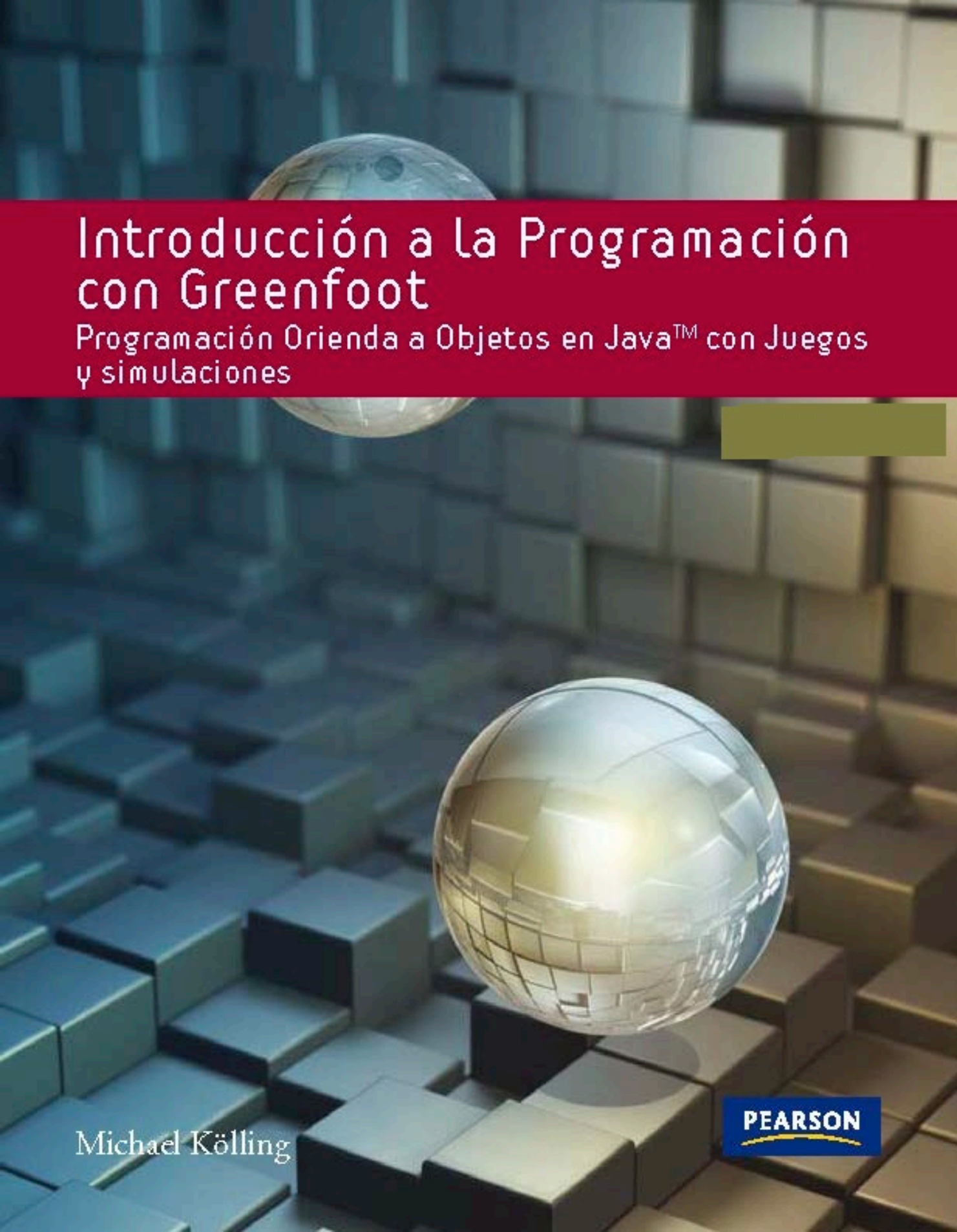

Introducción a la Programación con Greenfoot
Programación Orienda a Objetos en Java™ con Juegos y simulaciones
Michael Kölling
PEARSON

Introducción a la Programación con Greenfoot

Programación Orienta a Objetos en Java™ con Juegos y Simulaciones

Michael Kölling

TRADUCCIÓN Y REVISIÓN TÉCNICA

Carlos A. Iglesias
Departamento de Ingeniería de Sistemas Telemáticos
Universidad Politécnica de Madrid

REVISIÓN TÉCNICA PARA LATINOAMÉRICA

Ingeniero Marcelo Giura
Titular Informática II
Secretario Académico UTN
UTN Regional Buenos Aires

Ingeniero Marcelo Ángel Trujillo
Profesor asociado Informática I y II
UTN Regional Buenos Aires

Mag. Ingeniero Santiago Cristóbal Pérez
Profesor Asociado Arquitectura de Computadoras
UTN Regional Mendoza
Universidad Nacional de Chilecito (La Rioja)

Ingeniero Atilio Ranzuglia
JTP Programación Avanzada y Diseño de Sistemas
UTN Regional Mendoza

Prentice Hall
es un sello editorial de

PEARSON

Harlow, England • London • New York • Boston • San Francisco • Toronto • Sydney • Singapore • Hong Kong
Tokyo • Seoul • Taipei • New Delhi • Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Datos de catalogación bibliográfica

INTRODUCCIÓN A LA PROGRAMACIÓN CON GREENFOOT. Programación orientada a Objetos en Java™ con Juegos y Simulaciones
Michael Kölling

PEARSON EDUCACIÓN, S.A. 2011
ISBN: 978-84-8322-766-4

Materia: 004, Informática

ISBN: 978-84-8322-766-4

Depósito Legal:

Equipo editorial:
Editor: Miguel Martín-Romo
Técnico Editorial: Esther Martín
Equipo de producción:
Director: José A. Clares
Técnico: Irene Iriarte
Diseño de cubierta: COPIBOOK, S. L.
Composición:
Impreso por:

IMPRESO EN ESPAÑA - *PRINTED IN SPAIN*
Este libro ha sido impreso con papel y tintas ecológicos

A Krümel y Cracker—su imaginación no puede nunca desvanecerse.

—mk

La educación no es rellenar un cubo, sino encender un fuego.

—William Butler Yeats

Contenido

Lista de escenarios tratados en este libro

Hojas y Wombátidos (Capítulo 1)

Éste es un ejemplo sencillo que muestra cómo los wombátidos se mueven por la pantallas, comiendo hojas ocasionalmente. El escenario no tiene ningún otro propósito que ilustrar algunos conceptos importantes de la orientación a objetos así como las interacciones con Greenfoot.

Asteroides 1 (Capítulo 1)

Ésta es una versión simplificada del juego clásico de arcade. Vuelas una nave espacial e intentas evitar ser golpeado por asteroides. En esta etapa, sólo usamos el escenario para hacer algunos pequeños cambios e ilustrar algunos conceptos básicos.

Cangrejito (Capítulo 2)

Éste es nuestro primer desarrollo completo. A partir de casi nada, desarrollamos un juego simple poco a poco, añadiendo muchas cosas tales como movimiento, control del teclado, sonido y muchos otros elementos típicos de los juegos.

Piano (Capítulo 5)

Un piano virtual donde puedes tocar realmente.

El laboratorio de Newton (Capítulo 6)

El laboratorio de Newton es una simulación de los movimientos de las estrellas y los planetas en el espacio. La gravedad juega un papel central aquí. También hacemos una variante de esto que combina la gravedad con la creación de música, terminando con una salida musical activada por los objetos bajo movimientos gravitatorios.

Asteroides 2 (Capítulo 7)

Volvemos al ejemplo de asteroides del Capítulo 2. Esta vez, investigamos con más detalle cómo implementarlo.

Hormigas (Capítulo 9)

Aúna simulación de colonias de hormigas buscando comida, y comunicándose mediante gotas de feromonas dejadas en la tierra.

Los siguientes escenarios se presentan en el Capítulo 10 y algunos aspectos seleccionados se describen brevemente. Están pensados como inspiración para otros proyectos.

Canicas

Una simulación del juego de las canicas. Las canicas deben sacarse del tablero en un número limitado de movimientos. Contiene física simple.

Ascensores

Un inicio de una simulación de un ascensor. Está incompleto en esta fase —puede ser usado como comienzo de un proyecto.

Boids
Una demostración de comportamiento gregario: una bandada de aves vuela por la pantalla, con el objetivo de permanecer unidas mientras evitan obstáculos.

Círculos
Realiza patrones en diferentes colores en la pantalla con círculos en movimiento.

Explosión
Una demostración de un efecto de explosión más sofisticado.

Breakout
Éste es el inicio de una implementación del juego clásico Breakout. Muy incompleto, pero con unos efectos visuales interesantes.

Salto de plataformas
Una demostración de una implementación parcial de un género siempre popular de videojuegos: salto de plataformas.

Onda
Este escenario es una demostración simple de un efecto físico: la programación de una onda en una cadena.

Prefacio

Greenfoot es un entorno de programación que puede ser usado tanto para auto-aprendizaje, como en escuelas o en cursos universitarios introductorios para aprender y enseñar los principios de programación. Es suficientemente flexible para ser adecuado para adolescentes o estudiantes mayores.

Greenfoot soporta el lenguaje de programación Java, de forma que los estudiantes aprenden de forma convencional programación orientada a objetos en Java. El entorno ha sido diseñado específicamente para cubrir los conceptos y principios orientados a objetos de una forma limpia y asequible.

El entorno Greenfoot hace que la creación de gráficos e interacciones sea fácil. Los estudiantes pueden concentrarse en modificar la lógica de la aplicación, y combinar y experimentar con los objetos. El desarrollo de simulaciones y juegos interactivos se convierte en algo sencillo, que proporciona resultados de forma inmediata.

El entorno ha sido diseñado para atraer rápidamente a estudiantes que pueden no tener conocimientos previos o experiencia en programación. Se consigue rápidamente realizar animaciones simples, y es posible realizar escenarios con un aspecto profesional.

