KAPITEL

7 Objektinteraktion: eine Einführung

Lernziele

Themen: Kommunikation mit anderen Objekten, Klassen aus der Java-Bibliothek benutzen, Listen von Objekten verwenden

Konzepte: **null**, Java-Klassenbibliothek, Sammlung (Collection), Liste, **for-each**-Schleife

In diesem Kapitel führen wir zwei sehr wichtige Konzepte in Vorbereitung auf das folgende Kapitel ein: mit anderen Objekten kommunizieren und mit Listen von Objekten umgehen. In ***Kapitel 8*** werden wir ein deutlich komplexeres Beispiel entwickeln – dabei geht es um Planeten im Weltraum –, das viele neue Konstrukte verwendet, wovon einige ein wenig Erklärung und Praxis bedürfen. Um die Dinge Schritt für Schritt anzugehen und nicht zu viel auf einmal zu machen, wollen wir hier zuerst diese zwei Konzepte mit einem kleineren Beispiel einführen.

Das Szenario, das wir in diesem Kapitel besprechen, ist eine Art Übungsfeld für diese Konstrukte: *autumn* (Abbildung 7.1).

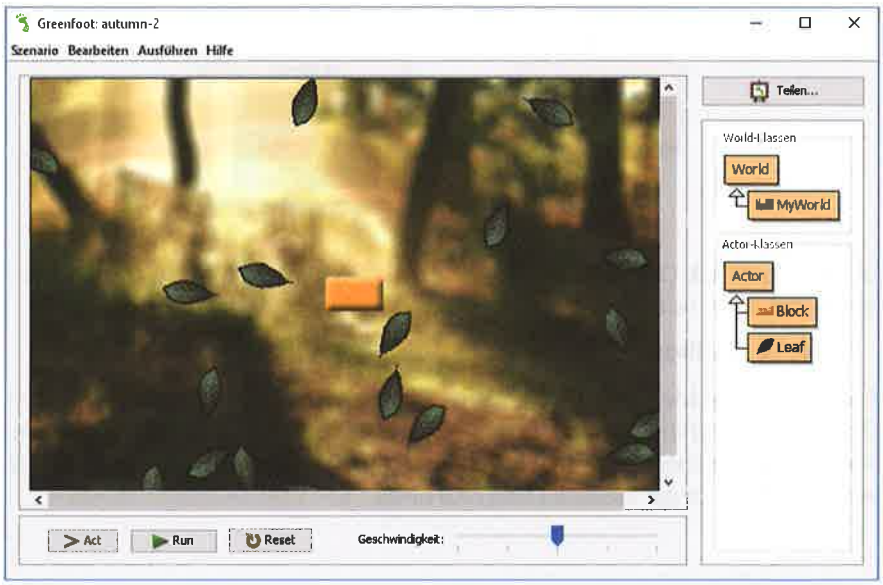

Abbildung 7.1
Das Szenario *autumn*.

Übung 7.1 Öffne das Szenario *autumn-1* und untersuche es. Führe es aus und beobachte, was es macht. Lies den Quelltext der drei Szenario-Klassen und notiere dir jedes Codesegment, dass du nicht vollständig verstehst.

7.1 Interagierende Objekte

Objekte können mit anderen Objekten interagieren („sprechen"), indem sie Methoden dieser anderen Objekte aufrufen.

Wir sind in den vorherigen Kapiteln mehreren Beispielen dafür begegnet, doch nun ist es an der Zeit, alles, was wir bisher gesehen und besprochen haben, systematisch zusammenzutragen. Damit wollen wir sicherstellen, dass wir nichts Wichtiges ausgelassen haben. Objektinteraktion ist so zentral für die objektorientierte Programmierung, dass wir es uns nicht leisten können, hier irgendetwas hier zu versäumen.

7.2 Objektreferenzen

Wenn ein Objekt mit einem anderen sprechen möchte, muss es stets eine Referenz auf dieses andere Objekt besitzen. Diese Referenz wird in der Regel in einer Variablen gespeichert. Nehmen wir an, unser **Block**-Objekt aus dem *autumn*-Szenario möchte mit dem **MyWorld**-Objekt sprechen. Dann muss es zuerst eine Referenz auf das **MyWorld**-Objekt haben. Abbildung 7.2 zeigt diese Situation: Das Objekt vom Typ **Block** hält eine Referenz auf ein **MyWorld**-Objekt in einer Variablen namens **world**.

