraschender, dass unser Sonnensystem so stabil ist, wie es ist. Doch hierfür gibt es eine Erklärung: Als sich das Sonnensystem entwickelte, verdichtete sich Materie aus einer Gaswolke um die Sonne und bildete Massehaufen, die durch die Kollision mit anderen Massehaufen langsam immer größer wurden und sich zu immer weiter wachsenden Massehaufen zusammenschlossen. Am Anfang gab es unzählige dieser Massehaufen in der Umlaufbahn. Mit der Zeit fielen einige in die Sonne und andere verloren sich in den Tiefen des Weltalls. Dieser Prozess endet, wenn die einzigen Stücke, die übrig bleiben, weit genug voneinander getrennt sind und sich auf allgemein stabilen Umlaufbahnen bewegen.

Es wäre möglich, auch diesen Effekt zu simulieren. Wenn wir korrekt das Wachsen von Planeten aus Milliarden von kleinen zufälligen Massehaufen modellieren, würden wir den gleichen Effekt feststellen: Es bilden sich große Planeten, die sich auf ziemlich stabilen Umlaufbahnen bewegen. Hierfür würden wir jedoch eine wesentlich ausführlichere und kompliziertere Simulation benötigen sowie viel Zeit: Diesen Effekt zu simulieren würde sehr, sehr lange dauern, sogar auf sehr schnellen Computern.

8.9 Gravitationskraft und Musik

Bevor wir unser Szenario *Newton's Lab* verlassen, wollen wir uns noch mit einer weiteren Sache beschäftigen: Wir wollen Musik – oder sagen wir besser Geräusche – hinzufügen.[5]

Die Idee ist folgende: Wir fügen eine Reihe von *Hindernissen* (*obstacles*) in unsere Welt ein, die, wenn sie von einem unserer Planeten berührt werden, einen Ton von sich geben. Anschließend erzeugen wir einige Planeten oder Sterne, lassen sie umherfliegen und schauen, was passiert.

Wir wollen diese Implementierung hier nicht im Detail besprechen, sondern dir zum Selbststudium überlassen und nur einige interessante Punkte ansprechen. Du findest eine Implementierung dieser Idee in den Buchszenarien unter *Newtons-Lab-3*.

4 Wenn dich dieser Themenkomplex interessiert, ist Wikipedia ein guter Ausgangspunkt: *http://de.wikipedia.org/wiki/Sonnensystem*.

5 Die Idee, unser Gravitationskraftprojekt mit Musik zu verbinden, wurde von Keplers Modell des Sonnensystems inspiriert (siehe *https://keplers-orrery.dev.java.net/* oder suche nach „Kepler's Orrery" auf YouTube).

Übung 8.34 Öffne das Szenario namens *Newtons-Lab-3* und führe es aus. Schaue dir den Quelltext an. Versuche zu verstehen, wie er funktioniert.

Im Folgenden findest du eine Übersicht über die interessantesten Änderungen, die wir an der Vorgängerversion vorgenommen haben:

- Wir haben für die Hindernisse eine neue Klasse namens **`Obstacle`** eingefügt. Objekte dieser Klasse sind gut auf dem Bildschirm zu erkennen. Hindernisse haben zwei Bilder: ein oranges Rechteck, das du meistens siehst, und eine hellere Rechteckversion, die bei Berührung mit einem Planeten angezeigt wird. Dadurch soll ein „Aufhell"-Effekt erzeugt werden. Hindernisse werden außerdem – wie die Klaviertasten in unserem Piano-Szenario – mit einer Sounddatei verbunden. Wir verwenden hier sogar die Sounddateien unseres Piano-Szenarios, sodass die Töne gleich klingen.
- Wir haben Änderungen an der Klasse **Body** vorgenommen, sodass die Körper jetzt von den Bildschirmrändern abprallen. Damit erhältst du für diese Art von Szenario einen besseren Effekt. Außerdem haben wir für eine schnellere Bewegung die Gravitationskraft etwas erhöht und Codeänderungen vorgenommen, die die Körper automatisch abbremsen, sobald sie zu schnell werden. Ansonsten könnten sich die Körper immer weiter bis ins Unendliche beschleunigen.
- Und schließlich haben wir Code in die Klasse **Space** eingefügt, um eine feste Reihe von Hindernissen sowie fünf zufällige Planeten (Zufallsgröße, -masse und -farbe) zu erzeugen.

Die Implementierung dieser drei Änderungen umfasst einige Codeabschnitte, auf die wir besonders hinweisen wollen.

- In der Klasse **`Obstacle`** verwenden wir wieder die Methode **`getOneIntersectingObject`**, um zu prüfen, ob das Hindernis von einem Planeten getroffen wurde. Der Code dazu sieht folgendermaßen aus:

```
Object body = getOneIntersectingObject(Body.class);
if (body != null)
{
    ...
}
```

  Wir haben diese Methode bereits im vorigen Kapitel verwendet.