Accediendo a materiales complementarios

Las ayudas al aprendizaje y materiales complementarios citados en el libro pueden ser accedidos a través del sitio web complementario o a través del sitio web de la editorial:

Sitio web complementario: http://www.greenfoot.org/book/
Sitio web de la editorial: http://www.prenhall.com/kolling

Materiales complementarios disponibles para los estudiantes

Los siguientes complementos están disponibles para los estudiantes:

- El software de Greenfoot.
- Los escenarios tratados en este libro.
- La galería de Greenfoot —un sitio público de escenarios.
- Videotutoriales.
- Un foro de discusión.
- Soporte técnico.

Materiales complementarios disponibles para instructores

Los siguientes complementos están disponibles para instructores:

- Un foro de discusión de profesores.
- Ejercicios adicionales relacionados con el libro.
- El sitio web «Green Room» que contiene ejercicios y otros recursos.

Para más detalles sobre Greenfoot y este libro, por favor, mira la Introducción después de los Agradecimientos.

Agradecimientos

Este libro es el resultado de más de cinco años de trabajo de un grupo de personas. En primer lugar, y de forma más destacada, están las personas que han contribuido al desarrollo del entorno Greenfoot, que hace que sea posible este enfoque educativo. Paul Henriksen comenzó la implementación de Greenfoot como su proyecto de Master y desarrolló el primer prototipo. Él también aceptó el reto de pasar este prototipo a un sistema en producción. Durante el primer año o así, fuimos un equipo de dos personas, y el trabajo de Poul condujo a la calidad y robustez del sistema actual.

Bruce Quig y Davin McCall fueron los siguientes desarrolladores que se unieron al proyecto, y Poul, Bruce, y Davin desarrollaron conjuntamente la mayoría de lo que es Greenfoot hoy. Los tres son desarrolladores software excepcionales, y no puede dejar de recalcarse la importancia de su contribución al proyecto. Es un placer trabajar con ellos.

Finalmente, el grupo entero BlueJ se involucró en el proyecto Greenfoot, incluyendo a John Rosenberg y Ian Utting, y este libro se ha elaborado a partir de las contribuciones y trabajo común de todos los miembros del grupo.

Los colegas del Laboratorio de Computación de la Universidad de Kent también me han ayudado en gran medida, especialmente nuestro director de departamento Simon Thompson, que vio el valor de Greenfoot desde el principio y nos ha apoyado y motivado para que continuáramos su desarrollo.

Otra contribución importante, sin la que el desarrollo de Greenfoot (y en última instancia, este libro) no hubiera sido posible, es el generoso apoyo de Sun Microsystems. Emil Sarpa, Katherine Hartsell, Jessica Orquina, Sarah Hammond, y muchos otros en Sun creyeron en el valor de nuestro sistema y nos proporcionaron apoyo importante.

Todo el mundo en Pearson Education trabajaron duramente para conseguir que este libre se publicara a tiempo, con una planificación muy ajustada, e incluso a veces en circunstancias difíciles. Tracy Dunkelberger trabajó conmigo en este libro desde el principio. Ella gestionó asombrosamente bien el mantener una actitud positiva y entusiasta a la vez que soportaba mis reiterados retrasos en las entregas, y aún me animaba para continuar escribiendo. Melinda Haggerty hizo un montón de cosas diferentes, incluyendo la gestión de las revisiones.

Un especial agradecimiento debe ir a los revisores del libro, que han proporcionado comentarios muy detallados, meditados y útiles. Han sido Carolyn Oates, Damianne President, Detlef Rick, Gunnar Johannesmeyer, Josh Fishburn, Mark Hayes, Marla Parker, Matt Jadud, Todd O'Bryan, Lael Grant, Jason Green, Mark Lewis, Rodney Hoffman y Michael Kadri. Han ayudado señalando muchos errores así como oportunidades de mejora.

Mi buen amigo Michael Caspersen también merece un agradecimiento por proporcionarme comentarios desde el principio y animarme, lo que ha sido muy importante para mí, porque ayudó a mejorar el libro, y, aún más importante, porque me animó a creer que este trabajo podría ser interesante a los profesores y merecía la pena terminarlo.

Introducción

¡Bienvenido a Greenfoot! En este libro, aprenderemos cómo programar programas gráficos, tales como simulaciones o juegos, usando el lenguaje de programación Java y el entorno Greenfoot.

Hay varios objetivos para hacer esto: uno es aprender a programar, otro es divertirse en el camino. Aunque los ejemplos que emplearemos en el libro son específicos del entorno de programación Greenfot, los conceptos son generales. Es decir, mientras avanzas con el libro aprenderás principios generales de programación, siguiendo un lenguaje moderno orientado a objetos. Además, aprenderás cómo realizar tu propio juego de ordenador, una simulación biológica o un piano virtual.

Este libro sigue una orientación muy práctica. Los capítulos y ejercicios están estructurados mediante tareas prácticas de desarrollo. En primer lugar, se presenta un problema que debemos resolver, y a continuación se muestran los constructores del lenguaje y las estrategias que nos ayudan a resolver el problema. Este enfoque es bastante diferente de muchos libros de programación que a menudo se estructuran siguiendo el orden de los constructores del lenguaje.

Como resultado, este libro comienza con menos teoría y más actividades prácticas que la mayoría de libros de programación. Ésta es también la razón por la que usamos Greenfoot, que es el entorno que lo hace posible. Greenfoot nos permite jugar. Y eso no se limita a jugar con juegos de ordenador, significa también jugar con la programación: podemos crear objetos, moverlos en la pantalla, llamar a sus métodos, y observar lo que hacen, todo de forma fácil e interactiva. Esto conduce a un enfoque más práctico a la programación que no sería posible sin contar con este entorno.

Un enfoque más práctico no significa que el libro no cubra también la teoría y principios necesarios de programación. Simplemente se ha cambiado el orden. En vez de introducir un concepto teóricamente primero, y luego hacer ejercicios, a menudo vamos a emplear directamente un constructor, explicando sólo lo que sea imprescindible para realizar la tarea, volviendo más tarde con el soporte teórico. Seguiremos normalmente un enfoque en espiral: introducimos algunos aspectos de un concepto la primera vez que lo encontramos, luego volvemos a revisarlo más tarde en otro contexto, y profundizamos en su comprensión de forma gradual.

El objetivo es hacer que el trabajo que hagamos sea interesante, relevante y divertido. No hay ninguna razón por la que la programación tenga que ser árida, formal o aburrida. Es conveniente divertirse mientras aprendemos. Creemos que debemos conseguir que la experiencia de aprender sea interesante y pedagógicamente consistente a la vez. Esto es un enfoque que ha sido llamado *diversión seria* —hacemos algo interesante, y aprendemos algo útil a la vez.

Este libro puede ser usado como libro de auto-estudio al igual que como libro de texto en un curso de programación. Los ejercicios se desarrollan a lo largo del libro —si los haces todos, acabarás siendo un programador bastante competente.

Los proyectos que se desarrollan en el libro son suficientemente sencillos como para que puedan ser resueltos por estudiantes de secundaria, pero también son suficientemente abiertos y extensibles como para que incluso programadores avezados puedan encontrar aspectos interesantes y desafiantes para realizarlos. Aunque Greenfoot es un entorno educativo, Java no es un lenguaje de juguete. Como

Java es el lenguaje elegido en este libro, los proyectos desarrollados (y cualquier otro que quieras crear con Greenfoot) pueden ser tan complejos y complicados como quieras.

Aunque es posible crear juegos rápida y fácilmente en Greenfoot, es igualmente posible construir simulaciones muy sofisticadas de sistemas complejos, pudiendo emplear algoritmos de inteligencia artificial, tecnología de agentes, conectividad con base de datos, o cualquier otra tecnología en la que puedas pensar. Java es un lenguaje muy completo que te permite usar todo el potencial del mundo de la programación, y Greenfoot no impone ninguna restricción en qué aspectos del lenguaje puedes usar.

En otras palabras, Greenfoot escala bien. Permite empezar a programar fácilmente a los jóvenes programadores, pero también permite que programadores experimentados puedan implementar escenarios interesantes y sofisticados.

¡Sólo estás limitado por tu imaginación!

CAPÍTULO

1 Comenzando a conocer Greenfoot

temas:	la interfaz de Greenfoot, interaccionando con objetos, invocando métodos, ejecutando un escenario
conceptos:	objeto, clase, llamada a un método, parámetro, valor de vuelta

Este libro mostrará cómo desarrollar juegos y simulaciones con Greenfoot, un entorno de desarrollo software. En este capítulo, presentaremos el entorno Greenfoot, qué puede hacer y cómo se usa, empleando programas ya hechos.

Una vez que nos hayamos familiarizado con el uso de Greenfoot, empezaremos directamente a escribir juegos nosotros mismos.

El mejor modo para leer este capítulo (y de hecho todo el libro) es sentándote con tu ordenador abierto en tu mesa. Te pediremos frecuentemente que hagas cosas con Greenfoot mientras lees. Algunas de las tareas te las puedes saltar; sin embargo, tendrás que hacer algunas para poder progresar en el capítulo. En cualquier caso, aprenderás mejor si las realizas.

En este punto, asumimos que tienes ya instalado el software de Greenfoot así como los escenarios del libro (descritos en el Apéndice A). Si no, lee este apéndice primero [1].

1.1 Comenzando

Lanza Greenfoot y abre el escenario *leaves-and-wombats* del directorio *book-scenarios* de Greenfoot.

Nota Si es la primera vez que lanzas Greenfoot, verás un cuadro de diálogo que te pregunta qué quieres hacer. Selecciona ***Elegir un escenario***. En otro caso, usa ***Escenario-Abrir*** [2] del menú.

Comprueba que has abierto el escenario *leaves-and-wombats* que encuentras en el directorio *book-scenarios*, y no uno similar, escenario *wombats*, de la instalación estándar de Greenfoot.

Verás la interfaz de la ventana principal de Greenfoot, con el escenario abierto, de forma similar a la Figura 1.1. La ventana principal consta de tres áreas principales y un par de botones adicionales. Las áreas principales son:

---

[1] N. del T. En el libro se asume que usas la interfaz de usuario en español. Consulta el manual para ver cómo lanzarla en http://www.greenfoot.org/doc/translations.html.

[2] Usamos esta notación para decirte cómo seleccionar una función del menú. *Escenario-Abrir* te indica que selecciones el elemento *Abrir* del menú *Escenario*.