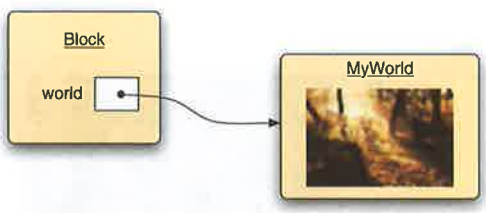

Abbildung 7.2
Ein **Block**-Objekt hält eine Referenz auf ein **MyWorld**-Objekt.

Sobald das **Block**-Objekt eine Referenz auf das **MyWorld**-Objekt in einer Variablen besitzt, kann es dessen Methoden aufrufen. Zum Beispiel:

```
world.addObject(new Leaf(), 100, 100);
```

Der Aufruf einer Methode eines anderen Objekts erfüllt die Funktion, mit diesem Objekt zu „sprechen". Häufig wird dies gemacht, um dem anderen Objekt eine Anweisung zu geben (in diesem Fall, um dem Welt-Objekt zu sagen: „Bitte füge ein neues Blatt an der Position (100,100) hinzu.").

7.3 Interaktion mit der Welt

Da wir immer eine Objektreferenz benötigen, um mit einem Objekt zu sprechen, ist eine offensichtliche Frage: Woher bekommen wir diese Objektreferenz? Nun, hier gibt es mehrere Möglichkeiten.

In obigem Beispiel spricht unser **Block**-Objekt mit dem Welt-Objekt. Wir können eine Referenz auf das Welt-Objekt erhalten, indem wir die **getWorld()**-Methode verwenden, die jeder Akteur besitzt.

Die **getWorld()**-Methode liefert uns eine Referenz auf die aktuell instanziierte Welt, die immer eine Unterklasse der Klasse **World** ist. Wir können eine Variable vom Typ **World** deklarieren und unser Welt-Objekt darin abspeichern. Wenn wir dies tun, bevor wir unsere Methode aufrufen, erhalten wir:

```
World world = getWorld();
world.addObject(new Leaf(), 100, 100);
```

Übung 7.2 Füge Code in deiner **Block**-Klasse hinzu, der bewirkt, dass jedes Mal, wenn der Block auf die Ränder der Welt trifft, neue Blätter in die Welt kommen. Füge zunächst die Blätter an der Position (100,100) hinzu, wie in obigem Beispiel.

Übung 7.3 Verändere deinen Code dahingehend, dass das neue Blatt an der aktuellen Position des Blocks erscheint. Du kannst dazu die **getX()**- und **getY()**-Methoden des Blocks verwenden.

7.4 Mit Akteuren interagieren

Oben haben wir gesehen, wie eine Methode eines Welt-Objekts aufgerufen wird. (Wir waren diesem Konzept sogar in einem früheren Kapitel schon einmal begegnet.) Als Nächstes untersuchen wir, wie eine Methode eines anderen Akteur-Objekts aufgerufen wird.

Dazu verwenden wir folgendes Beispiel: Wir programmieren unseren Block so, dass sich jedes Mal, wenn er ein Blatt berührt, dieses Blatt dreht.

Da **Leaf** eine Unterklasse von **Actor** ist und **Actor** eine **turn**-Methode besitzt, können wir

```
leaf.turn(9);
```

aufrufen. Dies setzt voraus, dass wir eine Variable namens **leaf** besitzen, die eine Referenz auf ein **Leaf**-Objekt hält. Die Zahl 9 ist ein willkürlicher Wert, um das Blatt ein bisschen zu drehen.

Wir können sehen, dass das Aufrufen einer Methode eines Akteur-Objekts genauso abläuft wie beim Welt-Objekt. Der Unterschied besteht darin, woher wir die Referenz auf das Blatt bekommen.

Eine Möglichkeit, Referenzen auf andere Akteure in der Welt zu erhalten, ist über die *Kollisionserkennungsmethoden* von Greenfoot. Dies sind Methoden, die prüfen, ob sich unser Akteur mit einem anderen Akteur überschneidet, und die uns gegebenenfalls eine Referenz zu diesem anderen Akteur zurückliefern. Sobald wir eine Referenz auf das Blatt haben, das wir berühren, ist der Rest ist einfach.

Wir werden eine Methode namens **`getOneIntersectingObject`** verwenden. Diese Methode gibt uns eine Referenz auf den Akteur, mit dem wir uns überschneiden. (Es gibt noch verschiedene andere Kollisionserkennungsmethoden; wir werden in ***Kapitel 9*** noch einmal darauf zurückkommen. Eine vollständige Liste findest du in ***Anhang C***).

