KAPITEL

5 Punktezählung

Lernziele

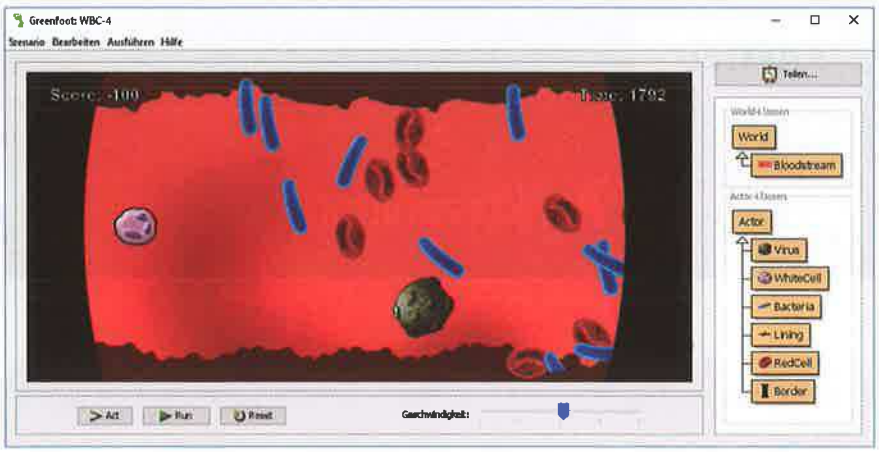
Themen: Eingeschränkte Bewegung, Textnachrichten anzeigen, Punkte zählen, Objektinteraktion

Konzepte: Strings, String-Verknüpfung, Abstraktion (ein erster Blick), Casting, **`this`** (Schlüsselwort)

Ein offensichtliches Element, das in unserem Krabbenbeispiel fehlt, ist die Möglichkeit der Punktezählung. Den ersten Schritt dazu haben wir schon gemacht: die Krabbe führt innerlich Buch darüber, wie viele Würmer sie gefressen hat. Dazu hatten wir eine Integer-Variable verwendet. Wir haben gesehen, dass Variablen für das Zählen von Punkten sehr wichtig sind, aber sie sind nur ein Teil der Lösung. Zunächst einmal kann es sein, dass wir auch andere Aktionen einbeziehen möchten, um Punkte zu gewinnen oder zu verlieren. Zweitens haben wir bisher den Punktestand noch nicht auf dem Bildschirm dargestellt. Um etwas mehr Abwechslung zu erhalten, wollen wir dies anhand eines anderen Beispiels diskutieren. Wenn du dieses Kapitel durchgelesen hast, kannst du, wenn du möchtest, zum Krabbenspiel zurückkehren und dort ebenfalls eine Punktezählung einbauen – bis zum Ende dieses Kapitels solltest du gelernt haben, wie man das macht.

Für unsere Diskussion verwenden wir ein Szenario namens *WBC* (was für „White Blood Cell" – weiße Blutzelle – steht, Abbildung 5.1). Dies ist ein kleines Spiel, bei dem wir eine weiße Blutzelle steuern, die in der Blutbahn irgendeines Lebewesens schwimmt. Unsere Aufgabe besteht darin, Bakterien zu fangen und zu entfernen. Um die Sache etwas interessanter zu machen, gibt es auch Viren, und wir stellen uns vor, dass unsere Art von weißer Blutzelle nur Bakterien neutralisieren kann, aber keine Viren. Viren sind zu stark für uns und beschädigen unsere Zelle, daher müssen wir sie meiden.

Abbildung 5.1
Das WBC-Szenario.



Dieses Kapitel ist in drei Teile gegliedert: Im ersten Teil werfen wir zuerst einen kurzen Blick auf das Anfangsszenario und analysieren dieses (Abschnitt 5.2 bis 5.4). Dann fügen wir einige weitere Objekte sowie mehr Funktionalität hinzu, dabei verwenden wir Techniken, die wir schon in den vorherigen Kapiteln kennengelernt haben (Abschnitte 5.5 bis 5.10). Wir werden diese beiden Teile eher kurz abhandeln, da wir die Konzepte bereits kennen. Zu guter Letzt gehen wir dazu über, einige neue Konstrukte zu untersuchen, mit deren Hilfe wir die Punktezählung einführen können.

5.1 WBC: der Ausgangspunkt

Öffne das Szenario *WBC-1*, das du in den Buchprojekten findest, und probiere es aus.

Übung 5.1 Öffne das Szenario *WBC-1* und führe es aus. Beschreibe, was du beobachtest.

Übung 5.2 Gib für jede Klasse des Szenarios eine kurze Beschreibung (ein bis zwei Sätze), was diese Klasse repräsentiert.

Übung 5.3 Öffne den Quelltext für jede der vier Klassen. Untersuche den Code und versuche herauszufinden, was er macht. Markiere jedes Codestück, das dir unklar ist.

Du bist sicher in der Lage, den Großteil des Codes zu verstehen. Wir werden jetzt kurz die interessanten Elemente dieser Klassen durchgehen.

Die Klasse **`Lining`** ist trivial – sie macht nichts. Objekte dieser Klasse dienen nur der Dekoration: sie werden entlang der unteren und oberen Ränder des Bildschirms platziert und interagieren mit keinem anderen Objekt.

Die anderen Klassen verdienen eine genauere Betrachtung.

5.2 WhiteCell: eingeschränkte Bewegung

Die Klasse **WhiteCell** definiert eine weiße Blutzelle – dies ist unser Spielobjekt, das wir mit unserer Tastatur steuern. Die Struktur des Codes ist relativ geradlinig: die **act**-Methode ruft nur eine andere Methode auf, die auf Tasteneingabe prüft und bei Drücken der ↑- und ↓-Tasten reagiert (Listing 5.1).

Listing 5.1
Die Methoden von **WhiteCell**.

```
/**
 * Aktion: auf- und abbewegen, wenn Pfeiltasten gedrückt werden.
 */
public void act()
{
    checkKeyPress();
}

/**
 * Prüft, ob eine Taste auf der Tastatur gedrückt wurde und
 * reagiert, falls dies zutrifft.
 */
private void checkKeyPress()
{
    if (Greenfoot.isKeyDown("up"))
    {
        setLocation(getX(), getY()-4);
    }

    if (Greenfoot.isKeyDown("down"))
    {
        setLocation(getX(), getY()+4);
    }
}
```

Die einzigen interessanten Teile sind die beiden Zeilen der Form

```
setLocation(getX(), getY()-4);
```

Diese Zeilen sind in der Tat für die Bewegung des Objekts verantwortlich. Wir benutzen hier drei Methoden der Klasse **Actor**, die wir bisher noch nicht kennen: **setLocation(x,y)**, **getX()** und **getY()**.

Übung 5.4 Suche diese drei Methoden in der Dokumentation der Klasse **Actor**. Schreibe ihre Signatur auf.

Die Methoden **getX()** und **getY()** geben die aktuellen *x*- bzw. *y*-Koordinaten als Integerwert zurück.

Die Methode **setLocation(x,y)** erwartet zwei Parameter, eine *x*- und eine *y*-Koordinate, und platziert den Akteur an dieser Stelle. Die Signatur lautet:

```
void setLocation(int x, int y)
```

Wird also die Methode mit den Koordinaten 120 und 200 durch

```
setLocation(120, 200);
```

aufgerufen, dann wird unsere Blutzelle an die Stelle (120, 200) teleportiert.