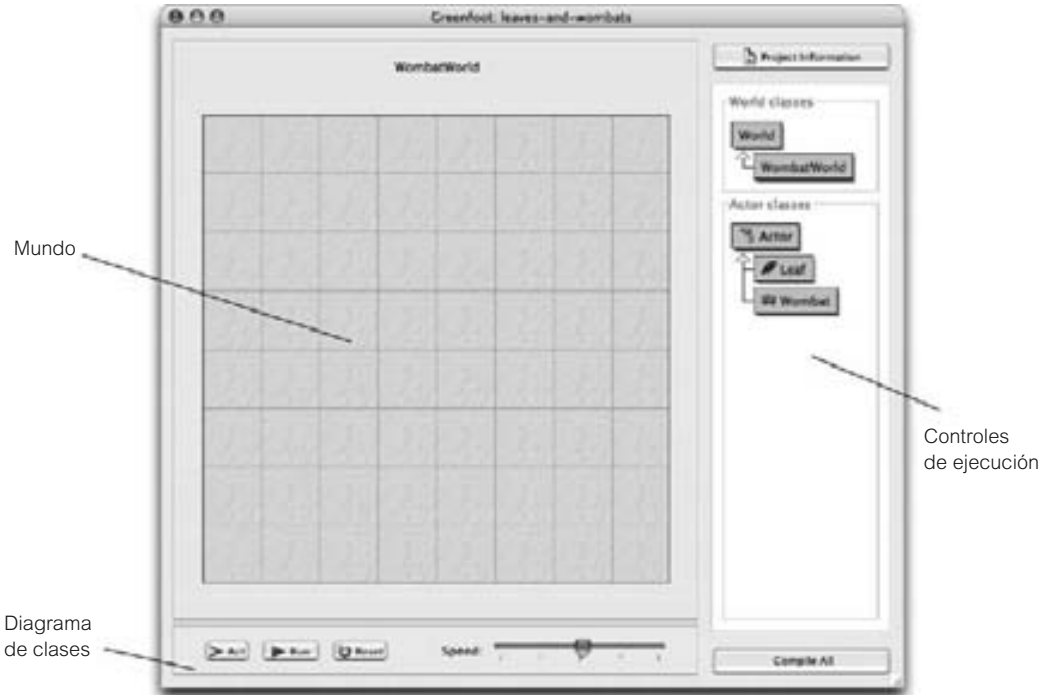


Figura 1.1
La ventana principal de Greenfoot

- El *mundo.* Se llama mundo al área más grande que ocupa la mayor parte de la pantalla (una rejilla de color tierra en este caso). Es donde el programa se ejecutará y verás lo que ocurre.
- El *diagrama de clases.* Se llama diagrama de clases al área de la derecha con cajas y flechas de color beis. Lo describiremos con más detalle en breve.
- Los *controles de ejecución.* Los controles de ejecución son los botones *Accionar*, *Ejecutar*, y *Reiniciar* y la barra inferior para ajustar la velocidad. Volveremos también más tarde con ellos.

1.2 Objetos y clases

Comenzaremos con el diagrama de clases. El diagrama de clases nos muestra las clases que participan en este escenario. En este caso, tenemos `World`, `WombatWorld`, `Actor`, `Leaf` y `Wombat`.

Emplearemos el lenguaje de programación Java en nuestros proyectos. Java es un lenguaje *orientado a objetos*. Los conceptos de clases y objetos son fundamentales en la orientación a objetos.

Comencemos observando la clase `Wombat`. La clase `Wombat` representa el concepto general de un wombátido —describe a todos los wombátidos. Una vez que tenemos una clase en Greenfoot, podemos crear *objetos* de esa clase. (Los objetos se llaman a menudo instancias o ejemplares en programación— estos términos son sinónimos.)

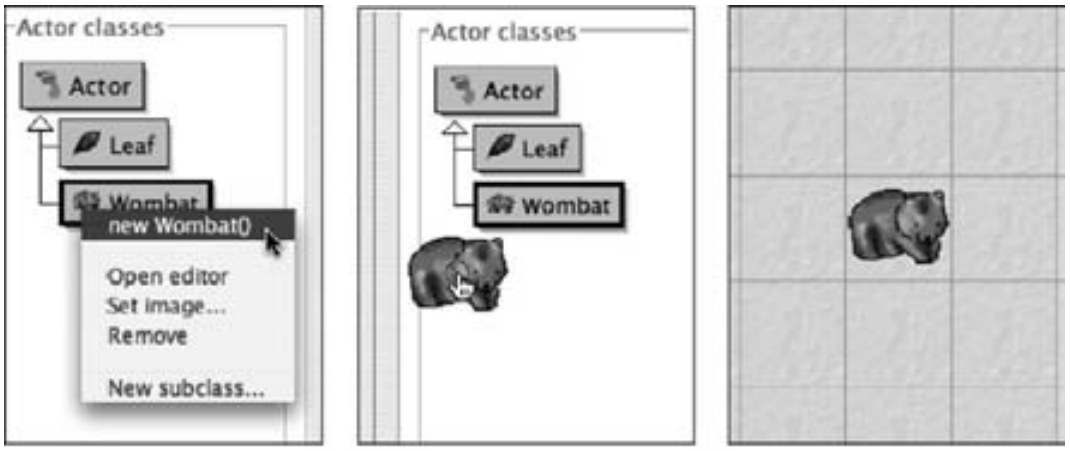
Un wombátido, por cierto, es un marsupial australiano (Figura 1.2). Si quieres saber más sobre ellos, busca en Internet— encontrarás muchos resultados.

Pincha con el botón derecho[3] en la clase `Wombat`, y verás una ventana emergente con el *menú de clase* (Figura 1.3a). Esta primera opción del menú, `new Wombat()`, nos permite crear nuevos objetos de la clase `Wombat`. Inténtalo.

[3] En Mac OS, pincha mientras mantienes pulsada la tecla Control, en vez de pinchar con el botón derecho si tienes un ratón de un botón.

Verás que obtienes una pequeña imagen de un objeto `Wombat`, que puedes mover por la pantalla con tu ratón (Figura 1.3b). Coloca el objeto *wombat* en el mundo simplemente pinchando en cualquier sitio del mundo (Figura 1.3c).

Figura 1.2
Un wombátido en el Parque Nacional de Narawntapu, Tasmania [4]



a) b) c)

Figura 1.3
a) El menú de clase
b) Arrastrando un nuevo objeto
c) Colocando el objeto

Concepto:

Se pueden crear muchos **objetos** de una **clase**.

Una vez que tienes una clase en Greenfoot, puedes crear tantos objetos como quieras.

Ejercicio 1.1 Crea algunos wombátidos más en el mundo. Crea algunas hojas (clase Leaf [5]).

Actualmente, sólo nos interesan las clases `Wombat` y `Leaf`. Trataremos el resto de clases más tarde.

1.3 Interaccionando con objetos

Una vez que has colocado varios objetos en el mundo, podemos interactuar con estos objetos pinchando con el botón derecho sobre ellos. Esto abrirá una ventana emergente con el *menú de los objetos* (Fi-

[4] Fuente de la imagen: Wikipedia, sujeta a la licencia GNU Free Documentation License.
[5] N. del T. Leaf es hoja en inglés.

Concepto:

Los objetos tienen **métodos**. Si invocas un método, se realiza una acción.

gura 1.4). El menú de los objetos nos muestra todas las operaciones que este objeto específico puede realizar. Por ejemplo, el menú de un objeto de la clase `Wombat`, nos muestra lo que el wombátido puede hacer (más dos funciones adicionales, *Inspeccionar* y *Eliminar*, que comentaremos más tarde).

Ejercicio 1.2 Invoca el método `move()` de un wombátido. ¿Qué hace? Pruébalo varias veces. Invoca el método `turnLeft()` (`girarIzquierda()`). Coloca enfrentados dos wombátidos en tu mundo.

En Java, estas operaciones se llaman *métodos.* Es bueno usar la terminología estándar, así que también les llamaremos métodos a partir de ahora. Podemos *invocar* un método seleccionándolo del menú.

Figura 1.4
El menú del objeto wombátido

En breve, podemos comenzar a hacer que pasen cosas si creamos objetos de una de las clases proporcionadas, y podemos dar órdenes a los objetos invocando sus métodos.

Observemos con más atención el objeto menú. Los métodos `move` y `turnLeft` se listan como:

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

Podemos ver que los nombres de los métodos no son la única cosa que se muestra. También aparece la palabra `void` al comienzo y un par de paréntesis al final. Estas dos piezas crípticas de información nos dicen qué datos recibe una llamada a un método y qué datos nos devuelve dicha llamada.

1.4 Tipos de vuelta

La palabra del principio se llama *tipo de vuelta (return type).* Nos dice lo que el método nos devolverá si lo invocamos. La palabra `void` significa «nada» en este contexto: los métodos con un tipo de vuelta `void` no devuelven ninguna información. Estos métodos simplemente ejecutan su acción, y luego paran.

Concepto:

El **tipo de vuelta** de un método especifica lo que devolverá una llamada a dicho método.

Cualquier otra palabra diferente de `void` nos dice que el método devuelve alguna información cuando es llamado, y de qué tipo es esa información. En el menú del wombátido (Figura 1.4) podemos ver también las palabras `int` y `boolean`. La palabra `int` es la abreviatura de «integer» (entero) y se refiere a todos los números enteros (números sin punto decimal). Algunos ejemplos de números enteros son 3, 42, -3, y 12000000.

El tipo `booleano` (boolean) tiene sólo dos valores posibles: `true` (cierto) y `false` (falso). Un método que devuelve un `boolean` nos devolverá o bien el valor `true` o bien el valor `false`.

Concepto:

Un método con tipo de vuelta **void** no devuelve ningún valor.

Los métodos con tipo de vuelta `void` son como órdenes. Si invocamos el método `turnLeft`, el wombátido obedece y gira a la izquierda. Los métodos con tipos de vuelta diferentes de `void` son como preguntas. Considera el método `canMove`:

```
boolean canMove()
```

Cuando invocamos este método, vemos un resultado similar al mostrado en la Figura 1.5, que muestra una ventana de diálogo. La información importante aquí es la palabra true, que es lo que el método nos ha devuelto.