Wir rufen unsere Methode folgendermaßen auf:

```
Leaf leaf = (Leaf) getOneIntersectingObject(leaf.class);
```

Übung 7.4 Suche die Methode **`getOneIntersectingObject`** in der Klassendokumentation von Greenfoot. Zu welcher Klasse gehört die Methode? Welchen Rückgabetyp hat sie?

Wir wollen die obige Codezeile noch ein wenig analysieren:

- Wir rufen die Methode **`getOneIntersectingObject`** auf. Über den Parameter können wir spezifizieren, an welchen Arten von Objekten wir interessiert sind. Wir müssen eine Klasse angeben. Durch die Wahl von **`Leaf.class`** als Parameter drücken wir aus, dass wir prüfen möchten, ob wir uns mit irgendeinem **`Leaf`**-Objekt überschneiden.
- Auf der linken Seite der Gleichung deklarieren wir eine neue lokale Variable **`leaf`** (vom Typ **`Leaf`**), die danach bereit ist, ein **`Leaf`**-Objekt aufzunehmen.
- Wenn die Methode **`getOneIntersectingObject`** ein überschneidendes Blatt findet, dann gibt sie eine Referenz auf dieses **`Leaf`**-Objekt zurück und wir weisen es unserer **`leaf`**-Variablen zu.
- Wir müssen eine Typanpassung vornehmen: **`(Leaf)`**. Dies liegt daran, dass die Methode **`getOneIntersectingObject`** laut ihrer Deklaration einen **`Actor`**, kein **`Leaf`** zurückgibt. Wir müssen unserem Compiler mitteilen, dass wir hier aber ein Objekt vom Typ **`Leaf`** erwarten.

7.5 Der Wert null

Wir haben gesehen, was passiert, wenn unsere Methode ein überschneidendes Blatt findet: Sie gibt eine Referenz zurück und wir speichern diese in einer Variablen. Was passiert aber, wenn wir uns mit keinem Blatt überschneiden?

Wenn wir **`getOneIntersectingObject`** aufrufen, uns aber im Moment mit keinem Objekt des angefragten Typs überschneiden, dann gibt die Methode den speziellen Wert **`null`** zurück.

Der **null**-Wert kann jeder beliebigen Objektvariablen zugewiesen werden. Wenn er in einer Variablen gespeichert ist, bedeutet das, dass die Variable aktuell keine Referenz hält – im Prinzip ist sie leer.

Die Methode **getOneIntersectingObject** gibt entweder **null** oder eine Objektreferenz zurück, je nachdem, ob wir uns mit einem Objekt überschneiden oder nicht. Wir können nach dem Aufruf überprüfen, ob wir wirklich irgendein Objekt berühren:

```
Leaf leaf = (Leaf) getOneIntersectingObject(Leaf.class);
if (leaf != null)
{
    // wenn wir hier sind, dann berühren wir ein Blatt
}
```

Konzept

Der Wert **null** ist ein spezieller Wert, der jeder Objektvariablen zugewiesen werden kann. Enthält eine Variable **null**, dann hat sie aktuell keine Referenz auf ein Objekt.

In diesem Codesegment untersuchen wir die **leaf**-Variable: Ist sie nicht **null**, dann wissen wir, dass wir ein Blatt berühren, und die Referenz ist in unserer Variablen gespeichert. Somit können wir diese Aufgabe vervollständigen, indem wir im Rumpf der **if**-Anweisung dafür sorgen, dass sich das Blatt dreht:

```
leaf.turn(9);
```

Übung 7.5 Erzeuge in deiner Klasse **Block** eine neue private Methode namens **checkLeaf()**. Die Methode gibt keinen Wert zurück. Gib hier den oben abgebildeten Code ein, um zu prüfen, ob wir uns mit einem Blatt überschneiden. Ist dies der Fall, dann weise das Blatt an, sich zu drehen.

Übung 7.6 Stelle sicher, dass du tatsächlich auf Blätter überprüfst. Rufe deine **chekLeaf()**-Methode aus deiner **act**-Methode heraus auf. Teste deinen Code.