Wir können nun die Methoden **getX()** und **getY()** anstelle der erwarteten Koordinaten verwenden:

```
setLocation(getX(), getY());
```

Konzept

Die Methode **setLocation** legt den Standort eines Akteurs fest, und zwar an der Position, die durch die *x*- und *y*-Koordinaten bestimmt ist.

Wenn ein Methodenaufruf (**getX()**) als Parameter innerhalb eines anderen Methodenaufrufs (**setLocation(..)**) eingesetzt wird, so wird zuerst der innere Methodenaufruf ausgewertet und das Ergebnis wird als Parameter für den äußeren Methodenaufruf verwendet.

In diesem Fall heißt das: Zuerst wird unsere aktuelle *x*-Koordinate ermittelt, dann die aktuelle *y*-Koordinate, danach bestimmen diese Koordinaten unsere Position. Die Anweisung hat also in dieser Form keinen sichtbaren Effekt: Wir positionieren uns dort, wo wir sowieso schon sind.

Mit einer kleinen Änderung können wir jedoch eine Bewegung erzeugen:

```
setLocation(getX(), getY()-4);
```

Jetzt ermitteln wir zuerst unsere *y*-Koordinate, subtrahieren davon 4 und verwenden dieses Ergebnis als neue *y*-Koordinate. Die *x*-Koordinate belassen wir unverändert. Unsere neue Position liegt dann vier Zellen über unserem letzten Standort.

Wir haben diese Bewegungsart anstelle der bisher verwendeten **move()**-Methode gewählt, weil sie unabhängig davon ist, ob wir uns umdrehen. Die **move()**-Methode bewegt das Objekt in die Blickrichtung, wohingegen wir uns immer auf- und abbewegen möchten, ohne unsere Rotation zu verändern.

Konzept

Zugriffsmodifikatoren (privat oder öffentlich) bestimmen, wer eine Methode aufrufen kann.

Konzept

Private Methoden sind nur innerhalb der Klasse sichtbar, in der sie deklariert sind. Sie werden verwendet, um die Struktur des Codes zu verbessern.

Hinweis: private und public

Dir ist vermutlich aufgefallen, dass wir das Wort **private** in der Deklaration der Methode **checkKeyPress** benutzt haben:

```
private void checkKeyPress()
```

Diese Methode nennt man eine **private** Methode. Bisher hatten wir alle Methoden als **öffentlich** (*public*) deklariert.

Die Schlüsselwörter **private** und **public** werden **Zugriffsmodifikatoren** (*access modifier*) genannt. Sie legen fest, wer eine Methode sehen und aufrufen kann. Ist eine Methode öffentlich, dann kann sie von anderen Klassen in unserem Programm aufgerufen werden. Ist sie dagegen privat, so kann sie nur von Methoden innerhalb derselben Klasse aufgerufen werden.

Der Sinn von öffentlichen Methoden ist, anderen Teilen des Systems bestimmte Funktionen anzubieten, sodass unser Objekt aufgerufen werden kann, um eine Aufgabe zu erfüllen. Der Sinn von privaten Methoden ist, die Struktur unseres Codes zu verbessern, indem Aufgaben in kleinere Unteraufgaben aufgeteilt werden. Es ist nicht beabsichtigt, dass man sie von außen aufruft.

Von jetzt an werden wir Methoden als privat deklarieren, wenn sie nur für den internen Aufruf gedacht sind. Wir werden sie als öffentlich deklarieren, wenn sie von anderen Klassen aufgerufen werden sollen.

Variablen werden immer privat sein. (Java erlaubt es zwar, sie ebenfalls öffentlich zu machen, aber dies wird als sehr schlechter Stil angesehen.)

5.3 Bakterien: wie man sich selbst verschwinden lässt

Du hast gesehen, dass das Bakterien-Objekt von rechts nach links schwimmt und dabei langsam rotiert. Dafür sorgt die **act**-Methode von **Bacteria** (Listing 5.2).

Die erste Zeile benutzt zur Erzeugung der Bewegung dieselbe Methode wie die Klasse **WhiteCell**: **setLocation(x,y)** im Verbund mit **getX()** und **getY()**. Dieses Mal bewegen wir uns nach links, indem wir von der *x*-Koordinate subtrahieren. Auch hier können wir die Methode **move()** nicht verwenden, da wir die Bewegungsrichtung unabhängig von der Rotation halten möchten.

```
/**
 * Schwimmen in der Blutbahn, dabei langsam rotieren.
 */
public void act()
{
    setLocation(getX()-2, getY());
    turn(1);

    if (getX() == 0)
    {
        getWorld().removeObject(this);
    }
}
```

Listing 5.2
Die **act**-Methode von **Bacteria**.

Die zweite Zeile ist einfach: Wir rufen die Methode **turn** auf, um die langsame Rotation auszuführen.

Danach folgt eine **if**-Anweisung, um das Objekt aus der Welt zu entfernen, wenn es den linken Bildschirmrand erreicht. Wir können testen, ob wir den linken Rand erreicht haben, indem wir prüfen, ob die *x*-Koordinate 0 ist. Falls dies zutrifft, entfernen wir uns selbst mithilfe der folgenden Codezeile aus der Welt:

```
getWorld().removeObject(this);
```

Hier verwenden wir die Methode **removeObject** aus der **World**-Klasse. Ihre Signatur lautet

```
void removeObject(Actor object)
```

Wir können diese Methode aufrufen, um ein Objekt aus der Welt zu entfernen. Wir müssen uns über zwei Dinge im Klaren sein:

- Wir müssen ein Objekt als Parameter übergeben. Dies ist das Objekt, das aus der Welt entfernt wird.
- Diese Methode ist in der Klasse **World** definiert. Sie muss für ein **World**-Objekt aufgerufen werden.

Das Schlüsselwort this

Um ein Objekt als Parameter zu übergeben, verwenden wir das Java-Schlüsselwort **this**. Dieses Schlüsselwort verweist auf das aktuelle Objekt, das in diesem Moment ausgeführt wird. Wenn wir also sagen, dass „**this**" aus der Welt entfernt werden sollte, dann entfernt sich das Bakterium selbst.

Konzept

Das Schlüsselwort **this** kann verwendet werden, um auf das aktuelle Objekt zu verweisen.

Verketten von Methodenaufrufen

Die Methode **removeObject** gehört zu den Welt-Objekten, doch wir schreiben Code in der Klasse **Bacteria**. Daher können wir nicht einfach

```
removeObject(this);
```

aufrufen. Dies würde versuchen, die Methode für das **Bacteria**-Objekt aufzurufen, doch dieses Objekt besitzt keine derartige Methode.

Um eine Methode für andere Objekte aufzurufen, müssen wir zuerst das Objekt genauer bestimmen, dann einen Punkt schreiben und danach den Methodenaufruf:

```
my-world-object.removeObject(this);
```

Wir benötigen also Zugriff auf das Welt-Objekt – die Welt, in der wir uns gerade befinden. Zum Glück besitzt die **Actor**-Klasse eine Methode dafür. Sie heißt

```
getWorld()
```

und gibt eine Referenz auf das aktuelle Welt-Objekt zurück. Wir können diesen Methodenaufruf nun anstelle des Welt-Objekts einsetzen:

```
getWorld().removeObject(this);
```

Konzept

Über die Methode **getWorld** erhalten wir aus einem Akteur-Objekt heraus Zugriff auf die Welt.

Wir verketten hier zwei Methodenaufrufe: **getWorld()** und **removeObject(..)**. Zuerst wird **getWorld()** aufgerufen, diese Methode liefert uns das Welt-Objekt. Dann wird **removeObject(..)** für dieses Welt-Objekt aufgerufen, das wir gerade erhalten haben. Wir teilen also dem Welt-Objekt mit, dass es uns selbst aus der Welt entfernen soll.