- In der Klasse **Space** haben wir eine Methode ergänzt, **`createObstacles`**. Sie erzeugt die Hindernisse mit den damit verbundenen Sounddateinamen, recht ähnlich dem Initialisierungscode unseres Klavier-Beispiels.
- Wir haben eine weitere Methode der Klasse Space namens RandomBodies hinzugefügt. Diese verwendet eine **`while`**-Schleife, um eine Reihe von **Body**-Objekten zu erzeugen. Die Körper werden mit Zufallswerten initialisiert. Die **`while`**-Schleife zählt von einer gegebenen Zahl abwärts bis null, um die korrekte Anzahl an Objekten zu erzeugen. Es lohnt sich, sie als eine weitere Form einer Schleife näher zu betrachten.

Wir hoffen, dass dir dieses Szenario einige Ideen bringt, was man noch alles mit diesem Projekt ausprobieren kann. Wenn du erst einmal angefangen hast, Kreativität und Programmierung zu verbinden, dann sind deinem Einfallsreichtum keine Grenzen gesetzt.

Zusammenfassung der Programmiertechniken

In diesem Kapitel haben wir eine ganze Reihe von neuen Konzepten behandelt. Wir haben ein neues Szenario kennengelernt, das unter Zugrundelegung von Newtons Gesetz der Gravitationskraft Sterne und Planeten im Weltraum simuliert. Simulationen sind allgemein ein sehr interessantes Thema, sodass wir in ***Kapitel 11*** noch einmal darauf zurückkommen werden.

Wir haben die Bekanntschaft zweier nützlicher Hilfsklassen (**`SmoothMover`** und **`Vector`**) gemacht, mit deren Hilfe wir anspruchsvollere Bewegungen erzeugen können.

Das Überladen von Methoden ist ein weiteres Konzept, dem wir begegnet sind: Hier kann derselbe Methodenname für mehr als eine Methode eingesetzt werden. Wir haben außerdem verschiedene Verwendungsmöglichkeiten des Schlüsselworts **`this`** entdeckt, das eingesetzt werden kann, um auf das aktuelle Objekt zu verweisen, um auf Felder mit überladenen Namen zuzugreifen und um einen Konstruktor durch einem anderen aufzurufen.

Schließlich haben wir noch einmal die Verwendung einer Objektliste im Zusammenhang mit einer **`for-each`**-Schleife gesehen. Dies ist das wichtigste – und wahrscheinlich schwierigste – Konzept, das wir hier behandelt haben, deshalb haben wir es ausführlich eingeführt.

Zusammenfassung der Konzepte

- **Überladen** bedeutet, den gleichen Methodennamen für zwei verschiedene Methoden oder Konstruktoren zu verwenden.
- Das Schlüsselwort **`this`** wird verwendet, um einen Konstruktor von einem anderen Konstruktor aus aufzurufen oder um sich auf das aktuelle Objekt zu beziehen.
- Eine **Konstante** ist ein benannter Wert, der ähnlich wie eine Variable verwendet werden kann, sich jedoch niemals ändert.

Vertiefende Aufgaben

Die letzte Version des Newton-Lab-Szenarios enthält viele Möglichkeiten, um unterschiedliche Konstrukte einzuüben. Hier sind ein paar Ideen.

Übung 8.35 Ändere die Anzahl der Körper, die standardmäßig in diesem Szenario erzeugt werden.

Übung 8.36 Spiele mit den Werten für die Bewegungsparameter, um zu sehen, ob du nicht bessere Bewegungen der Planeten erzeugen kannst. Die Parameter sind: der Wert für **GRAVITY**; der Beschleunigungswert, der beim Abprallen vom Rand verwendet wird (zurzeit 0.9); und die in der Methode **applyForces** verwendeten Werte für die Geschwindigkeitsschwelle (zurzeit 7) und die Beschleunigung (0.9), um die schnellen Objekte zu verlangsamen; und die Anfangsmasse der Planeten (in der Klasse **Space**).

Übung 8.37 Erzeuge eine andere Anordnung der Hindernisse in deinem Szenario.

Übung 8.38 Verwende andere Töne (andere Sounddateien) für deine Hindernisse.

Übung 8.39 Verwende andere Bilder für deine Hindernisse.

Übung 8.40 Lasse deine Planeten jedes Mal die Farbe wechseln, wenn sie vom Rand des Universums abprallen.

Übung 8.41 Lasse deine Planeten jedes Mal die Farbe wechseln, wenn sie auf ein Hindernis treffen.

Übung 8.42 Erzeuge Hindernisse, die bei jedem Treffer an- und ausgeschaltet werden. Wenn sie angeschaltet sind, sollen sie ständig blinken und in regelmäßigen Abständen einen Ton erzeugen.

Übung 8.43 Baue in dein Szenario Tastatursteuerung ein. So könnte zum Beispiel das Drücken der rechten Pfeiltaste auf alle **Body**-Objekte eine kleine Kraft nach rechts ausüben.

Übung 8.44 Erlaube das Hinzufügen weiterer Planeten. Ein Mausklick in das Universum soll während der Ausführung einen neuen Planeten an dieser Position erzeugen.

Es gibt zahllose weitere Möglichkeiten, dieses Szenario optisch und technisch interessanter zu gestalten. Lasse deine Fantasie spielen und implementiere deine Ideen!