Übung 7.7 Entferne deinen Code, der Blätter hinzufügt, wenn der Block an die Ränder der Welt stößt. Schreibe stattdessen Code in der **setUp()**-Methode der **MyWorld**-Klasse, der 18 Blätter hinzufügt. Verwende dazu eine **while**-Schleife. Füge die Blätter an zufälligen Positionen ein. Teste deinen Code.

7.6 Interaktionen mit Gruppen von Akteuren

Wir haben nun eine Möglichkeit gesehen, eine Referenz auf einen anderen Akteur zu erhalten und dann mit diesem Akteur zu kommunizieren, indem wir eine seiner Methoden aufrufen.

Der nächste interessante Fall ist, mit mehreren Akteuren auf einmal zu kommunizieren: Manchmal möchten wir etwas mit jedem Akteur einer bestimmten Klasse machen oder mit jedem Akteur, der sich innerhalb eines bestimmten Bereichs um uns herum befindet.

Das Beispiel, das wir dazu benutzen, wirkt ein wenig konstruiert, es dient nur dem Zweck, dieses Konzept einzuüben: Wenn der Benutzer mit der Maus klickt, dann möchten wir, dass sich die Bilder aller Blätter verändern.

Übung 7.8 Die Klasse **Leaf** besitzt eine Methode namens **changeImage()**, mit der ein anderes Bild verbunden werden kann. Probiere die Methode aus. Pausiere dein Szenario, rufe die **changeImage()**-Methode interaktiv für einige Blätter auf (indem du mit einem Rechtsklick das **Leaf**-Menü öffnest und die Methode auswählst).

Die Klasse **Block** hat bereits eine Methode zum Testen auf Mausklicks (Listing 7.1). Wir können unseren Code hier eingeben. Was wir jetzt noch tun müssen, ist herauszufinden, wie die **changeImage()**-Methode für alle Blätter in der Welt aufgerufen wird.

Listing 7.1: Eine Methode, um auf Mausklicks zu testen.

```
/**
 * Prüft, ob die Maustaste geklickt wurde. Falls ja, werden alle Blätter verändert.
 */
private void checkMouseClick()
{
    if (Greenfoot.mouseClicked(null))
    {
        // auszuführende Aktion, wenn die Maus geklickt wurde. Aktuell: nichts
    }
}
```

Die **World**-Klasse von Greenfoot besitzt Methoden, die uns Zugriff auf Objekte ermöglichen, die sich in der Klasse befinden.

Übung 7.9 Suche die **World**-Klasse von Greenfoot in der Klassendokumentation. Finde alle Methoden, die uns Zugriff auf Objekte innerhalb der Welt geben. Schreibe sie auf.

Die für uns interessanteste Methode ist diese:

```
java.util.List getObjects(java.lang.Class cls)
```

Diese Methode liefert uns eine Liste aller Objekte einer bestimmten Klasse in der Welt. Sie verwendet zwei Typen, die ungewöhnlich aussehen: **java.lang.Class** als Parameter und **java.util.List** als Rückgabewert. Um dies richtig zu verstehen, müssen wir uns zwei Dinge angucken: Java-Klassenbibliotheken und den Typ **List**.

7.7 Die Verwendung der Java-Bibliotheksklassen

Die Klassen **Class** und **List** sind zwei der vielen Klassen aus der *Java-Klassenbibliothek*.[1] Zum Java-System gehört eine umfangreiche Sammlung von nützlichen Klassen, auf die wir selbstverständlich Zugriff haben. Mit der Zeit werden wir etliche davon kennenlernen.

Über den Befehl JAVA KLASSENBIBLIOTHEKEN im Greenfoot-Menü HILFE öffnest du die Dokumentation der Java-Bibliotheken in einem eigenen Browser-Fenster. Hier findest du englischsprachige Beschreibungen zu allen Klassen der Klassenbibliotheken (Abbildung 7.3).

Konzept

Die **Java-Klassenbibliotheken** bilden eine umfangreiche Sammlung von vordefinierten Klassen, die untrennbar mit dem Java-System verbunden sind. Wir können mithilfe einer **import**-Anweisung auf diese Klassen zugreifen.