5.4 Blutbahn: neue Objekte erzeugen

Die letzte Klasse, die wir uns ansehen wollen, ist unsere Welt-Unterklasse **Bloodstream**. Sie besitzt den üblichen Konstruktor sowie einen Aufruf der **prepare()**-Methode, um die Startobjekte (die weiße Blutzelle und die Wandverkleidung, die **Lining**-Objekte) zu platzieren.

Der interessante Teil dieser Klasse ist ihre **act**-Methode (Listing 5.3). Wenn das Szenario ausgeführt wird, platziert diese Methode neue **Bacteria**-Objekte in unregelmäßigen Abständen in der Welt. Jedes Mal, wenn eine Zufallszahl aus 100 kleiner als 3 ist – das heißt, im Durchschnitt dreimal pro 100 **act**-Schritte –, dann wird ein neues Bakterium erzeugt.

Listing 5.3: Die **act**-Methode von **Bloodstream**.

```
/**
 * Erzeugt neue herumschwimmende Objekte in unregelmäßigen Abständen.
 */
public void act()
{
    if (Greenfoot.getRandomNumber(100) < 3)
    {
        addObject(new Bacteria(), 779, Greenfoot.getRandomNumber(360));
    }
}
```

Die *x*-Koordinate für dieses neue Objekt ist stets 779 – der rechte Rand des Bildschirms (beachte, dass die Welt eine Größe von 780 × 360 Zellen hat, damit ist 779 die größtmögliche *x*-Koordinate).

Die *y*-Koordinate ist eine Zufallszahl aus 360 – die Höhe der Welt. Somit ist die *y*-Position des neuen Bakteriums eine zufällige Position an einer beliebigen Höhe. Wir haben all diese Konstrukte bereits früher gesehen: sie werden hier lediglich für einen neuen Zweck eingesetzt.

5.5 Seitliche Bewegung

Du solltest nun in der Lage sein, den Code in seiner jetzigen Form zu verstehen, und bist bereit, einige weitere Objekte und Funktionalitäten hinzuzufügen, um das Projekt interessanter zu machen. Wir werden dies in einer Reihe von Übungen tun, in denen wir einige der Konstrukte, die wir bereits kennen, anwenden.

Als Erstes möchten wir den Eindruck erzeugen, dass sich unsere weiße Blutzelle von links nach rechts durch die Blutbahn bewegt. Wir wollen jedoch die Blutzelle nicht wirklich bewegen, da wir sie stets auf dem Bildschirm behalten möchten. Stattdessen bewegen wir den Hintergrund – die umgebenden Wände der Blutbahn – nach links, um den Eindruck einer Bewegung zu erwecken. Dies ist eine häufig eingesetzte Technik in vielen sogenannten Side-Scroller-Spielen.

Dazu müssen wir drei Aufgaben erfüllen:

- die **`Lining`**-Objekte langsam nach links bewegen
- die **`Lining`**-Objekte löschen, wenn sie den linken Rand erreichen
- neue **`Lining`**-Objekte am rechten Rand auftauchen lassen

Übung 5.5 Bewege die **`Lining`**-Objekte kontinuierlich nach links. Sie sollen sich um eine Zelle pro **`act`**-Schritt weiterbewegen. Dies erreichst du, indem du die erste Zeile der **`act`**-Methode von **`Bacteria`** kopierst und die Bewegungsdistanz auf 1 änderst.

Übung 5.6 Lasse die **`Lining`**-Objekte verschwinden, wenn sie den linken Rand erreichen. Auch dafür findest du ein Beispiel in der **`Bacteria`**-Klasse, das du als Orientierungshilfe nutzen kannst.

Übung 5.7 Schreibe einen passenden Kommentar für die **`act`**-Methode.

Übung 5.8 Lasse neue **`Lining`**-Objekte am rechten Bildschirmrand auftauchen. Verwende hierzu die **`act`**-Methode von **`Bloodstream`**. Die Objekte sollten mit der Wahrscheinlichkeit von 1% erscheinen (durchschnittlich einmal pro 100 **`act`**-Schritte).

Für die letzte Übung kann die **`if`**-Anweisung in der **`act`**-Methode von **`Bloodstream`** zwei weitere Male kopiert werden: einmal für **`Lining`**-Objekte am oberen Bildschirmrand und einmal für Objekte am unteren Rand. Ändere die 3 in der **`if`**-

Anweisung in eine 1, um sie weniger häufig erscheinen zu lassen. Dann ändere die Klasse von Objekten, die erzeugt werden sollen, zu **Lining** und schließlich die *y*-Koordinate von einer Zufallszahl auf einen festen Wert: 0 für den oberen Rand, 359 für den unteren. Zum Beispiel fügt

```
addObject(new Lining(), 779, 359);
```

ein **Lining**-Objekt am unteren Rand des Bildschirms ein.

Falls es dir schwerfällt, kannst du alle Übungen, die wir hier besprechen, im Szenario *WBC-2* nachlesen, dort sind sie bereits implementiert.

5.6 Viren hinzufügen

Unsere Aufgabe in diesem Spiel wird es sein, die herumschwimmenden Bakterien zu fangen. Um die Herausforderung ein bisschen zu erhöhen, wollen wir eine Gefahr einbauen: Viren. Viren schwimmen ebenfalls durch die Blutbahn, doch unsere Zelle muss ihnen ausweichen.

Abbildung 5.2
Ein Virus.

Übung 5.9 Erzeuge eine neue Klasse für einen Virus in deinem Szenario (als Unterklasse von **Actor**) mit dem Namen **Virus**. Du findest ein vorbereitetes Bild im Szenario, das du direkt verwenden kannst (Abbildung 5.2).

Übung 5.10 Erweitere die **act**-Methode von **Bloodstream**, um neue **Virus**-Objekte am rechten Bildschirmrand hinzuzufügen. Die *y*-Koordinate soll zufällig gewählt sein. Setze die Wahrscheinlichkeit von neuen Objekten auf 1% (einmal in 100 **act**-Schritten). Teste deinen Code. (**Virus**-Objekte sollten gelegentlich am rechten Rand auftauchen, sich aber noch nicht bewegen.)

Übung 5.11 Lasse die **Virus**-Objekte eine Linksbewegung und eine Rotation ausführen (wie Bakterien). Die Viren sollen sich jedoch vier Zellen pro **act**-Schritt (nicht zwei) bewegen und gegen den Uhrzeigersinn rotieren.

Übung 5.12 Kommentiere deine **Virus**-Klasse. (Das heißt: fülle den Klassenkommentar ganz oben sowie den Methodenkommentar über der **act**-Methode aus.)

5.7 Kollision: Bakterien entfernen

Als Nächstes wollen wir überprüfen, ob wir Bakterien berühren (und diese entfernen, falls das zutrifft) oder auf einen Virus treffen (in diesem Fall haben wir verloren und das Spiel ist vorbei). Dies gleicht dem, was wir mit der Krabbe, den Würmern und den Hummern gemacht haben.

Übung 5.13 Erzeuge eine neue private Methode in der Klasse **`WhiteCell`** mit Namen **`checkCollision`**. Der Methodenrumpf kann anfänglich leer sein. Rufe diese Methode aus deiner **`act`**-Methode auf.

Übung 5.14 Füge Code in der Methode **`checkCollision`** ein, der Bakterien entfernt, wenn wir sie berührt haben. (Verwende die Methoden **`isTouching()`** und **`removeTouching(..)`** analog dem Krabben-Szenario.)

