KAPITEL

6 Musizieren: Ein Bildschirm-Klavier

Lernziele

Themen: Sound

Konzepte: Abstraktion, Schleifen, Felder, objektorientierte Strukturen

In diesem Kapitel wollen wir ein neues Szenario beginnen – und zwar wollen wir ein Klavier implementieren, das über die Computertastatur gespielt werden kann. Abbildung 6.1 soll dir einen ersten Eindruck vermitteln, wie das fertige Klavier aussehen könnte.

Wie üblich starten wir, indem wir eines unserer Buchszenarien öffnen: *piano-1*. Diese Version unseres Projekts verfügt über alle Ressourcen, die wir benötigen (die Bilder und die Sounddateien), aber kaum mehr. Wir werden dieses Szenario als Ausgangsbasis verwenden und nach und nach den Code für unser Klavier ergänzen.

Abbildung 6.1
Das Ziel dieses Kapitels: ein Bildschirm-Klavier.

Übung 6.1 Öffne das Szenario *piano-1* und rufe den Quelltext für die beiden bestehenden Klassen **Piano** und **Key** auf. Informiere dich darüber, welcher Code bereits vorhanden ist und was er macht.

Übung 6.2 Erzeuge ein Objekt der Klasse **Key** und füge es in die Welt ein. Erzeuge mehrere Objekte und setze sie nebeneinander.

6.1 Die Tasten animieren

Wenn du den vorhandenen Code genau betrachtest, wirst du feststellen, dass dieser zurzeit noch ziemlich rudimentär ist. Die Klasse **Piano** gibt lediglich Größe und Auflösung der Welt vor und die Klasse **Key** enthält nur leere Methodengerüste für den Konstruktor und die Methode **act** (Listing 6.1).

Listing 6.1: Die Klasse **Key** im Anfangszustand.

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

public class Key extends Actor
{
    /**
     * Erstellt eine neue Taste.
     */
    public Key()
    {
    }

    /**
     * Die Aktion für diese Taste.
     */
    public void act()
    {
    }
}
```

Zum Herumexperimentieren wollen wir erst einmal ein Objekt der Klasse **Key** erzeugen und in der Welt platzieren. Wie du siehst, erhältst du damit lediglich eine einfache weiße Taste, die nichts macht, wenn du das Szenario ausführst.

Unsere erste Aufgabe wird also sein, die Klaviertaste zu animieren. Wenn wir eine Taste auf der Computertastatur drücken, soll die Klaviertaste in den Gedrückt-Zustand wechseln. Das Ausgangsszenario enthält hierfür bereits zwei Bilder (*white-key.png* und *white-key-down.png*), die wir für die beiden Zustände verwenden können. (Es gibt darüber hinaus zwei weitere Bilddateien namens *black-key.png* und *black-key-down.png*, die wir später für die schwarzen Tasten verwenden.) Wenn wir im Moment eine Taste erzeugen, erhalten wir das Bild *white-key.png*. Den Effekt einer gedrückten Taste können wir ganz leicht realisieren, indem wir beim Drücken einer bestimmten Taste auf der Computertastatur zu dem anderen Bild wechseln. Listing 6.2 enthält einen ersten Versuch.

```
/**
 * Die Aktion für diese Taste.
 */
public void act()
{
    if ( Greenfoot.isKeyDown("g") ) {
        setImage("white-key-down.png");
        isDown = true;
    }
    else {
        setImage("white-key.png");
    }
}
```

Listing 6.2
Erste Version der **act**-Methode: Bilder wechseln.

In diesem Code haben wir eine beliebige Taste auf der Computertastatur (die Taste „g") gewählt, auf die reagiert werden soll. Welche Taste wir wählen, spielt zu diesem Zeitpunkt keine Rolle – es geht uns vor allem darum zu sehen, wie wir die einzelnen Klaviertasten mit jeweils eigenen Computertasten verbinden können. Wenn die Taste auf der Computertastatur gedrückt wird, zeigen wir das Bild für die gedrückte Taste (*white-key-down.png*), und wenn nichts gedrückt wird, zeigen wir das normale Tasten-Bild (*white-key.png*).

Übung 6.3 Implementiere diese Version der **act**-Methode in deinem eigenen Szenario. Teste sie und stelle sicher, dass alles funktioniert.

Diese Version funktioniert zwar, ist aber mit einem Problem behaftet: Das Bild wird nicht nur einmal beim Wechseln gesetzt, sondern ständig. Das heißt, jedes Mal, wenn die **act**-Methode ausgeführt wird, wird das Bild auf eines der beiden Bilder gesetzt – also auch dann, wenn bereits das korrekte Bild angezeigt wird. Wird zum Beispiel die Taste „g" gerade nicht gedrückt, wird das Bild auf *white-key.png* gesetzt, unabhängig davon, ob dieses Bild bereits zuvor angezeigt wurde oder nicht.

Dieses Problem scheint auf den ersten Blick nicht allzu schwerwiegend zu sein. Schließlich ist das Setzen des Bildes zwar unnötig, aber kein wirklicher Fehler. Dennoch gibt es mehrere Gründe, warum wir diesen Makel beheben sollten. Ein Grund ist, dass es eine gute Angewohnheit ist, keine Prozessorressourcen durch unnötige Arbeit zu verschwenden. Ein weiterer Grund ist, dass wir bald die Taste mit einem Sound verbinden werden, und dann wird dieser Punkt wichtig, denn es ist ein großer Unterschied, ob der Sound einer Taste nach dem Drücken der Taste nur einmal zu hören ist oder ob er immer wieder abgespielt wird.

Aus diesem Grund wollen wir unseren Code verbessern und sicherstellen, dass das Bild nur gesetzt wird, wenn es sich auch wirklich geändert hat. Dazu fügen wir in unsere Klasse ein boolesches Zustandsfeld ein, in dem gespeichert ist, ob die Computertaste zurzeit gedrückt ist oder nicht. Wir nennen dieses Zustandsfeld **isDown** und seine Deklaration lautet wie folgt:

```
private boolean isDown;
```

Wir speichern in diesem Zustandsfeld den Wert **true**, wenn die Taste gedrückt ist, und **false**, wenn nicht.

Jetzt können wir prüfen, ob unsere Taste auf der Computertastatur gerade gedrückt worden ist: Wenn unser Zustandsfeld **isDown** den Wert **false** hat, aber die Taste „g" gedrückt wird, muss die Taste gerade gedrückt worden sein. Umgekehrt, wenn unser Zustandsfeld **isDown** den Wert **true** hat (wir gehen davon aus, dass die Taste gedrückt ist), aber die Taste „g" nicht gedrückt ist, muss sie gerade losgelassen worden sein. In diesen beiden Situationen können wir das Bild ändern. Listing 6.3 zeigt die vollständige Methode, die diese Idee implementiert.

Listing 6.3: Das Bild wird nur gesetzt, wenn es erforderlich ist.