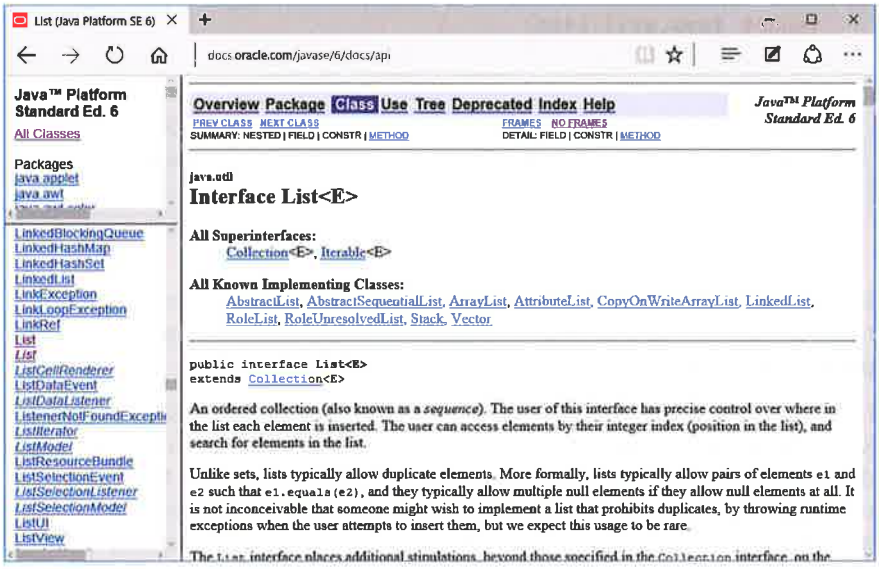

Abbildung 7.3
Die Java-Klassenbibliotheken.

Der Bereich links unten im Browser-Fenster zeigt eine Liste aller Klassen der Java-Bibliotheken. (Es gibt sehr viele Klassen!) Wir können die Beschreibung jeder beliebigen Klasse einsehen, indem wir hier nach der Klasse suchen und ihren Namen anklicken. Anschließend wird im Hauptbereich die Beschreibung der angeklickten Klasse angezeigt.

Übung 7.10 Suche in dieser Klassenliste die Klasse **Color**. Klicke die Klasse an und wirf einen Blick auf die Beschreibung dieser Klasse. Wie viele Konstruktoren hat sie?

1 **List** ist eigentlich eine *Schnittstelle* (*Interface*), keine Klasse – wir kommen später noch darauf zurück.

Wie du siehst, gibt es im wahrsten Sinne des Wortes Tausende von Klassen in der Java-Bibliothek. Um etwas Ordnung in diese lange Liste zu bringen, werden Klassen zu *Paketen* zusammengefasst. Ein Paket ist eine Gruppe von logisch zusammenhängenden Klassen. Ganz oben in der Beschreibung einer jeden Klasse können wir nachlesen, zu welchem Paket die Klasse gehört. Die Klasse **`Color`** befindet sich zum Beispiel in einem Paket namens **`java.awt`**.

Wenn wir eine der Klassen aus der Java-Bibliothek in unserem eigenen Szenario verwenden möchten, müssen wir sie zuerst mit einer Import-Anweisung *importieren*. Die Import-Anweisung wird ganz oben in Quelltext der Klasse geschrieben und gibt das Paket und, durch einen Punkt getrennt, die darin enthaltene Klasse an, die wir verwenden wollen. Um zum Beispiel den **`List`**-Typ in unserem Code zu verwenden, der sich im Paket **`java.util`** befindet, schreiben wir

```
import java.util.List;
```

Übung 7.11 Schreibe die Import-Anweisung für **`List`** in unserer eigenen **`Block`**-Klasse, direkt unter die Import-Anweisung für **`greenfoot.*`**, die dort bereits steht.

Mithilfe des Sternchens (*) können alle Klassen eines gegebenen Pakets importiert werden. Mit **`greenfoot.*`** werden zum Beispiel alle Klassen aus dem **`greenfoot`**-Paket importiert.

Durch das Importieren einer Klasse kannst du diese Klasse in deinem eigenen Szenario verfügbar machen – als wäre sie eine deiner eigenen Klassen. Nach dem Import können wir Variablen dieses Typs deklarieren, ihre Methoden aufrufen und alles damit machen, was wir auch mit anderen Klassen machen können. Wir könnten also jetzt in unserem eigenen Code eine Variable vom Typ **`List`** deklarieren:

```
List myList;
```

Das Paket **`java.lang`** ist speziell: Es enthält die am häufigsten benutzten Klassen und wir müssen dieses Paket nicht explizit importieren – es wird stets automatisch importiert. Man muss also keine Import-Anweisung für **`java.lang.Class`** schreiben.