Übung 5.15 Spiele einen Sound ab, wenn Bakterien entfernt werden. Den Sound „slurp.wav" findest du auch wieder im Szenario – du kannst ihn verwenden.

Übung 5.16 Denselben Soundeffekt noch einmal zu benutzen, ist ein bisschen zu bequem. Erzeuge und verwende einen neuen Sound für das Löschen von Bakterien.

Übung 5.17 Füge wieder ähnlichen Code zur Methode **`checkCollision`** hinzu (eine weitere **`if`**-Anweisung), um zu prüfen, ob wir einen Virus berühren. Falls ja, spiele einen Sound ab (es gibt einen Sound namens „gameover.wav" für diesen Zweck) und beende die Ausführung.

Eine Lösung für all diese Übungen bis hierher ist in der Szenario-Version *WBC-2* enthalten.

Hinweis

Bei genauem Hinsehen hast du vielleicht bemerkt, dass du manchmal verlierst und das Spiel beendet ist, wenn die weiße Blutzelle nur in der Nähe eines Virus ist, diesen aber noch gar nicht berührt hat. Es ist noch ein kleiner Abstand zwischen den Objekten.

Dies liegt an der Art und Weise, wie Bilder in Computern gespeichert werden. Wir besprechen dieses Thema ausführlich in ***Kapitel 9***. Wenn du neugierig bist, wirf einen Blick auf ***Abbildung 9.2*** und die dazugehörige Erklärung.

5.8 Variable Geschwindigkeit

Damit die Bewegung interessanter als bisher aussieht (und dadurch das Spielen interessanter wird), möchten wir die Bewegungsgeschwindigkeit der Bakterien variabel gestalten. Im Moment ist die Geschwindigkeit 2 (Bakterien bewegen sich in jedem **`act`**-Schritt zwei Zellen weiter). Wir möchten dies so verändern, dass die Bewegungsgeschwindigkeit einen zufälligen Wert zwischen 1 und 3 bekommt.

Dazu werden wir eine Variable für die Geschwindigkeit definieren, diese mit dem entsprechenden Zufallswert initialisieren und in unserer Bewegungsanweisung verwenden.

Übung 5.18 Füge in deiner **Bacteria**-Klasse eine Instanzvariable vom Typ **int** mit Namen **speed** hinzu.

Übung 5.19 Weise der Variable **speed** im **Bacteria**-Konstruktor einen Zufallswert aus dem Bereich 1 bis 3 zu.

Du fragst dich vielleicht, wie man einen Zufallsbereich von 1 bis 3 festlegt, wo wir doch früher gelernt haben, dass Zufallszahlen, die von der Greenfoot-Methode **getRandomNumber** kommen, immer bei 0 anfangen. Die Antwort ist einfach: Nimm eine Zahl von 0 bis 2 und addiere dann 1:

```
speed = Greenfoot.getRandomNumber(3) + 1;
```

Die Addition der 1 ist wichtig, weil wir niemals eine Geschwindigkeit von 0 haben möchten – das Bakterium würde sich dann überhaupt nicht fortbewegen.

Übung 5.20 Ändere die Anweisung für die Bewegung in deiner **act**-Methode, sodass du die Variable **speed** von deiner *x*-Koordinate subtrahierst, anstatt zwei abzuziehen. Teste deinen Code.

Das ist auch schon alles, was dazu nötig ist. Jedes Bakterium bekommt nun seine eigene Geschwindigkeit zwischen 1 und 3 zugewiesen, wenn es erzeugt wird, und bewegt sich mit dieser Geschwindigkeit. Beobachte den Ablauf des Spiels nun, du solltest sehen können, dass sich einige Bakterien schneller als andere bewegen.

5.9 Rote Blutzellen

Nun wollen wir unser Blutgemisch noch durch rote Blutzellen abrunden. Dieser Schritt ist rein kosmetisch: die anderen Objekte interagieren nicht mit den roten Blutzellen und diese beeinflussen das Spiel nicht. Aber es sieht gut aus!

Übung 5.21 Füge eine neue Klasse namens **RedCell** hinzu. Du findest ein Bild dafür in deinem Szenario.

Übung 5.22 Gestalte die Bewegung der roten Blutzellen genau wie die der Bakterien, das heißt: Sie bewegen sich von rechts nach links mit variabler Geschwindigkeit und rotieren dabei langsam. Es gibt einen kleinen Unterschied: Die Geschwindigkeit der roten Blutzellen liegt nur zwischen 1 und 2 (anstatt zwischen 1 und 3). Unsere roten Blutzellen sind langsam.

Übung 5.23 Erweitere deine **act**-Methode in der Klasse **Bloodstream**, um rote Blutzellen zu erzeugen. Das ähnelt dem Prozess der Erzeugung von Bakterien oder Viren, doch rote Blutzellen sollen häufiger vorkommen: Lass sie mit einer Wahrscheinlichkeit von 6% auftreten (6 von jeden 100 **act**-Schritten).

Das sieht besser aus! Es wird langsam ein wenig eng in unserer Blutbahn, aber das ist gut so.

Bevor wir zu unserer nächsten Aufgabe übergehen, wollen wir noch eine kleine Verbesserung vornehmen. Rote Blutzellen haben, wenn sie erzeugt werden, alle dieselbe Rotation (sie werden alle mit Rotation 0 erzeugt). Deshalb rotieren sie alle gleichförmig und nahe beieinander. Dies sieht ein bisschen zu sehr nach Synchronschwimmen aus und erscheint uns nicht zufällig genug.

Um dies zu beheben, wollen wir die roten Blutzellen mit einer zufälligen Rotation starten lassen. Dazu fügen wir eine Zuweisung im Konstruktor der roten Blutzellen ein, um die Rotation auf eine Zufallszahl zu setzen:

```
setRotation(Greenfoot.getRandomNumber(360));
```

Übung 5.24 Initialisiere deine roten Blutzellen mit einer zufälligen Rotation mithilfe der oben gezeigten Anweisung.

5.10 Begrenzungen hinzufügen

Wir werden noch eine letzte kosmetische Verbesserung vornehmen, bevor wir uns dem interessanteren neuen Material zuwenden: Begrenzungen an den Bildschirmrändern hinzufügen.

Der Hauptgrund dafür ist, dass wir einen unschönen Effekt von Greenfoot umgehen wollen: In Greenfoot werden Objekte positioniert, indem man die Koordinate des Mittelpunkts des Objektbildes angibt. Daher erscheint die erste Hälfte des Objektbildes, wenn wir es am rechten Bildschirmrand platzieren, mit einem Mal, anstatt langsam von der Seite einzuschweben. Dasselbe Problem haben wir auf der anderen Seite, wenn die Objekte verschwinden: Sie sind schlagartig weg, sobald die erste Hälfte ihres Bildes den Bildschirm verlassen hat. Dadurch wirkt das Erscheinen und Verschwinden unserer Objekte ein wenig ruckartig. Es sieht auf jeden Fall nicht sehr schön aus.

Um dieses Problem zu beheben, verwenden wir einen einfachen Trick: wir setzen schwarze Grenzobjekte links und rechts an die Bildschirmränder, um den problematischen Raum zu verdecken. Wir beabsichtigen, das Ganze so aussehen zu lassen, als betrachte man die Blutbahn durch ein Mikroskop (Abbildung 5.1). Die Objekte werden nun hinter diesen Begrenzungen erscheinen, außer Sicht, und dann langsam in das Blickfeld einschweben.[1]

Übung 5.25 Erzeuge eine neue **Actor**-Unterklasse mit Namen **Border**. Es gibt dafür bereits ein Bild.