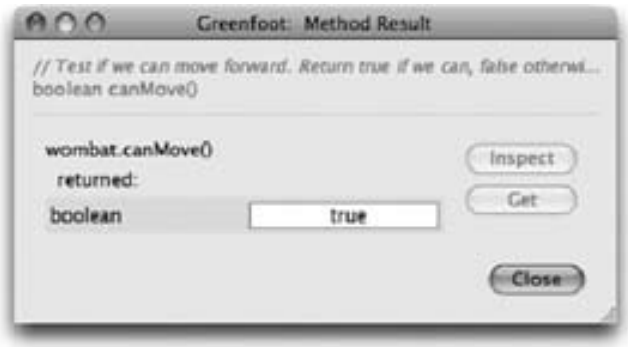


Figura 1.5
Un resultado de un método

En efecto, acabamos de preguntar al wombátido «¿Puedes moverte?» y el wombátido ha respondido «¡Sí!» (`true`).

Concepto:

Los métodos con tipos de vuelta void representan **órdenes**; los métodos con tipos de vuelta diferentes de void representan **preguntas**

Ejercicio 1.3 Invoca el método `canMove()` en tu wombátido. ¿Siempre devuelve *true*? ¿O puedes encontrar situaciones que devuelva *false*?

Prueba otro método que también devuelve un valor:

```
int getLeavesEaten()
```

Usando este método, podemos conseguir la información de cuántas hojas se ha comido el wombátido.

Ejercicio 1.4 Usando un objeto wombátido que acabes de crear, el método `getLeavesEaten()` siempre devolverá cero. ¿Puedes crear una situación en que el resultado de este método no sea cero? (En otras palabras, ¿puedes hacer que tu wombátido coma algunas hojas?)

Los métodos con tipos de vuelta que no son `void` normalmente nos dicen algo sobre el objeto (*¿Puede moverse? ¿Cuántas hojas ha comido?*), pero no cambian el objeto. El wombátido está igual que antes de que le preguntáramos sobre las hojas. Los métodos con tipos de vuelta `void` son normalmente órdenes a los objetos para que hagan algo.

1.5 Parámetros

La otra parte que aún no hemos visto del menú *método* son los paréntesis tras el nombre del método.

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

Concepto:

Un **parámetro** es un mecanismo para pasar datos adicionales a un método.

Los paréntesis después del nombre del método contienen la *lista de parámetros* (o *argumentos*). Esto nos dice que el método requiere alguna información adicional para poder ejecutarse, y si la requiere, qué tipo de información.

Si sólo vemos un par de paréntesis sin nada más dentro (tal como hemos visto en todos los métodos hasta ahora), entonces el método tiene una *lista vacía de parámetros.* En otras palabras, no espera ningún parámetro —cuando invoquemos el método, simplemente se ejecutará. Si no hay nada dentro de los paréntesis, entonces el método espera uno o más parámetros —se debe proporcionar información adicional.

Concepto:

Los parámetros y los valores de vuelta tienen **tipos**. Algunos ejemplos de tipos son int para números, y boolean para valores cierto/falso.

Probemos el método `setDirection`[2]. Podemos ver que tiene las palabras `int direction` escritas en su lista de parámetros. Cuando lo invocamos, vemos una ventana de diálogo similar a la mostrada en la Figura 1.6.

Las palabras `int direction` nos indican que este método espera un parámetro de tipo `int`, que especifica una *dirección.* Un parámetro es una información adicional que debemos proporcionar a un método para que se ejecute. Cada parámetro se define mediante dos palabras: primero, el tipo del parámetro (aquí: `int`) y luego el nombre, que nos da una idea de para qué se usa este parámetro. Si un método tiene un parámetro, necesitamos proporcionar esta información adicional cuando invoquemos el método.

Figura 1.6
Una ventana de llamada a un método

En este caso, el tipo `int` nos dice que deberíamos proporcionar un número entero, y el nombre sugiere que este número especifica de alguna forma la dirección para girar.

En la parte superior de la ventana hay un comentario que nos da un poco más de información: el parámetro `direction` debería valer entre 0 y 3.

[2] N. del T. Direction es dirección en inglés. El método setDirection se traduce por establece dirección.

Ejercicio 1.5 Invoca el método `setDirection(int direction)`. Proporciona un valor al parámetro y observa qué ocurre. ¿Qué número corresponde a cada dirección? Escríbelo. ¿Qué ocurre cuando escribes un número mayor de 3? ¿Qué ocurre si introduces algo que no sea un número entero, como por ejemplo un número (2.5) o una palabra (tres)?

Concepto:

La especificación de un método, que muestra su tipo de vuelta, nombre y parámetros se denomina **signatura**.

El método `setDirection` espera sólo un parámetro. Después, veremos casos de métodos que esperan más de un parámetro. En ese caso, el método listará todos los parámetros que espera dentro de los paréntesis.

La descripción de cada método se muestra en el menú del objeto, incluyendo el tipo de vuelta, el nombre del método, y la lista de parámetros, que se denomina *signatura del método.*

Hemos alcanzado ahora un punto donde puedes realizar las principales interacciones con los objetos de Greenfoot. Puedes crear objetos de clases, interpretar las signaturas de los métodos, e invocar los métodos (con y sin parámetros).

1.6 Ejecución con Greenfoot

Hay otra forma de interaccionar con los objetos de Greenfoot: los controles de ejecución.

Consejo:

Puedes colocar los objetos en el mundo más rápidamente si seleccionas una clase en el diagrama de clases, y entonces pinchas en el mundo mientras pulsas la tecla de mayúsculas.

Ejercicio 1.6 Coloca un wombátido y un buen número de hojas en el mundo, e invoca entonces varias veces el método `act`[6]`()` del wombátido. ¿Qué hace el método? ¿En qué se diferencia del método `move`? Asegúrate de comprobar situaciones diferentes, por ejemplo con un wombátido mirando en el borde del mundo, o sentado sobre una hoja

Ejercicio 1.7 Aún con un wombátido y algunas hojas en el mundo pincha en el botón *Accionar* situado en los controles de ejecución cerca de la parte inferior de la ventana de Greenfoot. ¿Qué hace?

Ejercicio 1.8 ¿Cuál es la diferencia entre pinchar en el botón *Actuar* e invocar el método `act()`? (Intenta con varios wombátidos en el mundo.)

Ejercicio 1.9 Pincha en el botón *Ejecutar*. ¿Qué hace?

Concepto:

Los objetos que se pueden colocar en el mundo se llaman **actores**.

El método `act` es un método fundamental de los objetos Greenfoot. Lo encontramos frecuentemente en todos los capítulos restantes. Todos los objetos en un mundo Greenfoot tienen este método `act`. Invocar al método `act` es como si le ordenáramos al objeto «Haz lo que quieres hacer ahora». Si lo has probado en nuestro wombátido, verás que el método `act` hace algo así:

- Si estoy sentado en una hoja, me la como.
- Si no, si puedo ir hacia delante, me muevo hacia delante.
- En el resto de casos, giro a la izquierda.

Los experimentos de los ejercicios anteriores deberían haberte mostrado que el botón *Actuar* de los controles de ejecución simplemente llama al método `act` de los actores en el mundo. La única dife-

[6] N. del T. Act significa actuar o accionar.

rencia con invocar el método a través del menú del objeto es que el botón *Actuar* llama al método `act` de todos los objetos del mundo, mientras que el menú de objeto sólo afecta al objeto escogido.

El botón *Ejecutar* simplemente llama al método `act` una y otra vez hasta que pinchas en *Pausa*. Vamos a probar lo que hemos visto hasta ahora en el contexto de otro escenario.

1.7 Un segundo ejemplo

Abre otro escenario, denominado *asteroids1*, de la carpeta *chapter01* enlos escenarios del libro. Debe ser como el de la Figura 1.7 (excepto que no verás aún el cohete o los asteroides en tu pantalla).

1.8 Comprendiendo el diagrama de clases

Concepto:

Una **subclase** es una clase que representa una especialización de otra. En Greenfoot, esto se denota con una flecha en el diagrama de clases.

Observemos primero con mayor detalle el diagrama de clases (Figura 1.8). En la parte superior, verás dos clases llamadas `World` (mundo) y `Space` (espacio), conectadas por una flecha.

La clase `World` está siempre en todos los escenarios de Greenfoot —forma parte de Greenfoot. La clase debajo de ella, `Space` en este caso, representa el mundo específico de este escenario concreto (el espacio). Su nombre puede ser diferente en cada escenario, pero todos los escenarios siempre tendrán un mundo específico aquí.

La flecha indica una relación *es-un*: `Space` *es un* `World` (en el sentido de mundos de Greenfoot: `Space`, aquí, es un mundo específico de Greenfoot). También a veces decimos que `Space` es una *subclase* de `World`.