Die Java-Bibliothek ist am Anfang vielleicht etwas einschüchternd, weil sie so viele Klassen enthält. Doch keine Sorge – wir werden nur einige wenige davon nutzen und diese erst dann einführen, wenn wir sie brauchen.

Da wir nun wissen, wie man den **`List`**-Typ importiert, wollen wir uns genauer ansehen, wie man ihn verwendet.

7.8 Der Typ List

Konzept

Eine **Sammlung** (*collection*) ist eine spezielle Art Objekt, das viele andere Objekte halten kann.

Konzept

`List` ist ein Beispiel für eine Sammlung. Einige Methoden der Greenfoot API liefern `List`-Objekte zurück.

Sammlungen von Objekten sind nicht nur für die Greenfoot-Programmierung, sondern für die Programmierung im Allgemeinen wichtig. Es gibt mehrere Greenfoot-Methoden, die Sammlungen von Objekten als Ergebnis zurückliefern, und zwar in der Regel in Form einer Liste. In diesem Fall ist der Typ des zurückgelieferten Objekts der Typ `List` aus dem Paket `java.util`.

Randbemerkung: Schnittstellen

Der Typ `List` unterscheidet sich ein wenig von den anderen Objekttypen, die wir bisher kennengelernt haben: Er ist keine Klasse, sondern eine *Schnittstelle*. Schnittstellen sind ein Java-Konstrukt, das eine Abstraktion verschiedener möglicher Implementierungsklassen darstellt. Die Einzelheiten sollen uns im Moment noch nicht interessieren – es reicht, zu wissen, dass wir den Typ `List` genauso wie alle anderen Typen verwenden können: Wir können ihn in den Java-Klassenbibliotheken nachschlagen und wir können die bestehenden Methoden für das Objekt aufrufen. Was wir nicht können, ist, direkt Objekte vom Typ `List` zu erzeugen. Doch dazu später mehr.

Übung 7.12 Suche nach `java.util.List` in den Java-Klassenbibliotheken. Welche Methoden gibt es, um ein Objekt zur Liste hinzuzufügen, aus der Liste zu entfernen und um festzustellen, wie viele Objekte gerade in der Liste sind?

Übung 7.13 Wie lautet der vollständige Name dieses Typs, wie er oben in der Dokumentation angezeigt wird?

Wie wir in einem früheren Abschnitt feststellen konnten, liefert die Methode `getObjects` ein Objekt vom Typ `java.util.List` zurück. Damit wir dieses Objekt speichern können, müssen wir eine Variable dieses Typs deklarieren. Dies wollen wir in der Methode `checkMouseClick` tun.

Der Typ `List` unterscheidet sich jedoch von den Typen, die wir bisher kennengelernt haben. In der Dokumentation steht oben

```
Interface List<E>
```

Abgesehen von dem Wort `Interface`, das anstelle von `class` verwendet wird, fällt noch eine weitere neue Notation ins Auge: das `<E>` nach dem Typnamen.

Konzept

Ein **generischer Typ** ist ein Typ, der einen zweiten Typnamen als Parameter übernimmt.

Formal wird dies auch als *generischer Typ* bezeichnet. Das bedeutet, dass der Typ **`List`** einen zusätzlichen Typ als Parameterangabe benötigt. Der zweite Typ gibt den Typ der Elemente an, die in der Liste verwahrt werden.

Wenn wir es zum Beispiel mit einer Liste von Strings zu tun haben, würden wir den Typ als

```
List<String>
```

angeben. Wenn wir es stattdessen mit einer Liste von Akteuren zu tun haben, schreiben wir:

```
List<Actor>
```

In beiden Fällen ist der Elementtyp in spitzen Klammern (<>) der Typ einer anderen bekannten Art von Objekt.

7.9 Eine Blätterliste

In unserem Fall müssen wir die Methode **`getObjects`** unserer Welt aufrufen, um eine Liste von Blättern zu erhalten, weshalb unsere Variablendeklaration wie folgt lautet:

```
List<Leaf> leaves
```

Dann können wir die Liste, die wir von der Methode **`getObjects`** erhalten, dieser Variablen zuweisen. Zusammen mit der Anweisung für das Holen des **`World`**-Objekts selbst erhalten wir:

```
World world = getWorld();
List<Leaf> leaves = world.getObjects(Leaf.class);
```

Nach Ausführung dieser Zeile hält unsere Variable **`leaves`** eine Liste aller Blätter, die sich zurzeit in der Welt befinden. Als Letztes müssen wir noch die Methode **`changeImage()`** für jedes Blatt in unserer Liste aufrufen. Wir werden dazu eine neue Schleifenart verwenden: die *for-each*-Schleife.