```
public void act()
{
    if ( !isDown && Greenfoot.isKeyDown("g") ) {
        setImage("white-key-down.png");
        isDown = true;
    }
    if ( isDown && !Greenfoot.isKeyDown("g") ) {
        setImage("white-key.png");
        isDown = false;
    }
}
```

In beiden Fällen stellen wir sicher, dass das Zustandsfeld **isDown** auf den neuen Zustand gesetzt wird, wenn wir eine Änderung feststellen.

Konzept

Logische Operatoren wie **&&** (UND) und **!** (NICHT) lassen sich einsetzen, um mehrere boolesche Ausdrücke in einem booleschen Ausdruck zusammenzufassen.

In diesem Listing findest du zwei neue Symbole: das Ausrufezeichen (**!**) und das doppelte kaufmännische Und (**&&**).

Beides sind logische Operatoren. Das Ausrufezeichen bedeutet *NICHT*, während das doppelte kaufmännische Und für *UND* steht.

Mit diesem Hintergrundwissen lassen sich die folgenden Zeilen der **act**-Methode

```
if ( !isDown && Greenfoot.isKeyDown("g") ) {
    setImage ("white-key-down.png");
    isDown = true;
}
```

ein wenig formlos wie folgt lesen (Achtung! Kein Java-Code!):

```
if ( NICHT isDown) UND Greenfoot.isKeyDown("g") ) ...
```

Oder wenn man den Code noch ein wenig formloser formulieren möchte:

```
if (die-Klaviertaste-NICHT-gedrückt-ist UND
die-Computertaste-gedrückt-ist)
{
    ändere das Bild in das „down"-Bild;
    merke dir, dass die Klaviertaste jetzt gedrückt ist;
}
```

Studiere jetzt noch einmal den Code aus Listing 6.3 und stelle sicher, dass du ihn verstehst.

Eine vollständige Liste aller verfügbaren logischen Operatoren findest du in ***Anhang D***.

Übung 6.4 Implementiere die neue Version der **act**-Methode in deinem eigenen Szenario. Auf den ersten Blick scheint sich das Szenario nicht anders zu verhalten als vorher, aber diese Maßnahme war notwendig für unseren nächsten Schritt. Achte auch darauf, dass du nicht vergisst, das boolesche Zustandsfeld **isDown** am Anfang deiner Klasse zu definieren.

6.2 Den Sound erzeugen

Als Nächstes wollen wir dafür sorgen, dass beim Drücken der Taste ein Sound abgespielt wird. Dafür definieren wir in der Klasse **Key** eine neue Methode namens **play**. Wir können diese Methode im Editor unterhalb der **act**-Methode einfügen. Für den Anfang setzen wir den Kommentar, die Signatur und einen leeren Rumpf für die neue Methode auf:

```
/**
 * Spielt die Note dieser Taste.
 */
public void play()
{

}
```

Auch wenn dieser Code nichts macht (der Rumpf der Methode ist leer), sollte er sich dennoch problemlos übersetzen lassen.

Die weitere Implementierung dieser Methode ist recht einfach: Wir wollen lediglich eine Sounddatei abspielen. In dem Szenario *piano-1*, das diesem Projekt zugrunde liegt, stehen (im Unterordner *sounds*) eine Reihe von Sounddateien zur Verfügung, die jeweils nur einen Ton der Klaviertastatur beinhalten. Diese Sounddateien heißen *2a.wav*, *2b.wav*, *2c.wav*, *2c#.wav*, *2d.wav*, *2d#.wav*, *2d.wav* usw. Unter all diesen Dateien wollen wir einen mehr oder weniger zufälligen Ton – sagen wir, die Datei *3a.wav* – auswählen, der für unsere Testtaste abgespielt werden soll.

Damit der Ton auch wirklich gespielt wird, können wir erneut auf die Methode **playSound** der **Greenfoot**-Klasse zurückgreifen:

```
Greenfoot.playSound("3a.wav");
```

Das ist schon der gesamte Code, der in die Methode **play** eingefügt werden muss. Die vollständige Implementierung der Methode zeigt Listing 6.4.

```
/**
 * Spielt die Note dieser Taste.
 */
public void play()
{
    Greenfoot.playSound("3a.wav");
}
```

Listing 6.4
Den Ton für die Taste abspielen.

Übung 6.5 Implementiere die **play**-Methode in deiner Szenario-Version. Achte darauf, dass sich der Code übersetzen lässt.

Übung 6.6 Teste deine Methode. Dazu musst du ein Objekt der Klasse **Key** erzeugen, das Objekt mit der rechten Maustaste anklicken und aus dem Kontextmenü des Objekts die Methode **play** aufrufen.

Damit wären wir schon einen großen Schritt weiter. Wir können den Ton der Taste erzeugen, indem wir interaktiv die Methode **play** aufrufen, und wir können eine Taste auf der Computertastatur („g") drücken mit dem Effekt, dass die Klaviertaste gedrückt erscheint.

Jetzt müssen wir nur noch dafür Sorge tragen, dass der Ton gespielt wird, wenn die Taste auf der Computertastatur gedrückt wird.

Um den Ton vom Programm aus zu spielen, müssen wir einfach unsere eigene **play**-Methode aufrufen:

```
play();
```

Übung 6.7 Füge in deine Klasse **Key** Code ein, der den Ton der Taste spielt, wenn die damit verbundene Computertaste gedrückt wird. Dazu musst du dir überlegen, wo der Aufruf der Methode **play** eingefügt werden muss. Teste deinen Code.

Übung 6.8 Was passiert, wenn du zwei Tasten erzeugst, das Szenario ausführst und die Taste „g" drückst? Hast du eine Idee, was wir machen müssen, damit diese beiden Klaviertasten auf zwei verschiedene Computertasten reagieren?

All die bisherigen Änderungen findest du in den Buchszenarien unter *piano-2*. Wenn du auf Probleme gestoßen bist, die du nicht lösen konntest, oder wenn du nur deine Lösung mit unserer vergleichen möchtest, wirf ruhig einen Blick in den Code von *piano-2*.

6.3 Abstraktion: mehrere Tasten erzeugen

Wir sind nun so weit, dass wir eine Klaviertaste erzeugen können, die auf das Drücken einer Computertaste anspricht und einen Klavierton abspielt. Doch es gibt ein Problem: Wenn wir mehrere Klaviertasten erzeugen, reagieren alle auf dieselbe Computertaste und erzeugen den gleichen Ton. Das müssen wir ändern.

Die aktuelle Unzulänglichkeit beruht darauf, dass wir den Namen der Computertaste („g") und den der Sounddatei („3a.wav") in unserer Klasse hartcodiert haben. Das heißt, wir haben diese Namen direkt verwendet und somit keine Chance, sie zu ändern, es sei denn, wir ändern den Quelltext und übersetzen ihn neu.

Wenn du Computerprogramme schreibst, ist Code für sehr spezielle Aufgaben – wie die Quadratwurzel von 1,764 zu berechnen oder den Klavierton A zu spielen – nicht besonders nützlich. Grundsätzlich würden wir lieber Code schreiben, der eine ganze *Klasse* von Problemen löst (wie das Berechnen der Quadratwurzel einer beliebigen Zahl oder das Abspielen von einer Reihe von Klaviertönen). Dann wäre unser Programm wesentlich nützlicher. Wir sind bereits einem Beispiel hierfür im letzten Kapitel begegnet, als wir unsere **addScore**-Methode parametrisiert haben, sodass sie, anstatt jedes Mal 20 Punkte hinzuzuzählen, eine beliebige Anzahl von Punkten addieren konnte. Wir benötigen hier etwas Ähnliches.

Konzept

In der Programmierung gibt es viele verschiedene Formen der **Abstraktion**. Eine davon ist die Technik, Code zu schreiben, der eine ganze Klasse von Problemen und nicht nur ein einzelnes spezifisches Problem lösen kann.

Das Prinzip, das wir hier anwenden, heißt *Abstraktion*. In der Informatik findet man viele verschiedene Formen der Abstraktion in unterschiedlichen Kontexten – und diese ist eine davon.

Mithilfe der Abstraktion werden wir unsere Klasse **Key** jetzt verwandeln: von einer Klasse, die Objekte erzeugen kann, welche ein eingestrichenes A abspielen, wenn die Taste „g" auf der Computertastatur gedrückt wird, in eine Klasse, die Objekte erzeugen kann, welche ein ganzes Spektrum von Tönen abspielen können, je nachdem, welche Taste auf der Computertastatur angeschlagen wurde.

Um dies zu erreichen, richten wir eine Variable für den Namen der Computertaste ein, auf die wir reagieren wollen, und eine weitere für den Namen der abzuspielenden Sounddatei.

```
public class Key extends Actor
{
    private boolean isDown;
    private String key;
    private String sound;

    /**
     * Erzeugt eine neue Taste, verbunden mit einer Taste auf der
     * Computertastatur und einem vorgegebenen Sound.
     */
    public Key(String keyName, String soundFile)
    {
        key = keyName;
        sound = soundFile;
    }

    // Methoden ausgelassen **
}
```

Listing 6.5
Den Code für mehrere Tasten verallgemeinern: Variablen für die Taste und die Sounddatei einrichten.

In Listing 6.5 findest du einen Lösungsansatz. Wir verwenden hier zwei zusätzliche Zustandsfelder (**key** und **sound**), in denen wir den Namen der Taste und die zu verwendende Sounddatei speichern. Außerdem ergänzen wir den Konstruktor um zwei Parameter, über die diese Informationen hineingereicht werden können, wenn das Tasten-Objekt erstellt wird, und wir stellen sicher, dass wir diese Parameterwerte in den Zustandsfeldern im Konstruktorrumpf speichern.

Damit haben wir unsere Klasse **Key** auf eine höhere Abstraktionsebene gehoben. Wenn wir jetzt ein neues **Key**-Objekt erzeugen, können wir angeben, auf welche Computertaste das Objekt reagieren und welche Sounddatei es abspielen soll. Es fehlt allerdings noch der Code, der diese Variablen am Ende nutzt – diesen zu ergänzen ...

... überlassen wir dir zur Übung.

Übung 6.9 Implementiere die oben besprochenen Änderungen. Das heißt: füge Zustandsfelder für die Taste und die Sounddatei hinzu und einen Konstruktor mit zwei Parametern, der diese Zustandsfelder initialisiert.

Übung 6.10 Ändere deinen Code, sodass dein **Key**-Objekt auf die im Konstruktor vorgegebene Taste reagiert und die entsprechende Sounddatei abspielt. Teste deinen Code! (Erstelle verschiedene Tasten mit unterschiedlichen Sounds.)

Jetzt sind wir so weit, dass wir einen Satz Tasten zum Abspielen mehrerer verschiedener Töne erzeugen können. (Momentan besteht unsere Klaviatur nur aus weißen Tasten, doch lässt sich damit bereits ein halbes Klavier „nachbauen"). Diese Version des Projekts findest du in den Buchszenarien unter *piano-3*.

Die Erstellung der Tasten ist allerdings recht mühsam. Zurzeit müssen wir nämlich alle Klaviertasten einschließlich der jeweils dazugehörigen Parameter von Hand eingeben. Und noch schlimmer ist, dass wir jedes Mal, wenn wir eine Änderung am Quelltext vornehmen, wieder von vorn beginnen müssen. Deshalb wollen wir unseren Code so ändern, dass er die Tasten für uns erzeugt.

6.4 Das Klavier erstellen

Unser Ziel ist es, den Code in der Klasse **Piano** so umzuformulieren, dass er die Klaviertasten für uns erzeugt und anordnet. Das Hinzufügen einer (oder mehrerer) Taste(n) ist recht einfach: Durch Einfügen der folgenden Zeile in den Konstruktor von **Piano** wird jedes Mal, wenn wir das Szenario neu initialisieren, eine Taste erzeugt und in der Welt platziert:

```
addObject (new Key ("g", "3a.wav"), 300, 180);
```

Zur Erinnerung: Der Ausdruck

```
new Key ("g", "3a.wav")
```

erzeugt ein neues Tasten-Objekt (mit einer vorgegebenen Taste und Sounddatei), während die Anweisung