Wir könnten nun ein **Border**-Objekt auf der linken und rechten Bildschirmseite von Hand einfügen und die Funktion DIE WELT SPEICHERN benutzen, um sie dort zu sichern. Wir möchten die Position jedoch punktgenau festlegen und das ist manuell schwierig zu bewerkstelligen. Es ist einfacher, Code zur Positionierung der Objekte zu schreiben.

In der **prepare()**-Methode von **Bloodstream** können wir den folgenden Code einfügen:

```
Border border = new Border();
addObject(border, 0, 180);
Border border2 = new Border();
addObject(border2, 770, 180);
```

Übung 5.26 Füge den obigen Code in der **prepare()**-Methode von **Bloodstream** ein. Teste den Code. Kannst du noch weitere Probleme beobachten?

Wir müssen zwei kleinere Probleme beheben: Erstens ist unsere weiße Blutzelle nun teilweise verdeckt und zweitens erscheinen unsere Objekte oberhalb der Begrenzung anstatt darunter. Diese beiden Fehler räumen wir nun aus.

Übung 5.27 Finde in der **prepare()**-Methode von **Bloodstream** die Stelle, an der die anfängliche *x*-Koordinate für die weiße Blutzelle definiert ist. Ändere sie auf einen größeren Wert, sodass die Zelle ein bisschen weiter rechts ist.

Übung 5.28 Füge im Konstruktor von **Bloodstream** die folgende Anweisung ein:

```
setPaintOrder(Border.class);
```

Dies wird dazu führen, dass die Begrenzungsobjekte oberhalb der anderen Objekte erscheinen. Teste dann den Code.

1 Es gibt noch eine andere Möglichkeit, dieses Problem anzugehen: Wir könnten eine *unbegrenzte Welt* erzeugen, die es uns erlauben würde, Akteure außerhalb der Weltgrenzen zu platzieren und dann langsam hereinkommen zu lassen. Dazu wird eine weiterer Unterklassen-Konstruktor von **World** aufgerufen, der in der Klassendokumentation von **World** beschrieben ist. Dies wird jedoch ein wenig komplizierter, deshalb haben wir hier einen anderen Weg beschritten.

Mithilfe der Methode **setPaintOrder** können wir festlegen, welche Objekte über welchen anderen Objekten gezeichnet werden sollen. Die Methode besitzt eine Parameterliste von veränderlicher Länge und wir dürfen so viele Klassen, wie wir möchten, spezifizieren. Zum Beispiel:

```
setPaintOrder(Class1.class, Class2.class, Class3.class);
```

In diesem Beispiel würden Objekte von **Class1** ganz oben gezeichnet, Objekte von **Class2** darunter und Objekte von **Class3** wieder darunter. Alle Objekte von Klassen, die hier nicht erwähnt sind, werden darunter in nicht festgelegter Reihenfolge dargestellt.

Indem wir schreiben

```
setPaintOrder(Border.class);
```

legen wir fest, dass **Border**-Objekte immer oben sein sollten (und uns ist es egal, in welcher Reihenfolge die anderen Objekte gezeichnet werden).

Übung 5.29 Gebe den weißen Blutzellen die Fähigkeit, sich nicht nur nach oben und unten, sondern auch nach rechts und links zu bewegen.

5.11 Zu guter Letzt: Punktezählung einbauen

Als Nächstes wollen wir ein paar Regeln zum Gewinnen von Spielpunkten einführen und diese implementieren. Die erste offensichtliche Idee ist, dass wir Punkte für das Neutralisieren von Bakterien bekommen sollten: jedes Bakterium, das wir fangen, soll uns 20 Punkte einbringen.

Im ersten Schritt implementieren wir dies in der **WhiteCell**-Klasse. Wir werden eine Integer-Variable benötigen, die unseren Punktestand speichert, die Punkte erhöht, wenn wir ein Bakterium erwischen, und den Punktestand auf dem Bildschirm anzeigt.

Übung 5.30 Füge in deine **WhiteCell**-Klasse eine Instanzvariable vom Typ **int** mit Namen **score** ein.

Übung 5.31 Füge eine Anweisung hinzu, um den Punktestand um 20 Punkte zu erhöhen, wenn wir ein Bakterium fangen.

Im vorigen Kapitel haben wir gesehen, wie man Instanzvariablen hinzufügt. Wir schreiben

```
private int score;
```

an den Anfang der Klasse, oberhalb der ersten Methode.

Wir können dann 20 Punkte zu unserem bisherigen Stand addieren, indem wir die folgende Anweisung direkt nach dem Entfernen des **Bacteria**-Objekts benutzen:

```
score = score + 20;
```

Was noch zu tun bleibt, ist den Punktestand auf dem Bildschirm anzuzeigen. Dazu verwenden wir eine Methode namens **showText(..)** aus der **World**-Klasse.

Übung 5.32 Suche die Methode **showText(..)** in der **World**-Klasse. Wie viele Parameter hat die Methode? Wie lauten diese?

Auch hier müssen wir wieder eine Methode der Welt-Klasse aufrufen, wie wir es schon früher getan haben, um ein Objekt aus der Welt zu entfernen. Wir verwenden dieselbe Technik wie vorher: Wir rufen **getWorld()** auf, um Zugriff auf die Welt-Objekte zu erhalten, dann fügen wir unsere **showText(..)**-Methode am Ende an:

```
getWorld().showText(..);
```

Jetzt müssen noch die Parameter für die **showText**-Methode berechnen. Die Signatur der Methode lautet

```
void showText(String text, int x, int y)
```

Konzept

Der Typ **String** wird verwendet, um Text zu repräsentieren, zum Beispiel Wörter oder Sätze. Strings werden in doppelte Anführungszeichen gesetzt.

Die letzten beiden Parameter sind einfach: es sind die *x*- und *y*-Koordinaten der Stelle, an der wir den Text anzeigen wollen. Der erste Parameter ist der Text, den wir anzeigen möchten. Er gehört zu einem Typ, den wir zwar schon gesehen, aber bisher noch nicht richtig besprochen haben: *String*.

Variablen vom Typ String können Text speichern, zum Beispiel Zeichen, Wörter oder Sätze. Strings werden mit doppelten Anführungszeichen geschrieben:

```
String name = "Fred";
String message = "Game over";
```

Wir sind bereits einem String begegnet, als wir einen Sound abgespielt haben:

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

Konzept

Variablen vom Typ **String** können String-Objekte speichern.

Die Methode **playSound** erwartet einen String als Parameter und der Wert **"slurp.wav"** ist ein String, den wir an die Methode übergeben. Dasselbe können wir mit der Methode **showText** machen. Zum Beispiel wird durch

```
getWorld().showText("Hallo", 80, 25);
```

das Wort „Hallo" auf dem Bildschirm an der angegebenen Position dargestellt.

Übung 5.33 Füge die oben stehende Anweisung ein, sodass das Wort „Hallo" auf dem Bildschirm erscheint, wenn du ein Bakterium erwischt hast.

Übung 5.34 Ändere die Anweisung so, dass sie nun deinen Namen ausgibt.

Übung 5.35 Ändere den Text noch einmal, um das Wort „Punkte:" oder „Score:" (einschließlich Doppelpunkt) anzuzeigen.

String-Verknüpfung

Konzept

String-Verknüpfung oder **String-Konkatenation** verbindet zwei Strings zu einem String. Diese Operation wird mithilfe des Pluszeichens (+) geschrieben.