7.10 Die for-each-Schleife

Konzept

Die **for-each-Schleife** ist eine weitere Schleifenvariante. Sie eignet sich besonders gut, um alle Elemente einer Sammlung zu verarbeiten.

Wir wollen nun unsere Liste der Blätter durchlaufen und der Reihe nach die Bilder der einzelnen Blätter ändern.

Java bietet eine besondere Schleife, um die einzelnen Elemente einer Sammlung zu durchlaufen, die wir hier wunderbar einsetzen können. Sie wird *for-each-Schleife* genannt und weist die folgende allgemeine Syntax auf:

```
for (ElementTyp variable : sammlung)
{
    anweisungen;
}
```

In dieser Syntax steht ***ElementTyp*** für den Typ eines jeden Elements in der Sammlung, ***variable*** ist eine Variable, die hier deklariert wird, sodass wir ihr einen beliebigen Namen geben können, ***sammlung*** ist der Name der Sammlung, die wir verarbeiten wollen, und ***anweisungen*** steht für eine Folge von Anweisungen, die ausgeführt werden sollen. Ein Beispiel soll dies veranschaulichen.

Unter Verwendung unserer Liste **leaves** können wir schreiben

```
for (Leaf leaf : leaves)
{
    leaf.changeImage();
}
```

(Denk daran, dass in Java zwischen Groß- und Kleinschreibung unterschieden wird: **Leaf** mit einem „großen L" unterscheidet sich von **leaf** mit einem „kleinen l". Der großgeschriebene Name bezieht sich auf die Klasse, der kleingeschriebene Name auf eine Variable, die ein Objekt verwahrt. Die Pluralversion – **leaves** – ist eine weitere Variable, die die ganze Liste verwahrt.)

Wir können die **for-each**-Schleife etwas leichter lesen, wenn wir das Schlüsselwort **for** als „für jedes" lesen, den Doppelpunkt als „in" und die öffnende geschweifte Klammer als „tue". Dann wird aus der Schleife:

```
für jedes leaf in leaves tue: ...
```

Diese Lesart gibt uns einen Hinweis darauf, was die Schleife macht: Sie führt die Anweisungen in den geschweiften Klammern für jedes Element in der Liste **leaves** einmal aus. Wenn es zum Beispiel zehn Elemente in dieser Liste gibt, werden die Anweisungen genau zehnmal ausgeführt. Jedes Mal, bevor die Anweisungen ausgeführt werden, wird der Variablen **leaf** (die im Schleifenkopf deklariert ist) eines der Listenelemente zugewiesen. Die Aktionen lauten also nacheinander

```
leaf = erstes Element aus 'leaves';
führe Schleifenanweisungen aus;
leaf = zweites Element aus 'leaves';
führe Schleifenanweisungen aus;
leaf = drittes Element aus 'leaves';
führe Schleifenanweisungen aus;
...
```

Die Variable **leaf** steht zur Verfügung, um in den Schleifenanweisungen auf das aktuelle Element aus der Liste zugreifen zu können. Wir könnten dann zum Beispiel wie oben eine Methode für das Objekt aufrufen oder das Objekt einer anderen Methode zur Weiterverarbeitung übergeben.

Jetzt sind wir so weit, alle Einzelteile zusammenzufügen und die **for-each**-Schleife in unseren Code einzubauen. Listing 7.2 zeigt die vollständige Methode, die das gerade besprochene Beispiel implementiert.

Listing 7.2: Die Bilder aller Blätter in der Welt ändern.

```
/**
 * Prüft, ob die Maustaste geklickt wurde. Falls ja, werden alle Blätter verändert.
 */
private void checkMouseClick()
{
    if (Greenfoot.mouseClicked(null))
    {
        World world = getWorld();
        List<Leaf> leaves = world.getObjects(Leaf.class);

        for (Leaf leaf : leaves)
        {
            leaf.changeImage();
        }
    }
}
```

Übung 7.14 Implementiere in deinem Szenario die Funktionalität, alle Blätter mit einem Mausklick zu verändern. Test deinen Code.

Wenn du dir mit deiner Lösung nicht ganz sicher bist und diese mit unserer vergleichen möchtest: das Szenario *autumn-2* enthält alle Funktionen, die wir in diesem Kapitel besprochen haben.