```
addObject( ein-Objekt, 300, 180);
```

das erzeugte Objekt an den angegebenen Koordinaten *x* und *y* in die Welt einfügt. Die genauen Koordinaten 300 und 180 wurden hier etwas willkürlich gewählt.

Übung 6.11 Erweitere deine Klasse **Piano** um Code, der automatisch eine Klaviertaste erzeugt und in der Welt platziert.

Übung 6.12 Ändere die *y*-Koordinate für die Taste, sodass die Taste genau oben am Rand des Klaviers beginnt (d.h., das obere Ende der Klaviertaste sollte mit dem oberen Rand des Klaviers abschließen). Hinweis: Das Bild der Taste ist 280 Pixel hoch und 63 Pixel breit.

Übung 6.13 Schreibe Code, der eine zweite Klaviertaste erzeugt, die beim Drücken der Computertaste „f" ein eingestrichenes G spielt (Sounddatei *3g.wav*). Platziere diese Taste genau links von der ersten Taste (ohne Lücke oder Überschneidung).

Weiter vorn in diesem Buch haben wir bereits besprochen, wie sinnvoll es ist, separate Aufgaben in separaten Methoden zu erledigen. Alle Tasten zu erzeugen, ist eine logisch eigenständige Aufgabe, sodass wir den Code dafür in eine eigene Methode auslagern wollen. An der Ausführung ändert sich dadurch nichts, aber der Code ist besser lesbar.

Übung 6.14 Lege in der Klasse **`Piano`** eine neue Methode namens **`makeKeys()`** an. Verschiebe deinen Code, der die Tasten erzeugt, in diese Methode. Rufe diese Methode aus dem **`Piano`**-Konstruktor heraus auf. Denke auch daran, einen Kommentar für deine neue Methode zu schreiben.

Wir könnten jetzt so weitermachen und eine ganze Reihe von **`addObject`**-Anweisungen einfügen, um alle Tasten für unsere Klaviatur zu erzeugen. Aber das ist nicht unbedingt der beste Weg, um zu unserem Ziel zu gelangen.

6.5 Schleifen: die while-Schleife

Programmiersprachen kennen ein spezielles Konstrukt, um ein und dieselbe Aufgabe mehrmals hintereinander auszuführen: die *Schleife*.

Konzept

Eine **Schleife** ist in der Programmierung eine Anweisung, die einen Codeabschnitt mehrere Male hintereinander ausführen kann.

Eine Schleife ist ein Programmiersprachenkonstrukt, das es uns ermöglicht, Befehle wie *Führe diese Anweisung 20-mal aus* oder *Rufe diese zwei Methoden drei Millionen Mal auf* einfach und prägnant auszudrücken (ohne drei Millionen Codezeilen zu schreiben). In Java gibt es mehrere Arten von Schleifen. Wir wollen uns an dieser Stelle mit der **`while`**-Schleife beschäftigen.

Eine **`while`**-Schleife hat die folgende Syntax:

```
while ( bedingung )
{
    anweisung;
    anweisung;
    ...
}
```

Auf das Java-Schlüsselwort **`while`** folgt eine Bedingung in Klammern und ein Block (in einem Paar geschweifter Klammern), in dem eine oder mehrere Anweisungen stehen. Diese Anweisungen werden so lange ausgeführt, wie die Bedingung erfüllt (**`true`**) ist.

Konzept

Eine **Schleifenvariable** ist eine lokale Variable, die für das Zählen der Schleifendurchläufe verwendet wird. Für eine `while`-Schleife sollte sie direkt vor der Schleife deklariert werden.

Weitverbreitet sind Schleifen, die einen Anweisungsblock so oft durchlaufen wie vorher festgelegt. Hierzu benötigen wir eine *Schleifenvariable* als Zähler. Üblicherweise werden Schleifenvariablen mit `i` bezeichnet, sodass wir dies hier auch so halten wollen. Das folgende Beispiel führt den Rumpf der `while`-Schleife 100-mal aus:

```
int i = 0;
while (i < 100)
{
    anweisung;
    anweisung;
    ...
    i = i + 1;
}
```

Dieser Code enthält einiges, auf das wir näher eingehen wollen. Zunächst einmal wird hier ein Konzept verwendet, das wir bereits kennen: eine lokale Variable. Unsere Variable heißt `i` und wir initialisieren sie mit 0. Dann führen wir mehrmals hintereinander die `while`-Schleife aus und zählen `i` bei jedem Durchlauf hoch. Das machen wir so lange, wie `i` kleiner ist als 100. Wenn wir 100 erreichen, beenden wir die Schleife. Die Ausführung wird dann mit dem Code fortfahren, der auf den Schleifenrumpf folgt.

Es gibt noch zwei Feinheiten, die erwähnt werden sollten:

- Wir verwenden die Anweisung `i = i + 1;` am Ende des Schleifenrumpfes, um unsere Schleifenvariable nach jedem Schleifendurchlauf um 1 zu erhöhen (inkrementieren). Dies ist wichtig. Häufig wird der Fehler gemacht, das Inkrementieren des Schleifenzählers zu vergessen. Die Folge wäre, dass sich die Variable nie ändert, die Bedingung also immer `true` bleibt und die Schleife immer und immer wieder durchlaufen wird. Dies wird auch als *Endlosschleife* bezeichnet und ist die Ursache für viele Fehler in Programmen.
- Unsere Bedingung besagt, dass wir die Schleife ausführen, solange `i` kleiner als (<) und nicht kleiner gleich (<=) 100 ist. Das heißt, die Schleife wird nicht ausgeführt, wenn `i` gleich 100 ist. Auf den ersten Blick könnte man denken, dass dann die Schleife nur 99-mal und nicht 100-mal ausgeführt wird. Dem ist aber nicht so. Da wir unsere Zählung mit 0 und nicht mit 1 beginnen, erhalten wir 100 Durchläufe (gezählt von 0 bis 99). In Computerprogrammen ist es allgemein üblich, die Zählung mit 0 zu beginnen – welche Vorteile das hat, werden wir bald sehen.

Nachdem wir jetzt wissen, wie `while`-Schleifen funktionieren, können wir dieses Konstrukt verwenden, um all unsere Klaviertasten zu erzeugen.

Unser Klavier soll zwölf weiße Tasten haben. Wir können diese zwölf Tasten erzeugen, indem wir unsere Anweisung zur Erzeugung einer Taste in den Rumpf einer Schleife packen, die wir zwölfmal ausführen:

```
int i = 0;
while (i < 12)
{
    addObject (new Key ("g", "3a.wav"), 300, 140);
    i = i + 1;
}
```

Übung 6.15 Ersetze den Code in deiner Methode **makeKeys** mit der hier vorgestellten Schleife. Teste deinen Code. Was kannst du beobachten?

Wenn du diesen Code ausführst, hat es zuerst den Anschein, als wenn nur eine Taste erzeugt worden wäre. Dies ist jedoch eine Täuschung. Es sind tatsächlich zwölf Tasten vorhanden; da sie sich jedoch alle an den gleichen Koordinaten befinden, liegen sie genau übereinander, sodass wir sie nicht unterscheiden können. Versuche einmal, die Tasten in der Piano-Welt mit deinem Mauszeiger zu verschieben und du wirst feststellen, dass sie alle da sind.

Übung 6.16 Welche Änderungen musst du an deinem Code vornehmen, damit die Tasten nicht alle an derselben Position erscheinen? Kannst du deinen Code so ändern, dass sie genau nebeneinander platziert werden?

Der Grund, warum die Tasten alle übereinanderliegen, ist, dass wir sie alle an einer festen Position (300, 140) in die Welt eingefügt haben. Jetzt wollen wir jede Taste an einer anderen Position einfügen. Das ist mit dem Wissen von oben nicht mehr schwer: Wir müssen uns nur der Schleifenvariablen **i** bedienen.

Übung 6.17 Wie oft wird unser Schleifenrumpf durchlaufen? Welche Werte nimmt **i** während der einzelnen Durchläufe an?

Wir können jetzt die feste *x*-Koordinate durch einen Ausdruck ersetzen, der die Variable **i** enthält:

```
addObject (new Key ("g", "3a.wav"), i*63, 140);
```

(Der Stern „*" ist der Multiplikationsoperator. In ***Anhang D*** findest du eine Liste weiterer Operatoren, die du mit ganzen Zahlen verwenden kannst.)

Wir haben **i*63** gewählt, weil wir wissen, dass jede Taste 63 Pixel breit ist. Die Werte für **i** bei der Ausführung der Schleife lauten 0, 1, 2, 3 usw. Die Tasten werden also an den *x*-Koordinaten 0, 63, 126, 189 usw. platziert.

Wenn wir dies ausprobieren, stellen wir fest, dass die linke Taste nicht optimal platziert wurde. Da die Platzierung von Objekten in Greenfoot sich auf den Mittelpunkt des Objekts bezieht, befindet sich die Mitte der ersten Taste an der *x*-Koordinate 0, wodurch die eine Hälfte der Taste aus dem Bildschirm rutscht. Um dieses Manko zu beheben, addieren wir einfach einen festen Offset zu jeder Tasten-Koordinate. Der Offset wurde so gewählt, dass die Klaviatur als Ganzes genau in der Mitte unseres Klaviers liegt:

```
addObject (new Key ("g", "3a.wav"), i*63 + 54, 140);
```

Die *y*-Koordinate kann unverändert bleiben, da alle Tasten auf der gleichen Höhe liegen sollen.

Übung 6.18 Übung für Fortgeschrittene. (Mache diese Übung nur, wenn du dich in der Programmierung schon etwas sicherer fühlst. Anfänger können diese Übung getrost erst einmal überspringen.)

Die Verwendung von festen Zahlen in deinem Code, wie 140 oder 63 in der Anweisung oben, ist normalerweise nicht die beste Lösung, da dein Code dadurch fehleranfälliger wird, wenn Änderungen vorgenommen werden müssen. Wenn wir beispielsweise das Bild unserer Taste durch ein netteres Bild ersetzen, das andere Abmaße aufweist, könnte unser Code die Tasten nicht korrekt platzieren.

Wir können die direkte Verwendung dieser Zahlen vermeiden, indem wir die Methoden **`getWidth()`** und **`getHeight()`** des Tasten-Bildes aufrufen. Dazu weisen wir zuerst das Tasten-Objekt bei seiner Erzeugung einer lokalen Variablen vom Typ **`Key`** zu und verwenden dann **`key.getImage().getWidth()`** anstelle von **`63`**. Mache das gleiche mit der Höhe.

Um die **`54`** ersetzen zu können, brauchst du die **`getWidth()`**-Methode des Klavier-Bildes.

Anschließend wird der Code die Tasten immer hübsch nebeneinander platzieren, egal wie groß die Tasten sind.

Jetzt platziert unser Code die weißen Tasten korrekt – das ist bereits ein großer Schritt vorwärts. Das wichtigste Problem, das sich uns noch stellt, ist, dass alle Klaviertasten auf die gleiche Computertaste reagieren und den gleichen Ton abspielen. Um dies zu beheben, benötigen wir ein neues Programmierkonstrukt: das *Feld*.

6.6 Felder

Zurzeit werden all unsere zwölf Klaviertasten korrekt erzeugt und an den gewünschten Positionen auf dem Bildschirm angezeigt. Doch leider reagieren sie nur auf die Taste „g" und spielen alle den gleichen Ton ab. Und das, obwohl wir unsere Tasten so konzipiert haben, dass sie über den Konstruktor verschiedene Computertasten und Sounddateien akzeptieren können. Da jedoch all unsere Tasten von derselben Quelltextzeile erzeugt werden (ausgeführt in einer Schleife), werden sie alle mit den Parametern „g" und „3a.wav" erzeugt.

Die Lösung gleicht den Änderungen, die wir für die *x*-Koordinate vorgenommen haben: Wir richten Variablen für die Computertaste und die Sounddatei ein und weisen diesen bei jedem Schleifendurchlauf andere Werte zu.

Dennoch ist dieses Problem schwieriger zu lösen als das der *x*-Koordinate. Dies liegt daran, dass die korrekten Tasten- und Sounddateinamen sich nicht so einfach berechnen lassen. Woher erhalten wir also unsere Werte?

Die Antwort lautet: Wir speichern sie in einem Feld (*array*).

Ein Feld ist ein Objekt, das viele Variablen enthalten und deshalb viele Werte speichern kann. Dies können wir anhand eines Diagramms veranschaulichen. Angenommen, wir haben eine Variable **name** vom Typ **String**. Dieser Variablen weisen wir den **String** „Fred" zu:

Konzept

Ein **Feld** ist ein Objekt, das mehrere Variablen hält. Der Zugriff darauf erfolgt über einen **Index**.

```
String name;
name = "Fred";
```

Abbildung 6.2 veranschaulicht dieses Beispiel (*array*).

Abbildung 6.2
Eine einfache **String**-Variable.

Dieser Fall ist sehr einfach. Die Variable ist ein Container, der einen Wert aufnehmen kann. Der Wert wird in der Variablen gespeichert.

Im Falle eines Feldes haben wir es mit einem separaten Objekt – dem Feld-Objekt – zu tun, das viele Variablen enthält. In unserer eigenen Variable können wir dann eine Referenz auf das Feld-Objekt speichern (Abbildung 6.3).

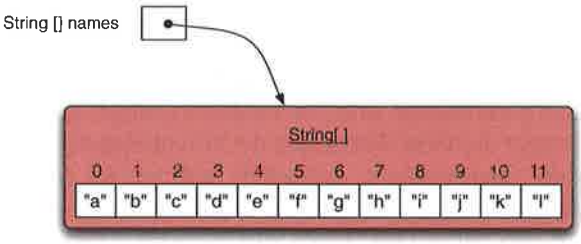

Abbildung 6.3
Ein Feld von Strings.

Der Java-Code hierfür sieht folgendermaßen aus:

```
String[] names;
names = { "a","b","c","d","e","f","g","h","i","j","k","l" };
```

In der Variablendeklaration wird durch das Paar eckiger Klammern (**[]**) angezeigt, dass es sich bei diesem Variablentyp um ein Feld handelt. Das Wort vor den eckigen Klammern gibt den *Elementtyp* des Feldes an, d.h. den Typ, den jeder Eintrag im Feld haben sollte. So wird mit **String[]** ein Feld von Strings bezeichnet, während **int[]** ein Feld von Integer beschreibt.

Der Ausdruck