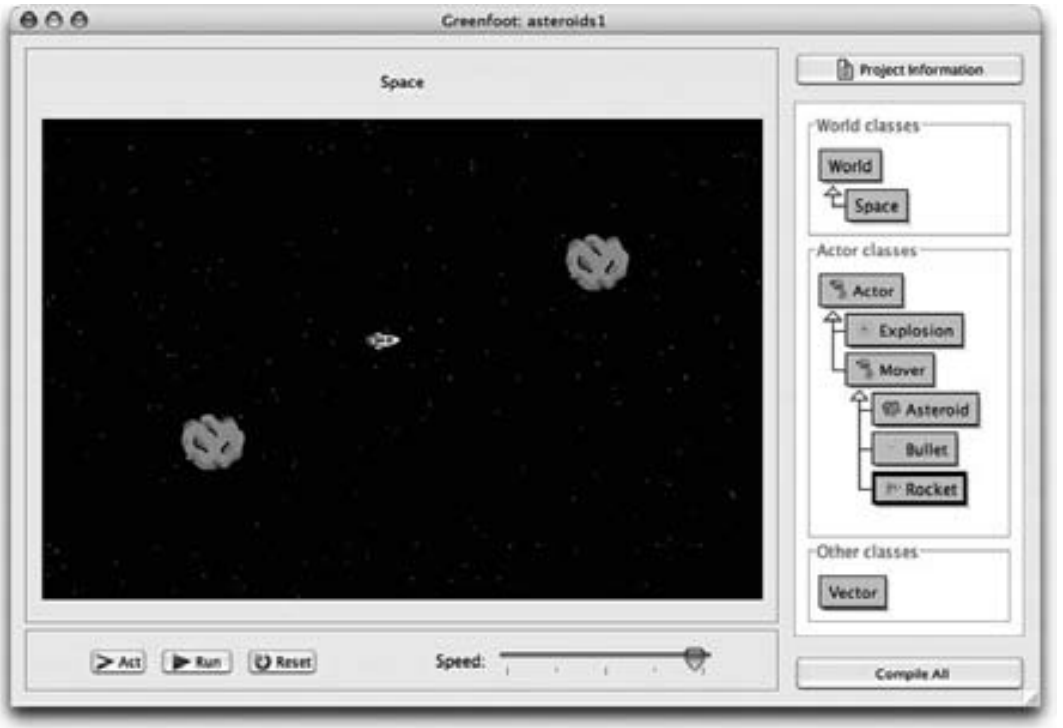


Figura 1.7
El escenario asteroids1

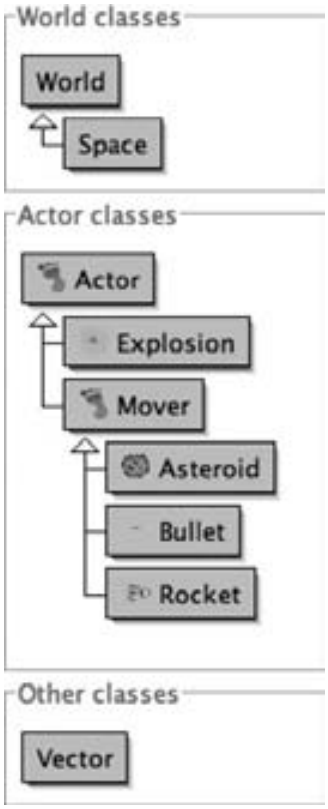


Figura 1.8
Un diagrama de clases

Normalmente no tenemos que crear objetos de las clases mundo —Greenfoot lo hace por nosotros. Cuando abrimos un escenario, Greenfoot crea automáticamente un objeto de la subclase mundo. El objeto se muestra entonces en la parte principal de la pantalla. (La imagen negra grande del espacio es un objeto de la clase `Space`.)

Debajo de esto, vemos otro grupo de seis clases, enlazadas por flechas. Cada clase representa a sus propios objetos. Listando desde abajo, vemos que tenemos cohetes (*rockets*), balas (*bullets*), y asteroides (*asteroids*), y que todos son «movibles» (*movers*), mientras que las explosiones (*explosions*) y los «movibles» son actores.

De nuevo, tenemos definidas relaciones de subclase: `Rocket`, por ejemplo, es una subclase de `Mover`, y tanto `Mover` como `Explosion` son subclases de `Actor`. (Recíprocamente, decimos que `Mover` es una *superclase* de `Rocket` y que `Actor` es una *superclase* de `Explosion`.)

La relación de subclase puede recorrer varios niveles: `Rocket`, por ejemplo, es también una subclase de `Actor` (porque es una subclase de `Mover`, la cual es una subclase de `Actor`). Trataremos más tarde más detalles del significado de subclases y superclases.

La clase `Vector`, mostrada en la parte inferior del diagrama bajo el encabezado *Otras clases,* es una clase de apoyo o auxiliar usada por otras clases. No podemos poner objetos de ella en el mundo.

1.9 Jugando con asteroides

Podemos comenzar a jugar en este escenario creando algunos objetos actores (objetos de subclases de `Actor`) y colocarlos en el mundo. En este caso, sólo creamos objetos de las clases que no tienen subclases: `Rocket`, `Bullet`, `Asteroid` y `Explosion`.

Comencemos colocando un cohete (rocket) y dos asteroides en el espacio. (Recuerda: puedes crear los objetos pinchando con el botón derecho en la clase, o seleccionando la clase y pinchando mientras pulsas mayúsculas.)

Cuando hayas colocado los objetos, pincha en el botón *Ejecutar*. Ya puedes controlar la nave espacial con las flechas del teclado, y puedes disparar usando la barra de espacio. Intenta evitar los asteroides antes de chocar con ellos.

Ejercicio 1.10 Si has jugado al juego durante un rato, habrás notado que no puedes disparar muy rápido. Vamos a ajustar nuestro software de disparo de la nave para que podamos disparar un poco más rápido. (¡Esto debe facilitar que nos libremos de los asteroides más fácilmente!). Coloca un cohete en el mundo e invoca su método `setGunReloadTime` (a través del menú de objeto), y fija el tiempo de recarga del arma (gun reload time) a 5. Juega de nuevo (con al menos dos asteroides) y pruébalo.

Ejercicio 1.11 Una vez que has conseguido eliminar todos los asteroides (o en otro punto del juego), para la ejecución (pulsa *Pausa*) y averigua cuántos disparos has disparado. Puedes saberlo usando un método del menú de objeto cohete. (Intenta destruir dos asteroides con el menor número de balas posible.)

Ejercicio 1.12 Habrás notado que el cohete se mueve un poco en cuanto lo colocas en el mundo. ¿Cuál es su velocidad inicial?

Ejercicio 1.13 Los asteroides tienen una *estabilidad* intrínseca. Cada vez que disparamos, su estabilidad decrece. Cuando llega a cero, explotan. ¿Cuál es su estabilidad inicial después de crearlos? ¿En cuánto decrece la estabilidad si reciben un disparo?

(Pista: Dispara a un asteroide sólo una vez, y mira cuánto vale la estabilidad antes y después del disparo. Otra pista: para disparar a un asteroide, debes ejecutar el juego. Para usar el menú de un objeto, debes pausar antes el juego.)

Ejercicio 1.14 Crea un asteroide muy grande.

1.10 Código fuente

Concepto:

En cada clase se define en su **código fuente**. Este código define qué pueden hacer los objetos de esta clase. Podemos ver el código fuente abriendo el editor de clases.

La conducta de cada objeto se define en su clase. Para especificar esta conducta, debemos escribir *código fuente* en el lenguaje de programación Java. El código fuente de una clase especifica todos los detalles sobre la clase y sus objetos. Para ver el código fuente de una clase, selecciona la opción *Abrir el editor* del menú de la clase, que te mostrará la ventana de edición (Figura 1.9) con el código fuente de la clase.

El código fuente de esta clase es relativamente complejo, y no necesitamos entenderlo por ahora. Sin embargo, si estudias el resto de este libro y programas tus propios juegos o simulaciones, aprenderás cómo programarlo.

En este momento, sólo es importante que entiendas que podemos cambiar la conducta de los objetos cambiando el código fuente de las clases. Vamos a intentarlo.

Hemos visto antes que la velocidad de disparo del cohete por defecto era bastante lenta. Podríamos cambiarla individualmente para cada cohete, invocando un método en cada nuevo cohete, pero tendríamos que repetir esta operación una y otra vez, cada vez que empezamos a jugar. En vez de esto,

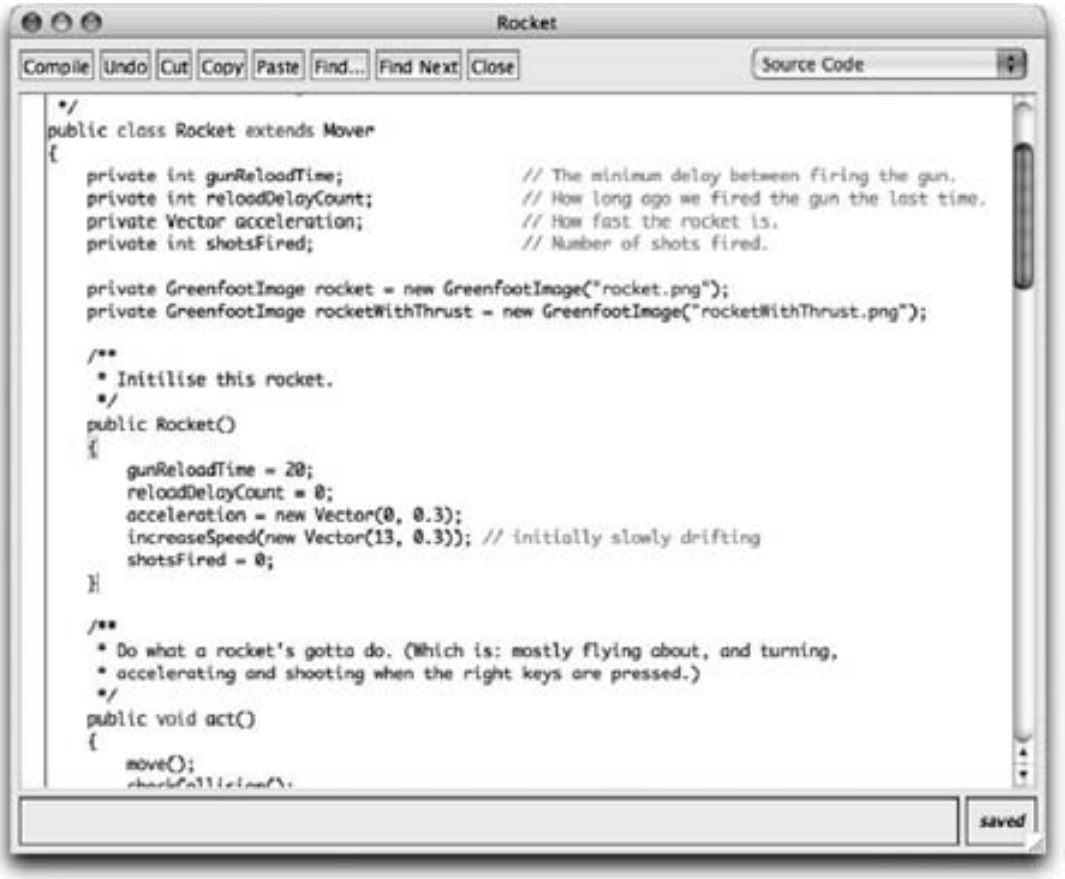


Figura 1.9
La ventana de edición de la clase `Rocket`

Pista:

Puedes editar una clase pinchando dos veces en la clase en el diagrama de clases.

podemos cambiar el código del cohete para que cambie su velocidad inicial (digamos a 5), de forma que todos los cohetes comiencen a partir de ahora con una conducta mejor.