Wir haben gesehen, wie wir einen Text auf dem Bildschirm anzeigen können – jetzt müssen wir nur noch herausfinden, wie wir unseren Punktestand ebenfalls darstellen, der ja in einer **int**-Variable gespeichert ist. Wir werden dazu *String-Verknüpfung* (auch *String-Konkatenation* genannt) verwenden. Eine String-Verknüpfung wird mit einem Pluszeichen (+) geschrieben und verbindet zwei Strings zu einem:

```
"abc" + "def"
```

wird zu

```
"abcdef"
```

und

```
"Wolfgang" + "Amadeus" + "Mozart"
```

wird zu

```
"WolfgangAmadeusMozart"
```

(Beachte, dass die String-Verknüpfung nicht automatisch Leerzeichen einfügt. Wenn du ein Leerzeichen zwischen zwei Teilen haben möchtest, dann musst du es explizit hinschreiben.)

Konzept

Die **String-Verknüpfung** kann auch zwischen String und Integer angewandt werden.

Die Operation kann auch zwischen einer String- und einer Integer-Variablen angewandt werden. Wenn wir einen String und einen Integerwert „addieren", dann wird der Integerwert in einen String umgewandelt und dann kann verknüpft werden:

```
"Punkte: " + 20
```

wird zu

```
"Punkte: 20"
```

Wir können auch eine Variable anstelle des Integerwerts einsetzen:

```
"Punkte: " + score
```

Hier wird der Wert, der in unserer **score**-Variablen gespeichert ist, in einen String umgewandelt, dann wird dieser mit dem String **"Punkte: "** verbunden. Nun haben wir alles, was wir benötigen, um unseren Punktestand auf dem Bildschirm anzuzeigen. Alle Elemente zusammen sind in Listing 5.4 zu sehen.

Listing 5.4: Punktezählung in der Klasse **WhiteCell**.

```
// Importanweisung und Kommentar ausgelassen
public class WhiteCell extends Actor
{
    private int score;
    // weitere Methoden ausgelassen

    /**
    * Prüft, ob wir ein Bakterium oder einen Virus berühren. Entfernt das Bakterium.
    * Das Spiel ist vorbei, wenn wir auf einen Virus treffen.
    */
    private void checkCollision()
    {
        if (isTouching(Bacteria.class))
        {
            Greenfoot.playSound("slurp.wav");
            removeTouching(Bacteria.class);
        }

        if (isTouching(Virus.class))
        {
            Greenfoot.playSound("game-over.wav");
            Greenfoot.stop();
        }
    }
}
```

Übung 5.36 Zeige den Punktestand in deinem Spiel an.

Übung 5.34 Das Spiel scheint im Moment ein wenig zu einfach zu sein. Beschleunige es! Vielleicht möchtest du einige der folgenden Werte ausprobieren: Bakterien haben eine zufällige Geschwindigkeit (Zellen pro **act**-Schritt) zwischen 1 und 5. Die weißen Blutzellen bewegen sich seitlich mit einer Geschwindigkeit von 4, hoch und runter mit einer Geschwindigkeit von 8. Viren bewegen sich mit Geschwindigkeit 8. Beschleunige auch das gesamte Szenario mithilfe des Geschwindigkeitsreglers unter dem Hauptfenster: setze es knapp über 50 Prozent. Experimentiere mit den Geschwindigkeitswerten, damit das Spiel anspruchsvoll, aber nicht unmöglich wird.

5.12 Punktezählung in der Welt

Bis hierhin war die Punktezählung relativ unkompliziert. Es gibt jedoch einen Nachteil dabei, die Punktezählung in der **WhiteCell**-Klasse durchzuführen: Punkte für andere Ereignisse aufzunehmen ist schwierig.

Nehmen wir beispielsweise an, unsere vollständigen Regeln zur Punktezählung sollten so aussehen:

- Das Neutralisieren eines Bakteriums bringt uns 20 Punkte (diese Regel haben wir eingearbeitet).
- Wir verlieren 15 Punkte für jedes Bakterium, das den linken Bildschirmrand erreicht.
- Von einem Virus getroffen zu werden, beendet das Spiel nicht direkt, stattdessen verlieren wir 100 Punkte.
- Wenn unser Punktestand unter 0 fällt, verlieren wir und das Spiel ist zu Ende.

Dies sind etwas interessantere Regeln. Wir haben allerdings ein Problem dabei. Das Eintreten des zweiten Ereignisses (ein Bakterium entkommt) wird in einem **Bacteria**-Objekt festgestellt. Von diesem aus haben wir aber keinen Zugriff auf unsere **score**-Variable, die sich ja im **WhiteCell**-Objekt befindet. Wir könnten versuchen, Zugriff auf das **WhiteCell**-Objekt zu bekommen, um den Punktestand zu aktualisieren. Es ist jedoch einfacher, die **score**-Variable in unsere World-Unterklasse (**Bloodstream**) zu verlegen, da alle anderen Objekte leicht auf dieses Objekt zugreifen können (mithilfe der **getWorld()**-Methode).

Dann befindet sich der aktuelle Punktstand im **Bloodstream**-Objekt und die **WhiteCell**- und **Bacteria**-Objekte rufen das **Bloodstream**-Objekt auf, um ihm nötigenfalls mitzuteilen, die Punkte zu aktualisieren.

Los geht's!

Übung 5.38 Verschiebe die **score**-Variable von **WhiteCell** zu **Bloodstream**.

Übung 5.39 Füge eine Zeile zum **Bloodstream**-Konstruktor hinzu, um den Punktestand mit 0 zu initialisieren. Dies ist nicht unbedingt notwendig, weil der standardmäßige Wert für Instanzvariablen 0 ist, aber dieses Vorgehen hat sich bewährt.

Übung 5.40 Füge eine öffentliche Methode namens **addScore()** zur **Bloodstream**-Klasse hinzu. Verschiebe den Code, der den Punktestand inkrementiert und auf dem Bildschirm anzeigt, in diese Methode. Beachte, dass du nun den Aufruf von **getWorld()** am Anfang des **showText(..)**-Aufrufs auslassen kannst, da wir uns ja bereits in der Welt befinden (Listing 5.5).

Übung 5.41 Kommentiere deine neue Methode.

```
public void addScore()
{
    score = score + 20;
    showText("Score: " + score, 80, 25);
}
```

Listing 5.5
Der erste Versuch einer **addScore**-Methode.

Zuletzt müssen wir das **WhiteCell**-Objekt dazu bringen, das Welt-Objekt zu benachrichtigen, wenn der Punktestand aktualisiert werden soll. Ein erster – leider falscher – Versuch könnte so aussehen:

```
getWorld().addScore();
```

Dies scheint eine vernünftige Idee zu sein, die aber leider nicht funktioniert, und zwar aufgrund eines verzwickten Problems: Die **getWorld()**-Methode, wie sie in der **World**-Klasse von Greenfoot definiert ist, gibt ein Objekt vom Typ **World** zurück. Und **World** besitzt keine **addScore()**-Methode. *Unser* Welt-Objekt ist tatsächlich ein **Bloodstream**-Objekt und *hat* diese Methode.

Wir müssen eine sogenannte *Typanpassung* (im Englischen auch *Casting* genannt) durchführen, um dem Greenfoot-Compiler mitzuteilen, dass unsere Welt vom Typ **Bloodstream** ist, die in einer **Bloodstream**-Variablen gespeichert wird. Danach können wir die **addScore()**-Methode aufrufen.