```
{ "a","b","c","d","e","f","g","h","i","j","k"l,"l" };
```

erzeugt das Feld-Objekt und füllt es mit den Strings „a" bis „l". Dieses Feld-Objekt wird dann unserer Variablen **names** zugewiesen. Wie wir an dem Diagramm erkennen, enthält eine Variable, der ein Feld-Objekt zugewiesen wurde, anschließend einen Zeiger auf dieses Objekt.

Sobald wir unsere Feldvariable definiert haben, können wir über einen *Index*, d.h. die Angabe einer Position im Feld-Objekt, auf die einzelnen Elemente in dem Feld zugreifen. In Abbildung 6.3 wird der Index der einzelnen Strings über jedem Feldelement angezeigt. Beachte, dass auch hier die Zählung wieder bei 0 beginnt, sodass der String „a" sich an der Position 0 befindet, „b" an der Position 1 usw.

Konzept

Der Zugriff auf einzelne **Elemente** in einem Feld erfolgt über eckige Klammern ([]) und der Angabe eines Indexes für das gewünschte Feldelement.

In Java greifen wir auf die Feldelemente zu, indem wir den Index in eckigen Klammern an den Feldnamen anhängen. So greifst du mit dem Code

```
names[3]
```

im Feld **names** auf das Element an der Indexposition 3 (dem String „d") zu.

Für unser Klavier-Projekt können wir jetzt zwei Felder einrichten: eines mit den Namen der Computertasten (in der gewünschten Reihenfolge) für unsere Klaviertasten und eines mit den Namen der Sounddateien für diese Klaviertasten. Wir können für diese Felder Zustandsfelder in der Klasse **Piano** deklarieren und in diesen die gefüllten Felder speichern. Wie dies geht, zeigt Listing 6.6.

Listing 6.6: Felder für Tasten und Töne erzeugen.

```
public class Piano extends World
{
    private String[] whiteKeys =
        { "a", "s", "d", "f", "g", "h", "j", "k", "l", ";", "'", "\\" };
    private String[] whiteNotes =
        { "3c", "3d", "3e", "3f", "3g", "3a", "3b", "4c", "4d", "4e", "4f", "4g" };

    // Konstruktor und Methoden ausgelassen
}
```

Beachte, dass die Werte in dem Feld **whiteKeys** die Tasten auf der mittleren Reihe meiner Computertastatur sind. Da Tastaturen jedoch system- und landesspezifisch sind, musst du unter Umständen die Tastenbelegung ändern, damit sie mit deiner Tastatur übereinstimmt. Außerdem möchten wir dich auf den etwas seltsamen String „\\" hinweisen. Das Backslash-Zeichen (\) wird auch *Escape-Zeichen* genannt und hat in Java-Strings eine besondere Bedeutung. Um einen String zu erzeugen, der den Backslash als normales Zeichen enthält, musst du den Backslash zweimal eingeben. Das heißt, durch das Eintippen des Strings „\\" in deinen Java-Quelltext erzeugst du eigentlich den String „\".

Damit verfügen wir über Felder mit Auflistungen der Computertasten und Sounddateien, die wir für unsere Klaviertasten verwenden wollen. Jetzt können wir in der **makeKeys**-Methode die notwendigen Änderungen an unserer Schleife vornehmen, um so mithilfe der Feldelemente die gewünschten Tasten zu erzeugen. Listing 6.7 zeigt den zugehörigen Quelltext.

Listing 6.7: Klaviertasten mithilfe von Computertasten und Tönen aus Feldern erzeugen.