Zusammenfassung der Programmiertechniken

Wir sind in diesem Kapitel einigen wichtigen Konstrukten begegnet, bei denen es zunächst nicht ganz einfach ist, sie vollständig zu verstehen. Du wirst einige Zeit (und viel Übung) brauchen, um dich an sie zu gewöhnen.

Der erste Teil war relativ leicht: Wir haben gesehen, wie man Zugriff auf Welt-Objekte erhält und deren Methoden aufruft. Dann haben wir besprochen, wie man auf einen anderen Akteur zugreift und eine seiner Methoden aufruft. Dazu haben wir eine der Kollisionserkennungsmethoden von Greenfoot benutzt.

Zum Schluss – und das war der schwierige Teil – haben wir gesehen, wie man eine Liste von mehreren Objekten erhält. Wir haben eine **for-each**-Schleife eingesetzt, um eine Methode für alle Objekte in der Liste aufzurufen.

Das Arbeiten mit Listen und der Umgang mit ihren Elementen ist ein sehr wichtiges Konzept, nicht nur in Greenfoot, sondern allgemein in der Programmierung. Dies war nur ein kleiner Blick auf ein erstes Beispiel. Wir werden davon mehr in den folgenden Kapiteln sehen und du wirst hoffentlich vertrauter mit diesen Techniken werden, wenn du sie immer wieder anwendest.

Zusammenfassung der Konzepte

- Der Wert **`null`** ist ein spezieller Wert, der jedem Objekt zugewiesen werden kann. Enthält eine Variable **`null`**, dann hält sie aktuell keine Referenz auf ein Objekt.
- Die **Java-Klassenbibliotheken** bilden eine umfangreiche Sammlung von vordefinierten Klassen, die untrennbar mit dem Java-System verbunden sind. Wir können mithilfe einer **`import`**-Anweisung auf diese Klassen zugreifen.
- Eine **Sammlung** (*collection*) ist eine spezielle Art Objekt, das viele andere Objekte halten kann.
- **`List`** ist ein Beispiel für eine Sammlung. Einige Methoden der Greenfoot API liefern **`List`**-Objekte zurück.
- Ein **generischer Typ** ist ein Typ, der einen zweiten Typnamen als Parameter übernimmt.
- Die **`for-each-Schleife`** ist eine weitere Schleifenvariante. Sie eignet sich besonders gut, um alle Elemente einer Sammlung zu verarbeiten.

Vertiefende Aufgaben

Die wichtigsten Konzepte, die wir hier weiter einüben wollen, sind die Arbeit mit Listen und die **for-each**-Schleife. Dazu haben wir dir ein paar Übungen zusammengestellt. Wir werden weiterhin das Szenario *autumn* dafür einsetzen.

Übung 7.15 Erstelle in deinem *autumn*-Szenario eine neue Klasse namens **Apple**. Weise der Klasse ein Bild zu. Schreibe Code in der Klasse **MyWorld**, um 12 Äpfel an zufälligen Positionen zu platzieren.

Übung 7.16 Schreibe Code in deiner **Block**-Klasse, um alle Äpfel jedes Mal um 90 Grad zu drehen, wenn der Block an den Rand der Welt kommt.

Übung 7.17 Erzeuge noch eine weitere Klasse namens **Pear** (Birne). Wähle ein passendes Bild dafür. Schreibe Code, um 8 Birnen in der Welt zu platzieren.

Übung 7.18 Schreibe Code in deiner **Block**-Klasse, sodass sich alle Birnen um 20 Zellen nach rechts bewegen, sobald der Block den Rand berührt (die Äpfel sollen sich weiterhin drehen).

Übung 7.19 Verändere den Code für die Bewegung der Birnen, sodass sich die Birnen nur dann nach rechts bewegen, wenn sie nicht bereits am rechten Rand sind. Falls sie dort schon sind, werden sie stattdessen zum linken Rang bewegt.

Übung 7.20 Verändere den Code, der für die Mausklicks zuständig ist, sodass sich nur die Bilder von den Blättern in der linken Hälfte der Welt (und nicht mehr alle) ändern.

Diese Übungen sollten dich ausreichend auf die nächsten Kapitel vorbereitet haben. Wir werden jetzt weitergehen und uns das nächste Szenario ansehen. Wir bekommen mit diesem Beispiel wieder die Gelegenheit, den Umgang mit Listen einzuüben.