Eine Typanpassung schreiben wir auf, indem wir den Typ (**Bloodstream**) in Klammern vor den Methodenaufruf setzen:

```
Bloodstream bloodstream = (Bloodstream)getWorld();
bloodstream.addScore();
```

Typanpassung ist ein kompliziertes Konzept und du wirst es zu diesem Zeitpunkt wahrscheinlich nicht vollends verstehen – keine Sorge, wir werden darauf zurückkommen und es später (in ***Abschnitt 9.6***) ausführlicher besprechen. Dann kennen wir auch schon ein paar mehr Einzelheiten zum Thema Typisierung.

Übung 5.42 Füge den Code zum Aufruf der **addScore()**-Methode von **Bloodstream** zur **WhiteCell**-Klasse hinzu. Teste deinen Code. Dein Szenario sollte nun wieder laufen und das Fangen von Bakterien sollte Punkte bringen.

Oberflächlich betrachtet sind wir wieder dort, wo wir bereits waren: das Fangen von Bakterien bringt Punkte und dieser Punktestand wird angezeigt, wenn er sich ändert. Das Verschieben des Codes zur Punktezählung von der **WhiteCell** zu **Bloodstream** hat – bis hierhin – keine sichtbaren Auswirkungen. Doch wir haben nun einen großen Vorteil: wir können jetzt auch Punkte für andere Ereignisse vergeben.

5.13 Abstraktion: Punktezählung generalisieren

Unsere nächste Aufgabe besteht darin, die zweite Punkteregel zu implementieren: Wir verlieren 15 Punkte, wenn ein Bakterium entkommt, indem es den linken Bildschirmrand erreicht.

Jetzt ist es einfach, die **addScore()**-Methode von **Bloodstream** aufzurufen: Wir können denselben Code in die Klasse **Bacteria** schreiben, den wir in der Klasse **WhiteCell** benutzt haben. Dies würde jedoch dazu führen, etwas (20 Punkte) zum Punktestand hinzuzufügen, anstatt etwas abzuziehen.

Eine Möglichkeit besteht darin, eine neue Methode in **Bloodstream** einzubauen, die beispielsweise **losePoints()** heißt und die 15 Punkte vom Punktestand abzieht, wenn wir sie aufrufen. Obwohl dies durchaus funktionieren würde, wollen wir eine elegantere Lösung vorziehen: Wir generalisieren unsere **addScore()**-Methode, sodass sie eine beliebige Anzahl von Punkten addieren oder subtrahieren kann.

Dazu ändern wir die **addScore**-Methode so ab, dass sie als Eingabewert die aktuelle Punktezahl erwartet, die addiert werden sollen (Listing 5.6).

Listing 5.6: Die Methode **addScore** mit einem Parameter.

```
/**
 * Fügt einen Punktestand hinzu.
 */
public void addScore(int points)
{
    score = score + points;
    showText("Score: " + score, 80, 25);
}
```

Unsere verbesserte **addScore**-Methode bekommt jetzt die Punktezahl, die zum Punktestand addiert werden sollen, als Parameter übergeben. An der Stelle, wo der neue Punktestand zugewiesen wird, addieren wir nun den Parameter **points**, anstatt einfach 20 zu addieren.

Nachdem wir unsere neue **addScore**-Methode definiert haben, können wir diese verwenden, um unterschiedliche Punktestände zu addieren:

```
bloodstream.addScore(20);
```

würde 20 Punkte hinzuzählen, während

```
bloodstream.addScore(100);
```

100 Punkte hinzufügt. Wir können auch leicht Punkte subtrahieren, indem wir eine negative Zahl verwenden:

```
bloodstream.addScore(-15);
```

Was wir gerade gemacht haben, ist ein Beispiel einer *Abstraktion* – ein wichtiges Konzept in der Programmierung. Anstatt eine Methode zu schreiben, die eine bestimmte Sache macht – *20 Punkte hinzuzählen* –, schreiben wir eine Methode, die viele ähnliche Dinge tun kann – *eine beliebige Anzahl von Punkten hinzufügen* –, sodass sie in unterschiedlichen Situationen eingesetzt werden kann.

Konzept

Die **Parametrisierung** von Methoden (das Hinzufügen von Parametern) kann diese flexibler und breiter einsetzbar machen.

Methoden zu generalisieren, indem man Parameter hinzufügt, sodass sie flexibler sind, ist im Allgemeinen eine gute Idee.

Übung 5.43 Ändere deine Methode **addScore** dahingehend, dass sie einen Parameter für die Punkte erwartet, wie oben besprochen.

Übung 5.44 Ändere den Code in der Klasse **WhiteCell** entsprechend: Übergib einen Parameter an **addScore**, sodass das Neutralisieren eines Bakteriums wieder 20 Punkte einbringt.

Übung 5.45 Füge Code in der **Bacteria**-Klasse hinzu, sodass du 15 Punkte verlierst, wenn ein Bakterium den Bildschirm verlässt.

Wenn du den Code zur Punktezählung der **Bacteria**-Klasse hinzufügst, ist es wichtig, nicht wieder **getWorld()** aufzurufen, nachdem das Objekt aus der Welt entfernt wurde. Denn in diesem Fall kann ein Aufruf von **getWorld()** nicht funktionieren, da wir uns nicht mehr in einer Welt befinden. (Er würde dann den speziellen Wert **null** zurückgeben und der Versuch, auf die Welt zuzugreifen, hätte einen Fehler zur Folge, der **NullPointerException** heißt.) Im Code von Listing 5.7 wurde darauf geachtet, dass dieser Fehler nicht auftritt. Wir sehen hier auch, dass wir nun die Variable **bloodstream** verwenden können, um die Methode **removeObject** aufzurufen (anstatt noch einmal **getWorld()**). Wir haben die Welt bereits in unserer lokalen Variable gespeichert, deshalb müssen wir sie nicht erneut holen.

Listing 5.7: Bakterien: erst Punkte verlieren, dann verschwinden.

```
/**
 * Schwimmen in der Blutbahn, dabei langsam rotieren.
 */
public void act()
{
    setLocation(getX()-speed, getY());
    turn(1);

    if (getX() == 0)
    {
        Bloodstream bloodstream = (Bloodstream)getWorld();
        bloodstream.addScore(-15);
        bloodstream.removeObject(this);
    }
}
```

Ein weiterer Schönheitsfehler ist, dass der Punktestand erst auf dem Bildschirm erscheint, wenn wir die ersten Punkte erhalten haben. Es wäre wirklich hübscher, wenn wir die Punkte von Anfang an anzeigen könnten. Die folgenden Übungen sollen hier Abhilfe schaffen.

Übung 5.46 Erstelle in **Bloodstream** eine neue private Methode mit Namen **showScore()**. Diese hat keine Parameter und keinen Rückgabewert. Verschiebe die **showText**-Anweisung, die für die Anzeige des Punktestands zuständig ist, in diese Methode. Rufe **showScore** an der Stelle auf, wo vorher **showText** aufgerufen wurde.

Übung 5.47 Füge im Konstruktor von **Bloodstream** einen Aufruf von **showScore** hinzu. Dies wird zur Folge haben, dass der Punktestand am Anfang angezeigt wird.

Wir lassen jetzt nicht nur den Punktestand durch den Konstruktor anzeigen, sondern haben auch eine eigene Methode dafür angelegt. Es lohnt sich immer, für eine Aufgabe, die mehr als einmal ausgeführt werden muss, eine eigens darauf abgestimmte Methode zu schreiben, selbst wenn die Aufgabe kurz ist.