```
/**
 * Erzeugt die Klaviertasten und platziert diese in der Welt.
 */
private void makeKeys()
{
    int i = 0;
    while (i < whiteKeys.length)
    {
        Key key = new Key(whiteKeys[i], whiteNotes[i] + ".wav");
        addObject(key, i*63 + 54, 140);
        i = i + 1;
    }
}
```

Es gibt eine Reihe von Dingen, die hier erwähnenswert sind:

- Wir haben die Erzeugung der neuen Taste aus dem **addObject**-Methodenaufruf herausgenommen und in eine separate Zeile verschoben, wo wir das Tasten-Objekt zunächst einer lokalen Variablen namens **key** zuweisen. Dies geschah allein wegen der Übersichtlichkeit: Die Zeile wurde sehr lang und unübersichtlich und war daher nur schwer zu lesen und zu verstehen. Durch die Zerlegung in zwei Schritte ist der Code leichter zu lesen.
- Die Parameter für den **Key**-Konstruktor greifen auf **whiteKeys[i]** und **whiteNotes[i]** zu. Das heißt, wir verwenden unsere Schleifenvariable **i** als Feldindex, um nacheinander auf die verschiedenen Tasten-Strings und Sounddateinamen zuzugreifen.
- Wir verwenden **whiteNotes[i]** zusammen mit einem Pluszeichen (+) und einem String („.wav"). Die Variable **whiteNotes[i]** ist ebenfalls ein String, wir haben es hier also wieder mit einer String-Verknüpfung zu tun, die wir schon im vorigen Kapitel kennengelernt haben. Wir hängen hier den String „.wav" an den Wert von **whiteNotes[i]** an. Der Grund ist, dass der im Feld gespeicherte Name die Form „3c" hat, während der Dateiname auf der Festplatte „3c.wav" lautet. Wir hätten auch ohne Weiteres den vollständigen Namen im Feld speichern können, aber da das Suffix für alle Sounddateien gleich lautete, schien dies überflüssig. Mit diesem Trick ersparen wir uns etwas Tipparbeit.
- Außerdem haben wir die zwölf in der Bedingung der **while**-Schleife durch **whiteKeys.length** ersetzt. Das Attribut **.length** eines Feldes liefert die Anzahl der Elemente in diesem Feld zurück. In unserem Fall haben wir zwölf Elemente, sodass unser Code auch funktioniert hätte, wenn wir in der Bedingung **12** geschrieben hätten. Mit der Verwendung des Attributs **length** gehst du jedoch auf Nummer sicher. Solltest du dich später entscheiden, mehr oder weniger Tasten zu verwenden, wird unsere Schleife immer noch funktionieren, ohne dass du Änderungen an der Bedingung vornehmen musst.

Nach all diesen Änderungen solltest du in der Lage sein, unser Piano mit der mittleren Tastenreihe auf der Tastatur zu spielen, und es sollte bei jeder Taste ein anderer Ton erklingen.

Übung 6.19 Ändere dein Szenario wie oben besprochen. Achte darauf, dass alle Tasten funktionieren. Wenn dein Tastaturlayout anders aussieht, passe das Feld **whiteKeys** an deine Tastatur an.

Übung 6.20 Der Ordner *sounds* des Klavier-Szenarios enthält mehr Sounddateien, als wir hier verwenden. Ändere das Klavier so, dass die Tasten eine Oktave tiefer sind als jetzt. Das bedeutet, verwende „2c" anstelle von „3c" als erste Taste und zähle von dort hoch.

Übung 6.21 Wenn du möchtest, kannst du den Code so ändern, dass völlig andere Sounds abgespielt werden. Nimm dazu eigene Sounds mit dem Klang-Rekorder von Greenfoot oder einer speziellen Aufnahmesoftware auf oder suche nach Sounddateien im Internet. Verschiebe die Sounddateien in den Ordner *sounds* und verbinde sie mit den Tasten.

Die Version, die wir bis hierher erarbeitet haben, findest du in den Buchszenarien unter *piano-4*.

Was jetzt noch fehlt, ist ziemlich offensichtlich: Wir müssen noch die schwarzen Tasten hinzufügen.

Die dafür nötigen Techniken sind uns bekannt und die Vorgehensweise ist derjenigen zum Hinzufügen der weißen Tasten sehr ähnlich. Wir überlassen dir daher das Hinzufügen der schwarzen Tasten als Übung. Alles auf einmal zu machen, ist jedoch ziemlich arbeitsaufwendig. Allgemein empfiehlt es sich, größere Aufgaben in mehrere kleinere zu zerlegen. Deshalb wollen wir diese Aufgabe in mehrere Übungen aufteilen, die sich der Lösung schrittweise nähern.

Übung 6.22 Im Moment kann unsere Klasse **Key** nur weiße Tasten erzeugen. Das liegt daran, dass wir die Dateinamen der Tasten-Bilder (*white-key.png* und *white-key-down.png*) hartcodiert haben. Verändere die Klasse **Key** mithilfe der Abstraktionstechnik so, dass sie entweder weiße oder schwarze Tasten anzeigen kann. Dies entspricht der Vorgehensweise für die Tastennamen und Sounddateinamen: Führe zwei Zustandsfelder ein und zwei Parameter für die Namen der beiden Bilddateien. Anschließend verwende die Variablen anstelle der hartcodierten Dateinamen. Teste deinen Code, indem du einige schwarze und weiße Tasten erzeugst.

Übung 6.23 Ändere deine Klasse **`Piano`**, sodass sie zwei schwarze Tasten an einer beliebigen Position einfügt.

Übung 6.24 Füge der Klasse **`Piano`** zwei weitere Felder für die Computertasten und die Töne der schwarzen Tasten hinzu.

Übung 6.25 Füge eine weitere Schleife in die Methode **`makeKeys`** der Klasse **`Piano`** ein, die die schwarzen Tasten erzeugt und platziert. Dies wird dadurch etwas erschwert, dass schwarze Tasten nicht so gleichmäßig verteilt sind wie die weißen, sondern Lücken aufweisen (siehe Abbildung 6.1). Fällt dir dazu eine Lösung ein? Tipp: Erzeuge einen speziellen Eintrag in deinem Feld, wo sich die Lücken befinden, und nutze diese Informationen, um die Lücken zu erkennen. (Lies zuerst den nachfolgenden Hinweis, bevor du loslegst. Diese Aufgabe ist ziemlich schwierig! Falls du keine Idee hast oder stecken bleibst, kannst du die Lösung in *piano-complete* nachschlagen.)

Hinweis: Die Klasse String

Wir haben die **`String`**-Klasse nun bereits mehrere Male benutzt und einige ihrer Eigenschaften im vorigen Kapitel besprochen. Nun wollen wir noch ein bisschen mehr darüber erfahren. Suche diese Klasse in den **Java-Klassenbibliotheken** und wirf einen Blick auf ihre Methoden. Es gibt davon eine ganze Menge und einige davon sind oft sehr nützlich.

Du findest dort Methoden zur Erzeugung von Teilstrings, zur Ermittlung der Stringlänge, zur Umwandlung der Groß- bzw. Kleinschreibung und vieles mehr.

Für Übung 6.25 dürfte vor allem die Methode **`equals`** interessant sein, die es dir erlaubt, den String mit einem anderen zu vergleichen. Sie liefert **`true`** zurück, wenn die beiden Strings gleich sind.

Weiter wollen wir dieses Projekt nicht verfolgen. Das Klavier ist mehr oder weniger fertig. Wir können einfache Melodien, ja sogar Akkorde spielen (indem wir mehrere Tasten gleichzeitig drücken).

Du kannst dieses Projekt jedoch ruhig erweitern. Wie wär's, wenn du einen zweiten Satz an Sounds hinzufügst und dann einen Schalter auf dem Bildschirm einblenden lässt, der es dir ermöglicht, zwischen den Klavier-Sounds und deinen alternativen Sounds zu wechseln?