Abre el editor de la clase `Rocket`. Si bajas unas 25 líneas, deberás encontrar una línea que pone

```
gunReloadTime = 20;
```

Aquí es donde se fija el tiempo inicial de recarga del arma. Cambia esta línea para que ponga

```
gunReloadTime = 5;
```

No cambies nada más. Verás muy pronto que hay que ser muy meticuloso al programar sistemas. Un simple carácter erróneo o que falte un carácter producirá errores. Por ejemplo, si borras el punto y coma del final de la línea, se produciría un error más adelante.

Concepto:

Los ordenadores no entienden código fuente. Debe transformarse antes en código máquina para que se pueda ejecutar. Este proceso se denomina **compilación**.

Cierra la ventana del editor (hemos terminado el cambio) y vuelve a mirar el diagrama de clases. Ahora ha cambiado: varias clases aparecen con rayas (Figura 1.10). Las rayas indican que una clase ha sido editada y debe ser *compilada.* La compilación es un proceso de traducción: el código fuente de la clase se traduce en un código máquina que tu ordenador puede ejecutar.

Las clases deben compilarse siempre después de que el código fuente haya cambiado, antes de que nuevos objetos de esa clase puedan ser creados. (Habrás notado que varias clases necesitan ser recompiladas aunque sólo hemos cambiado una clase. Esto sucede a menudo porque unas clases dependen de otras. Cuando una cambia, varias necesitan ser traducidas de nuevo.)

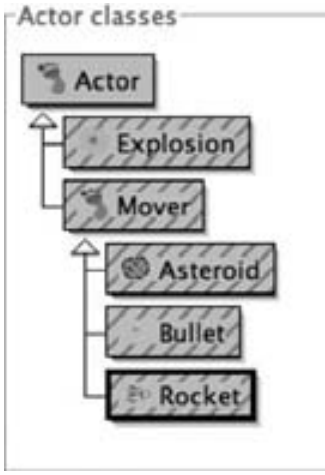


Figura 1.10
Las clases después de editar

Podemos compilar las clases pinchando en el botón *Compilar Todo* en la esquina inferior derecha de la venta principal de Greenfoot. Una vez que las clases han sido compiladas, las rayas desaparecen, y podemos crear los objetos otra vez.

Ejercicio 1.15 Haz los cambios descritos anteriormente en la clase `Rocket`. Cierra el editor y compila las clases. Pruébalo: los cohetes deberían ahora ser capaces de disparar más rápido desde el principio.

Volveremos al juego de los asteroides en el Capítulo 7, donde veremos cómo programar este juego.

1.11 Resumen

En este capítulo, hemos visto cómo son los escenarios de Greenfoot y cómo interactuar con ellos. Hemos visto cómo crear objetos y cómo comunicarnos con estos objetos invocando sus métodos. Algunos métodos son órdenes al objeto, mientras que otros objetos devuelven información sobre el objeto. Los parámetros son usados para proporcionar información adicional a los métodos, mientras que los tipos de vuelta devuelven información al llamante.

Los objetos se crean desde sus clases, y el código fuente controla la definición de la clase (y con esto, la conducta y características de todos los objetos de la clase).

Hemos visto que podemos cambiar el código fuente usando un editor. Después de editar el código fuente, las clases necesitan ser recompiladas.

Pasaremos la mayor parte del resto del libro aprendiendo cómo escribir código fuente en Java para crear escenarios que hagan cosas interesantes.

Resumen de conceptos

- Los escenarios de Greenfoot consisten en un conjunto de **clases**.
- Se pueden crear muchos **objetos** de una misma clase.

- Los objetos tienen **métodos**. Al invocarlos, realizan una acción.
- El **tipo de vuelta** de un método especifica lo que la llamada a un método devolverá.
- Un método con un tipo de vuelta **void** no devuelve ningún valor.
- Los métodos con tipos de vuelta void representan **órdenes**; los métodos con tipos de vuelta diferentes de void representan **preguntas**.
- Un **parámetro** es un mecanismo para pasar datos adicionales a un método.
- Los parámetros y los tipos de vuelta tienen **tipos**. Algunos ejemplos de tipos son **int** para números y **boolean** para valores cierto/falso.
- La especificación de un método, que muestra su tipo de vuelta, nombre, y parámetros se llama **signatura**.
- Los objetos que pueden ser colocados en el mundo se conoce como **actores**.
- Una subclase es una clase que representa una especialización de otra. En Greenfoot, esto se muestra con una flecha en el diagrama de clases.
- Cada clase se define por su **código fuente**. Este código define lo que los objetos de esta clase pueden hacer. Podemos ver el código fuente abriendo el editor de la clase.
- Los ordenadores no entienden código fuente. Necesita ser traducido a código máquina antes de ejecutarlo. Esto se denomina **compilación**.

Términos en inglés

Inglés	Español
object	objeto
class	clase
class diagram	diagrama de clases
compilation	compilación
compile	compile
helper class	clase de apoyo
invoke a method	invocar un método
instance	instancia, ejemplar
method	método
method call	llamada a un método
parameter	parámetro, argumento
signature	signatura
source code	código fuente
subclass	subclase
return type	tipo de vuelta

CAPÍTULO

2 El primer programa: el cangrejito

temas:	programación: movimiento, girar, reaccionar a las esquinas de la pantalla
conceptos:	código fuente, llamada a método, parámetro, secuencia, sentencia if

En el capitulo anterior, presentamos cómo usar los escenarios que vienen con Greenfoot. Hemos creado objetos, invocado métodos, y jugado un juego.

Ahora, comenzaremos a hacer nuestro propio juego.

2.1 El escenario del cangrejito

El escenarios que usaremos en este capítulo se llama *little-crab.* Encontrarás este escenario en los proyectos de este libro.

El escenario debe ser similar al mostrado en la Figura 2.1.

Ejercicio 2.1 Lanza Greenfoot y abre el escenario *little-crab*. Coloca un cangrejo en el mundo y ejecuta el programa (pincha en el botón *Ejecutar*). ¿Qué observas? (Recuerda: si los iconos de las clases aparecen rayados, debes compilar el proyecto antes.)

A la derecha, ves las clases del escenario (Figura 2.2). Fíjate que tenemos como siempre la clase `Actor` de Greenfot, una clase llamada `Animal`, y la clase `Crab` (cangrejo).

La jerarquía (denotada por flechas) indica una relación *es-un* (también denominada de *herencia*): un cangrejo *es un* animal, y un animal *es un* actor. (Se deduce entonces, que un cangrejo es también un actor.)

Inicialmente, trabajaremos sólo con la clase `Crab`. Hablaremos más adelante sobre las clases `Actor` y `Animal`.

Si has hecho el ejercicio anterior, sabrás que la respuesta a la pregunta «¿Qué observas?» es «nada».

El cangrejo no hace nada cuando ejecutas Greenfoot. Esto se debe a que la clase `Crab` tiene el código fuente sin rellenar, y no se especifica, por tanto, qué debe hacer el cangrejo.

En este capítulo, trabajaremos para cambiar esto. La primera cosa que haremos será conseguir que el cangrejo se mueva.

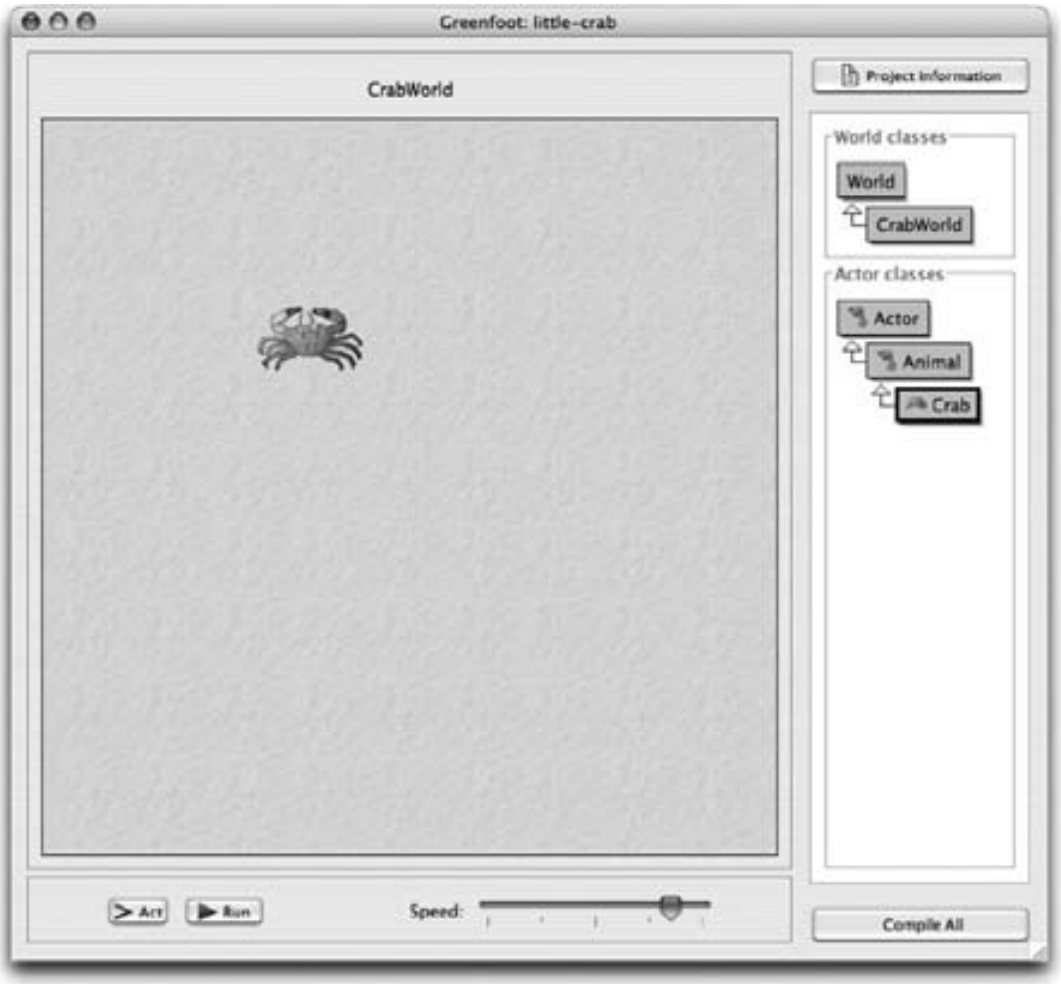


Figura 2.1
El escenario *Cangrejito*

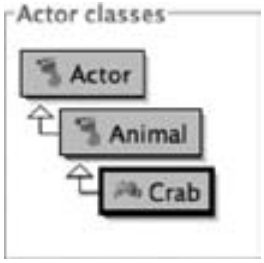


Figura 2.2
Las clases actor del *Cangrejito*

2.2 Haciendo que el cangrejo se mueva

Echemos un vistazo al código fuente de la clase `Crab`. Abre el editor y muestra el código fuente de `Crab`. (Puedes hacer esto seleccionando la opción *Abrir Editor* del menú emergente de la clase, o simplemente puedes pinchar dos veces en la clase.)