Wir sind nun bereit, unsere letzten beiden Regeln für die Punktezählung zu implementieren.

Übung 5.48 Ändere dein Programm, sodass die Berührung mit einem Virus nicht direkt das Spiel beendet. Stattdessen wird das Virus entfernt und du verlierst 100 Punkte. (Beachte: es ist wichtig, das Virus zu entfernen. Andernfalls würden wir ihn ständig berühren und in jedem ACT-Schritt erneut 100 Punkte verlieren.)

Übung 5.49 Erstelle einen neuen Sound für die Berührung mit einem Virus.

Übung 5.50 Verschiebe die Funktionen, die am Ende des Spiels ausgeführt werden (den Spiel-Ende-Sound abspielen und Greenfoot anhalten) in die **addScore**-Methode, sodass das Spiel beendet ist, sobald der Punktestand kleiner als 0 wird. (Du benötigst eine **if**-Anweisung hinter dem Ändern und Anzeigen des Punktestands.)

Wenn dir diese Aufgaben schwerfallen: das Szenario *WBC-4* zeigt die vollständige Implementierung (ebenso wie die zusätzlichen Änderungen, die wir noch besprechen werden). Versuche es jedoch zuerst selbst – eigentlich müsstest du alles wissen, was du dazu brauchst.

5.14 Spielzeit hinzufügen

Jetzt fehlt nur noch eine Möglichkeit, das Spiel auch zu gewinnen.

Dazu werden wir einen Zeitzähler einführen, der rückwärts zählt – wenn wir durchhalten, bis der Zähler abgelaufen ist, dann haben wir gewonnen.

Zum Einbau des Zeitzählers können wir ganz ähnliche Konstrukte wie beim Einsetzen des Punktestands verwenden. Wir sollten also in der Lage sein, dies in der folgenden Übungsreihe vorzunehmen.

Übung 5.51 Füge in der Klasse **`Bloodstream`** eine Instanzvariable vom Typ **`int`** mit dem Namen **`time`** hinzu.

Übung 5.52 Initialisiere im Konstruktor die Zeit auf 2000 (wir möchten versuchen, 2000 ACT-Schritte zu überstehen).

Übung 5.53 Definiere eine neue private Methode namens **`showTime`**, die die verbleibende Zeit (die Variable **`time`**) am rechten oberen Bildschirmrand darstellt. Rufe diese Methode aus dem Konstruktor heraus aus, um die Anfangszeit anzuzeigen.

Übung 5.54 Definiere eine neue private Methode namens **`countTime`**, die die Zeit bei jedem Aufruf um 1 dekrementiert und dann die aktuelle Zeit anzeigt (durch Aufruf von **`showTime`**). Rufe diese Methode aus vom Konstruktor aus deiner **`act`**-Methode heraus auf.

Übung 5.55 Füge eine **`if`**-Anweisung in deiner **`countTime`**-Methode hinzu, die die Ausführung beendet, wenn der Zähler 0 erreicht.

Übung 5.56 Füge eine neue private Methode namens **`showEndMessage`** hinzu. Wenn diese Methode aufgerufen wird, zeigt sie ungefähr in der Bildschirmmitte eine Nachricht an, die uns mitteilt, dass wir gewonnen haben und wie viele Punkte wir erreicht haben. Zum Beispiel

> Zeit ist abgelaufen – du hast gewonnen!
>
> Dein Endstand ist: 1455 Punkte

Rufe diese Methode auf, wenn der Zeitzähler abläuft und wir gewinnen.

Zusammenfassung der Programmiertechniken

In diesem Kapitel konnten wir weitere Praxis im Umgang mit wichtigen Konzepten sammeln, die wir wirklich kennen müssen: Variablen verwenden und Methoden aufrufen.

Wir haben auch ein erstes Beispiel von Objektinteraktion gesehen: Unser Akteur-Objekt hat unser Welt-Objekt aufgerufen und gebeten, eine Aktion auszuführen (den Punktestand ändern oder ein Objekt entfernen). Wir haben den Typ String kennengelernt und die String-Verknüpfung.

Am wichtigsten war jedoch unsere erste Begegnung mit dem Konzept der Abstraktion zur Verallgemeinerung: Wir haben gesehen, dass durch das Hinzufügen eines Parameters zu einer Methode diese Methode breiter anwendbar wird. Wir können dann diese Methode aus unterschiedlichen Kontexten heraus aufrufen, um leicht unterschiedliche (aber verwandte) Aufgaben auszuführen. Dies ist besser, als für jeden Fall eine eigene Methode zu schreiben.

Wir werden Abstraktion während der nächsten Kapitel noch häufiger antreffen.

Zusammenfassung der Konzepte

- Die Methode **`SetLocation`** positioniert einen Akteur an der Stelle, die durch die *x*- und *y*-Koordinaten bestimmt wird.
- **Zugriffsmodifikatoren** (privat oder öffentlich) legen fest, wer eine Methode aufrufen darf.
- **Private Methoden** sind nur innerhalb der Klasse sichtbar, in der sie deklariert sind. Sie werden verwendet, um die Struktur des Codes zu verbessern.
- Das Schlüsselwort **`this`** kann benutzt werden, um auf das aktuelle Objekt zu verweisen.
- Die Methode **`getWorld`** ermöglicht es uns, auf die Welt von einem Akteur-Objekt aus zuzugreifen.
- Der Typ **String** wird verwendet, um Text zu repräsentieren, wie Wörter oder Sätze. Strings werden in doppelte Anführungszeichen gesetzt.
- Variablen vom Typ **String** können String-Objekte speichern.
- Die **String-Verknüpfung** verbindet zwei Strings zu einem String. Sie wird mit einem Pluszeichen (+) geschrieben.
- Eine **String-Verknüpfung** kann auch zwischen einem String mit einem Integerwert durchgeführt werden.
- Durch das **Parametrisieren** von Methoden (Hinzufügen von Parametern) werden Methoden flexibler und breiter anwendbar.

Vertiefende Aufgaben

In diesem Kapitel gibt es nicht die üblichen vertiefenden Aufgaben: Im letzten Abschnitt, *Spielzeit hinzufügen*, hatten wir bereits die Möglichkeit, unsere neuen Konstrukte und Techniken einzuüben.

Stattdessen kannst du ein paar Dinge tun, um deinem Szenario eine persönliche Note zu geben.

Übung 5.57 Experimentiere mit den Parametern diese Spiels, um es spielenswerter und interessanter zu machen. Du kannst die folgenden Punkte verstellen:

- die Geschwindigkeit, mit der sich die einzelnen Akteure bewegen
- die Anzahl der Punkte, die du gewinnen und verlieren kannst
- die zur Verfügung stehende Zeit
- die Soundeffekte
- die Frequenz, mit der neue Akteure erscheinen
- die Ausführungsgeschwindigkeit des Szenarios

und alles, was dir sonst noch einfällt.

Übung 5.58 Ändere die Bilder des Szenarios, um das Spiel in ein völlig neues Umfeld zu verlegen. Du könntest zum Beispiel ein Raumschiff durch das Weltall fliegen lassen, das Astronauten einsammelt und Asteroiden ausweicht. Oder ein Kaninchen, das über ein Feld läuft und dabei Möhren einsammelt und Hunden aus dem Weg geht.

Alles, was du willst. Denk dir etwas aus. Sei kreativ.

Wenn du mit deiner Idee und deinen Bildern zufrieden bist, solltest du auch die Klassennamen ändern.

Unterschiedliche Umgebungen könnten dich auf weitere Ideen für zusätzliche Funktionen bringen.