Zusammenfassung der Programmiertechniken

In diesem Kapitel haben wir die Bekanntschaft mit zwei grundlegenden, sehr wichtigen Konzepten für die fortgeschrittene Programmierung gemacht: Schleifen und Felder. Schleifen erlauben es uns, Code zu schreiben, der eine Folge von Anweisungen mehrmals hintereinander ausführt. Das Schleifenkonstrukt, das wir in diesem Zusammenhang besprochen haben, heißt *while-Schleife*. Java kennt daneben weitere Schleifen, die wir im Verlauf dieses Buches noch kennenlernen werden. Wir werden Schleifen in vielen unserer Programme verwenden, sodass es wichtig ist, sie vom Prinzip her zu verstehen.

Innerhalb der Schleife verwenden wir oft einen Schleifenzähler, um Berechnungen durchzuführen oder um in jedem Schleifendurchlauf verschiedene Werte zu generieren.

Das zweite grundlegende Konzept, das wir verwendet haben, war das Feld. Ein Feld kann viele Variablen (alle vom gleichen Typ) in nur einem Objekt verwahren. In Situationen, wo wir mit jedem Element des Feldes etwas machen müssen, werden oft Schleifen dazu genutzt, um das Feld zu durchlaufen. Der Zugriff auf die einzelnen Elemente eines Feldes erfolgt mithilfe der eckigen Klammern.

Eine weitere grundlegende Technik, die uns begegnet ist, war die Abstraktion. In unserem Fall basierte die Abstraktion auf der Verwendung von Konstruktorparametern, welche es uns erlaubten, Code zu erzeugen, der eine ganze Klasse von Problemen behandelt und nicht nur ein spezifisches Problem.

Schließlich haben wir noch einige neue Operatoren kennengelernt: Wir haben die Operatoren UND und NICHT (**`&&`** und **`!`**) für boolesche Ausdrücke verwendet und wir sind noch einmal der String-Verknüpfung begegnet, die den Plus-Operator (**`+`**) auf String-Operanden einsetzt. Die Klasse **`String`** ist in den Java-Klassenbibliotheken dokumentiert und verfügt über viele nützliche Methoden.

Zusammenfassung der Konzepte

- **Logische Operatoren** wie **&&** (UND) und **!** (NICHT) lassen sich einsetzen, um mehrere boolesche Ausdrücke in einem booleschen Ausdruck zusammenzufassen.
- In der Programmierung gibt es viele verschiedene Formen der **Abstraktion**. Eine davon ist die Technik, Code zu schreiben, der eine ganze Klasse von Problemen und nicht nur ein einzelnes spezifisches Problem lösen kann.
- Eine **Schleife** ist in der Programmierung eine Anweisung, die einen Codeabschnitt mehrere Male hintereinander ausführen kann.
- Eine **Schleifenvariable** ist eine lokale Variable, die eingesetzt wird, um die Schleifendurchläufe in einer Schleife zu zählen. Für eine **`while`**-Schleife sollte sie direkt vor der Schleife deklariert werden.
- Ein **Feld** ist ein Objekt, das mehrere Variablen hält. Der Zugriff darauf erfolgt über einen **Index**.
- Der Zugriff auf einzelne **Elemente** in einem Feld erfolgt über eckige Klammern (**[]**) und der Angabe eines Indexes für das gewünschte Feldelement.
- Der Typ **String** wird durch eine normale Klasse definiert. Sie verfügt über viele nützliche Methoden, die wir in den Java-Klassenbibliotheken nachschlagen können.

Vertiefende Aufgaben

Dieses Mal wollen wir das wichtigste Konstrukt einüben, das wir gerade gelernt haben: die Verwendung der **`while`**-Schleife. Wir werden die Übungen mit einem anderen Szenario durchführen, das *bubbles* (Blasen) heißt (Abbildung 6.4). Öffne das Szenario, um damit die folgenden Übungen zu bearbeiten.

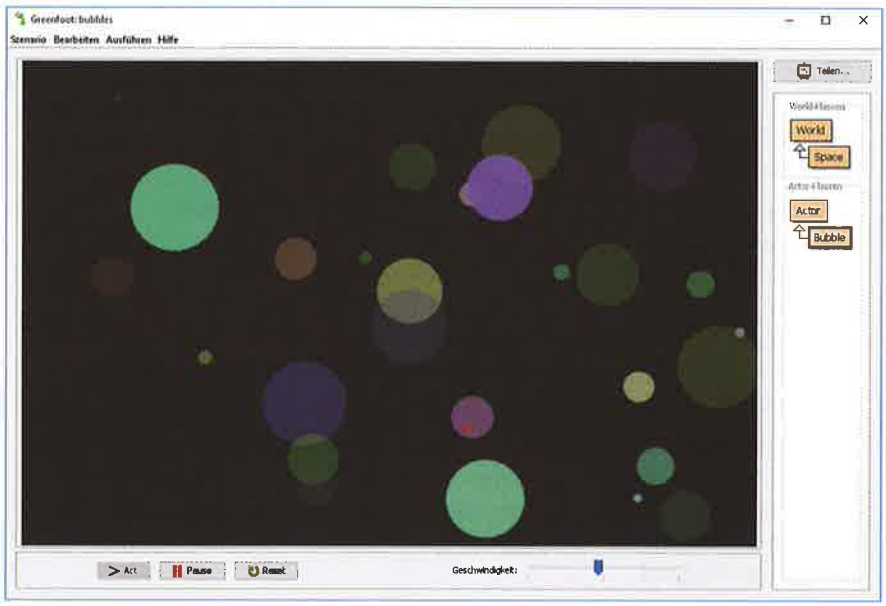

Abbildung 6.4
Schwebende Blasen.

Übung 6.26 Öffne das Szenario *bubbles*. Du wirst feststellen, dass die Welt leer ist. Platziere einige **`Bubble`**-Objekte in der Welt, benutze dazu den *Standardkonstruktor* (das ist derjenige ohne Parameter). Zur Erinnerung: du kannst dies auch erreichen, indem du bei gedrückter ⇧-Taste in die Welt klickst. Was beobachtest du?

Übung 6.27 Welches ist die Anfangsgröße einer neuen Blase? Welche Farbe hat sie? Wie ist die anfängliche Bewegungsrichtung?

Übung 6.28 Erzeuge in der Welt-Unterklasse **`Space`** eine neue private Methode mit Namen **`setup()`**. Rufe diese Methode vom Konstruktor in dieser Methode auf, erzeuge eine neue Blase mithilfe des Standardkonstruktors und platziere diese im Mittelpunkt der Welt. Kompiliere erst, dann klicke zum Testen ein paar Mal auf RESET.

Übung 6.29 Verändere deine **`setup()`**-Methode, sodass sie nun 21 Blasen erzeugt. Verwende eine **`while`**-Schleife. Alle Blasen werden in der Mitte der Welt platziert. Führe dein Szenario aus, um es zu testen.

Übung 6.30 Platziere die 21 Blasen an zufälligen Positionen in der Welt.

Übung 6.31 Platziere die Blasen auf einer Diagonalen mit *x*- und *y*-Abständen von 30. Die erste Blase kommt dann an Position (0,0), die nächste an (30,30) die nächste an (60,60) und so weiter. Die letzte Blase wird dann an Position (600,600) sein (bei 21 Blasen). Verwende dazu deine Schleifenzählervariable.

Übung 6.32 Platziere die Blasen auf einer etwas anderen diagonalen Linie, sodass sie von der oberen linken Ecke der Welt zur unteren rechten Ecke gehen. Die letzte Blase sollte am Punkt (900,600) sein.

Übung 6.33 Füge eine zweite **`while`**-Schleife zu deiner **`setup()`**-Methode hinzu, die einige weitere Blasen erzeugt. Diese Schleife soll 10 Blasen in einer horizontalen Linie platzieren, angefangen bei $x = 300$ und $y = 100$, wobei x jedes Mal um 40 erhöht wird und y konstant bleibt (das heißt, die Blasen sind an den Positionen (300,100), dann (340,100), (380,100) und so weiter). Die Größe der Blasen sollte ebenfalls anwachsen: die erste Blase hat die Größe 10, die zweite 20, die nächste 30 usw. Verwende den zweiten **`Bubble`**-Konstruktor dafür (derjenige mit einem Parameter).

Übung 6.34 Entferne die vorhandene Schleife aus deiner **`setup()`**-Methode. Schreibe eine neue **`while`**-Schleife, die Folgendes macht: sie erzeugt zunächst 18 Blasen; die Blasen werden alle im Zentrum der Welt platziert und ihre Größen beginnen bei 190 und werden jeweils um 10 kleiner. Die letzte Blase hat die Größe 10. Stelle sicher, dass die größte Blase zuerst erzeugt wird und die kleinste am Schluss, sodass du Blasen der Größen 190, 180, 170 usw. bekommst, die alle übereinander liegen.

Verwende den dritten Bubble-Konstruktor – den mit den zwei Parametern. Damit kannst du auch die Anfangsrichtung angeben. Gib die Richtung folgendermaßen vor: die erste Blase hat Richtung 0, die nächste 20, die nächste 40 und so weiter. Das heißt, die Richtung zwischen zwei Blasen erhöht sich jeweils um 20 Grad.

Teste deinen Code.