El código fuente que ves se muestra en el cuadro de Código 2.1.

Código 2.1
La versión inicial de la clase **Crab**

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

/**
 *   Esta clase define un cangrejo. Los cangrejos viven en la playa.
 */
public class Crab extends Animal
{
    public void act()
    {
        //Añade aquí tu código para hacer acciones.
    }
}
```

Ésta es una definición estándar de una clase Java. Esto es, este texto define lo que el cangrejo puede hacer. Lo veremos con más detalle un poco más adelante. Por ahora, nos concentraremos en conseguir que el cangrejo se mueva.

Dentro de esta definición de clase, podemos ver el denominado *método act.* Sigue este esquema:

```
public void act()
{
    // Añade tu código de acción aquí.
}
```

La primera línea es la *signatura* del método. Las tres últimas líneas —las dos llaves y nada entre ellas— es denominado el *cuerpo* del método. Aquí, podemos añadir el código que determine las acciones del cangrejo. Podemos reemplazar el texto gris de en medio con una orden. Una posible orden es:

```
move();
```

Fíjate que hay que escribirlo exactamente como está puesto, incluyendo los paréntesis y el punto y coma. El método `act` debería estar ahora así:

```
public void act()
{
    move();
}
```

Ejercicio 2.2 Cambia el método `act` en tu clase cangrejo para incluir la instrucción `move()` (muévete) como se ha mostrado antes. Compila el escenario (pincha en el botón *Compila Todo*) y coloca un cangrejo en el mundo. Prueba a pinchar los botones *Actuar* y *Ejecutar*.

Ejercicio 2.3 Coloca varios cangrejos en el mundo. Ejecuta el escenario. ¿Qué observas?

Verás que el cangrejo se puede mover ahora por la pantalla. La instrucción `move()` hace que el cangrejo se mueva un poco a la derecha. Cuando pinchamos en el botón *Actuar* en la ventana principal de Greenfoot, el método `act` se ejecuta una vez. Esto es, la instrucción que hemos escrito dentro del método `act (move())` se ejecuta.

Pinchar en el botón *Ejecutar* es como pinchar en el botón *Accionar* varias veces, muy rápidamente. Por tanto, el método `act` se ejecuta una y otra vez hasta que pinchamos en *Pausa*.

Terminología

La instrucción `move()` se denomina una **llamada a un método**. Un **método** es una acción que un objeto sabe cómo realizar (aquí el objeto es el cangrejo) y una **llamada a un método** es una instrucción que le dice al cangrejo lo que debe hacer. Los paréntesis son parte de la llamada a un método. Las instrucciones terminan con un punto y coma.

2.3 Girando

Veamos qué otras instrucciones podemos usar. El cangrejo también entiende la instrucción `turn` (*gira*). A continuación se muestra un ejemplo de uso:

```
turn(5);
```

El número 5 en la instrucción especifica cuántos grados debe girar el cangrejo. Se denomina *parámetro.* También podemos usar otros números, por ejemplo

```
turn(23);
```

Concepto:

Podemos pasar información adicional a algunos métodos entre los paréntesis. Los valores que pasamos se denominan **parámetros**.

El número de grados es 360, por lo que podemos dar un valor entre 0 y 359. (Girar 360 grados significa dar una vuelta entera, al igual que girar 0 grados significa no girar nada.)

Si quisiéramos girar en vez de movernos, podemos reemplazar la instrucción `move()` con una instrucción `turn(5)`. El método `act` será como sigue:

```
public  void act()
{
    turn(5);
}
```

Ejercicio 2.4 Reemplaza `move()` con `turn(5)` en tu escenario. Pruébalo. Prueba también con valores diferentes de 5, y mira qué pasa. Recuerda: cada vez que cambias el código, debes volver a compilar.

Ejercicio 2.5 ¿Cómo puedes hacer que el cangrejo gire a la izquierda?

Concepto:

Cuando hay varias instrucciones, se ejecutan **en secuencia**, una tras otra, en el orden en que han sido escritas.

La siguiente cosa que vamos a intentar es que se mueva y gire. El método `act` puede contener más de una instrucción —podemos simplemente escribir varias instrucciones seguidas.

El Código 2.2 muestra la clase completa `Crab`, como debe estar si queremos que se mueva y gire a continuación. En este caso, en cada paso, el cangrejo se moverá y girará después (pero estas acciones sucederán tan rápidamente una tras otra que parece que suceden a la vez).

Código 2.2
Haciendo que el cangrejo se mueva y gire

```
import greenfoot.*; // (World, Actor, GreenfootImage, y Greenfoot)
/**
 *   Esta clase define un cangrejo. Los cangrejos viven en la playa.
 */
public class Crab extends Animal
{
   public void act()
   {
       move();
       turn(5);
   }
}
```

Ejercicio 2.6 Usa las instrucciones `move()` y `turn(N)` en el método `act` de tu cangrejo. Prueba con varios valores de **N**.

Terminología

El número que va entre paréntesis en la instrucción `turn` —esto es, el 5 en `turn(5)`— se denomina **parámetro**. Un parámetro es una información adicional que tenemos que proporcionar cuando llamamos a algunos métodos.

Algunos métodos, como `move`, no esperan ningún parámetro. Pueden ejecutarse sin problemas cuando escribimos la instrucción `move()`. Otros métodos, como `turn`, necesitan más información: ¿Cuánto debería girar? En este caso, tenemos que proporcionar esta información mediante el valor de un parámetro entre los paréntesis, por ejemplo, `turn(17)`.

Nota al margen: Errores

Concepto:

Cuando se compila una clase, el compilador comprueba si hay algún error. Si encuentra alguno, muestra un **mensaje de error**.

Cuando escribimos código fuente, tenemos que ser muy cuidadosos —cada carácter cuenta—. Si tenemos un pequeño fallo, nuestro programa no funcionará. Normalmente, no compilará.

Esto nos pasará a menudo. Cuando escribimos programas, inevitablemente cometemos errores, y tenemos que corregirlos. Probemos esto ahora.

Si, por ejemplo, olvidamos escribir un punto y coma después de la instrucción `move()`, nos advertirá cuando intentemos compilarlo.

Ejercicio 2.7 Abre tu editor para mostrar el código fuente del cangrejo, y borra el punto y coma después de `move()`. Compila a continuación. Puedes probar también con otros errores, como una errata al escribir `move` o hacer algún cambio aleatorio al código. Comprueba bien que deshaces todos estos cambios después de terminar el ejercicio.

Ejercicio 2.8 Haz cambios diferentes para causar mensajes de error diferentes. Encuentra al menos cinco mensajes de error. Escribe cada mensaje de error y qué cambio produjo el error.

Pista:

Cuando un mensaje de error aparece en la parte inferior de la ventana de edición, un botón con una *interrogación* aparece a la derecha. Si pinchas en el botón, muestra información adicional del mensaje de error.

Como podemos ver en este ejercicio, si cometemos un pequeño error, Greenfoot abrirá el editor, destacará la línea, y mostrará un mensaje en la parte inferior de la ventana del editor. Este mensaje intenta explicar el error. Los mensajes, sin embargo, varían considerablemente en su precisión y utilidad. Algunas veces nos dicen con mucha precisión cuál es el problema, pero otras veces son algo crípticos y difíciles de entender. La línea que se resalta es a menudo la línea donde hay un problema, pero otras veces es una línea posterior al problema. Cuando ves, por ejemplo, un mensaje del tipo "`; expected`", es posible que lo que pase sea que falta el punto y coma de la línea anterior a la línea destacada.

Aprenderás a entender estos mensajes con el tiempo. Por ahora, si obtienes un mensaje, y no entiendes muy bien qué significa, revisa cuidadosamente tu código, y comprueba que has escrito todo correctamente.

2.4 Tratando los bordes de la pantalla

Cuando hicimos que los cangrejos se movieran y giraran en las secciones previas, se quedaban bloqueados cuando alcanzaban el borde de la pantalla. (Greenfoot está diseñado para que los actores no puedan salir del mundo y caerse en el borde.)

Ahora vamos a mejorar esta conducta de forma que el cangrejo detecte que ha alcanzado el final del mundo, y se gire. La pregunta es, ¿cómo hacemos esto?

Concepto:

Una subclase **hereda** todos los métodos de su superclase. Esto significa que tiene y puede usar todos los métodos que su superclase defina.

Anteriormente hemos usado los métodos `move` y `turn`, así que puede que haya otro método que nos ayude con este nuevo objetivo. (De hecho, lo hay.) Pero, ¿cómo averiguamos qué métodos hay disponibles?

Los métodos `move` y `turn` que hemos usado hasta ahora vienen de la clase `Animal`. Un cangrejo es un animal (esto está indicado por la flecha que va de `Crab` a `Animal` en el diagrama de clases), por tanto, puede hacer todo lo que los animales pueden hacer. Nuestra clase `Animal` sabe cómo moverse y girar —esta es la razón por la que el cangrejo también sabe cómo hacerlo—. Esto se denomina *herencia*: la clase `Crab` hereda todas las habilidades (métodos) de la clase `Animal`.

La pregunta ahora es, ¿qué más cosas pueden hacer los animales?

Para investigar esto, podemos abrir el editor de la clase `Animal`. El editor puede mostrar dos vistas diferentes: puede mostrar el código fuente (como hemos visto para la clase `Crab`) o puede mostrar la documentación. La vista puede ser cambiada seleccionando un menú de selección en la esquina superior derecha de la ventana del editor. Ahora queremos ver la clase `Animal` en la *Vista de documentación* (Figura 2.3).

Ejercicio 2.9 Abre el editor de la clase `Animal`. Cambia a la vista de documentación. Busca la lista de métodos de esta clase (el «resumen de métodos»). ¿Cuántos métodos tiene esta clase?

Si miramos el resumen de métodos, podemos ver todos los métodos que proporciona la clase `Animal`. Hay tres métodos que nos son especialmente interesantes en este momento:

```
boolean atWorldEdge()
```
Comprueba si estamos cerca de uno de los bordes del mundo.

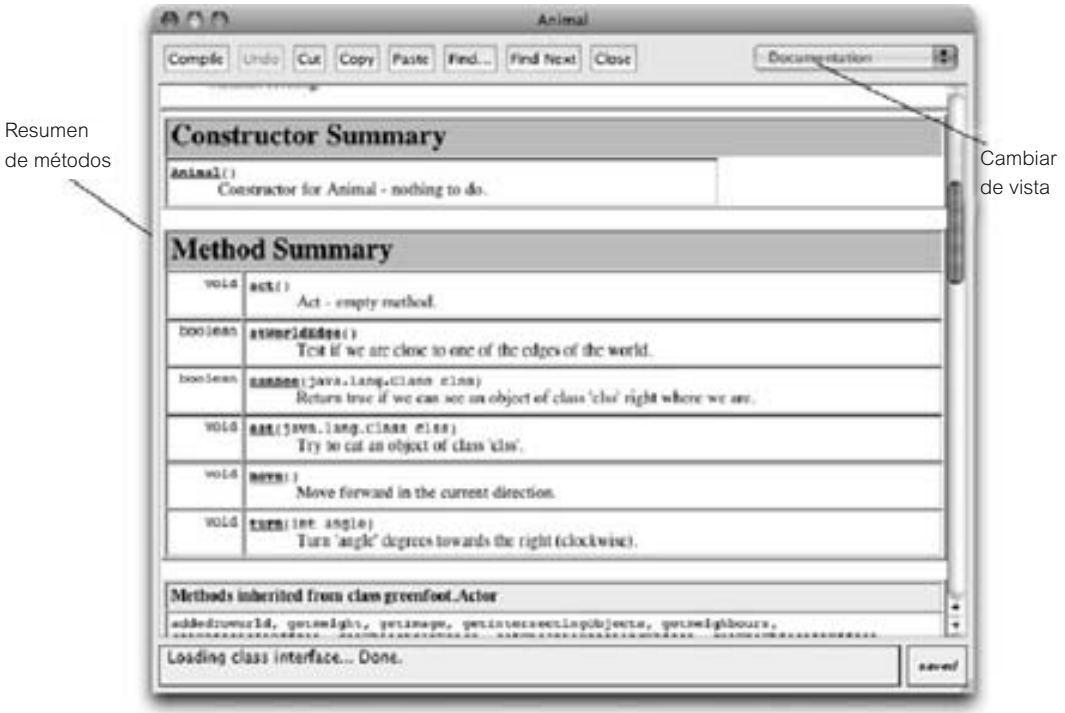


Figura 2.3
Vista de documentación (con resumen de métodos) de la clase `Animal`.

`void move()`
Moverse hacia delante en la dirección actual.

`void turn(int angulos)`
Girar los grados «angulos» hacia la derecha (sentido horario).

Aquí podemos ver las signaturas de los tres métodos, que ya vimos en el Capítulo 1. La signatura de cada método comienza con el tipo de vuelta, seguido por el nombre del método y la lista de parámetros. Antes de nombres de los tres métodos son `atWorldEdge`, `move`, y `turn`.

Los métodos `move` y `turn` son los que usamos en secciones previas. Si miramos a sus listas de parámetros, podemos ver lo que observamos antes: `move` no tiene parámetros (los paréntesis están vacíos), y `turn` espera un parámetro de `tipo int` (un número entero) para el ángulo. (Lee la sección 1.5 de nuevo si no recuerdas las listas de parámetros.)

Podemos ver también que los métodos `move` y `turn` tienen `void` como tipo de vuelta. Esto significa que no devuelven ningún valor. Estamos ordenando al objeto que se mueva o gire. El animal sólo tiene que obedecer la orden, pero no nos tiene que dar ninguna respuesta.

Concepto:

Cuando llamamos a un método con **tipo de vuelta void** realiza una orden. Sin embargo, cuando llamamos a un método con **tipo de vuelta que no es void** realiza una pregunta.

La signatura de `atWorldEdge` es un poco diferente. Es

```
boolean atWorldEdge()
```

Este método no tiene parámetros (no hay nada entre los paréntesis), pero especifica un tipo de vuelta: `boolean`. Nos hemos encontrado antes con el tipo `boolean type` en la sección 1.4 —es un tipo que puede tener dos valores: *true* o *false.*

Cuando llamamos a métodos que tienen valores de vuelta (en el caso de que el tipo de vuelta no sea `void`) no es como realizar una orden, sino como hacer una pregunta. Si usamos el método `atWorldEdge()`,

el método deberá responder o bien `true` (¡Sí!) o `false` (¡No!). Por tanto, podemos usar este método para comprobar si estamos en un borde del mundo.

Ejercicio 2.10 Crea un cangrejo. Pincha con el botón derecho en él, y busca el método `boolean atWorldEdge()` (está en el submenú *heredado de Animal*, dado que el cangrejo hereda este método de la clase `Animal`). Invoca este método. ¿Qué devuelve?

Ejercicio 2.11 Deja que el cangrejo corra hasta el borde de la pantalla (o muévelo allí manualmente), y llama entonces otra vez al método `atWorldEdge()`. ¿Qué devuelve ahora?

Podemos ahora combinar este método con una *sentencia-if* para escribir el código mostrado en Código 2.3.

Código 2.3
Girando alrededor del borde del mundo

```
import greenfoot.*;  // (World, Actor,  GreenfootImage, y  Greenfoot)
/**
 *   Esta clase define un cangrejo. Los cangrejos viven en la playa.
 */
public class  Crab extends Animal
{
   public void act()
   {
       if ( atWorldEdge() )
       {
           turn(17);
       }
       move();
   }
}
```

Concepto:

Una **sentencia if** puede ser usada para escribir instrucciones que son ejecutadas sólo si se cumple una condición.

La sentencia `if` es una parte del lenguaje Java que permite ejecutar órdenes sólo si se cumple una condición. Por ejemplo, aquí queremos girar sólo si estamos cerca del borde del mundo. El código que tenemos que escribir es:

```
if ( atWorldEdge() )
{
   turn(17);
}
move();
```

La forma general de una sentencia if es ésta:

```
if ( condición )
{
   instrucción;
   instrucción;
...
}
```

Donde pone *condición* se puede poner cualquier expresión que sea cierta o falsa (tales como nuestra llamada al método `atWorldEdge()`), y las *instrucciones* se ejecutarán sólo si la condición es cierta. Puede haber una o más instrucciones.

Si la condición es falsa, se salta las instrucciones, y la ejecución continúa después de la llave que cierra la sentencia if.

Observa que nuestra llamada al método `move()` está fuera de la sentencia if, por lo que será ejecutado siempre. En otras palabras, si estamos en el borde del mundo, giramos y nos movemos a continuación; si no estamos en el borde del mundo, sólo nos movemos.

Pista:

En el editor de Greenfoot, cuando colocas el cursor después de una llave que se abre o se cierra, Greenfoot muestra la llave correspondiente que estamos cerrando o abriendo. Esto se puede usar para comprobar si tus llaves están correctas, o falta o sobra alguna.

Ejercicio 2.12 ¡Pruébalo! Escribe el código discutido anteriormente y observa si tus cangrejos giran en el borde de la pantalla. Presta atención a los símbolos de abrir y cerrar llave —es fácil olvidar uno o poner demasiados.

Ejercicio 2.13 Experimenta con varios valores para el parámetro del método `turn`. Encuentra uno que te parezca adecuado.

Ejercicio 2.14 Coloca la sentencia `move()` dentro de la sentencia `if`, en vez de después de ella. Averigua cuál es el efecto y explica la conducta que observas. (Luego, arréglalo volviéndolo a dejar como estaba.)

Nota: Sangrado

En todos los ejemplos de código que hemos visto hasta ahora (por ejemplo, Código 2.3), puedes haber notado que hemos usado un sangrado cuidadoso. Cada vez que se abre una llave, las líneas siguientes están sangrada un nivel más que las anteriores. Cuando se cierra una llave, el sangrado retrocede un nivel, de forma que la llave que cierra está justo debajo de la llave correspondiente que abre. Esto facilita encontrar qué llave estamos cerrando.

Usamos cuatro espacios por nivel de sangrado. La tecla Tab inserta espacios en tu editor para un nivel de sangrado.

Es muy importante tener cuidado con el sangrado en tu código. Si no sangras con cuidado, algunos errores (en especial, que falten o sobren llaves) son difíciles de localizar. Un buen sangrado hace el código más legible, y evita por tanto errores potenciales.

2.5 Resumen de técnicas de programación

En este libro, estamos aprendiendo a programar con un enfoque guiado por ejemplos. Introducimos las técnicas generales de programación a medida que las necesitamos para mejorar nuestros escenarios. Resumiremos las técnicas de programación más importantes a final de cada capítulo, para que quede claro lo que necesitas saber del desarrollo de los ejercicios para progresar adecuadamente.

En este capítulo, hemos visto cómo invocar métodos (tales como `move()`), con y sin parámetros. Esto será la base de la programación en Java. También hemos aprendido cómo identificar el cuerpo del método `act` —aquí es donde comenzamos a escribir instrucciones.

Te has encontrado algunos mensajes de error. Esto será algo continuo durante todas tus tareas de programación. Todos cometemos errores y todos encontramos mensajes de error. Esto no es un signo de ser un programador malo —es una parte normal de la programación

Hemos comenzado a ver qué es la herencia: las clases heredan los métodos de sus superclases. La vista de documentación de una clase nos da un resumen de los métodos disponibles.

Y, muy importante, hemos visto cómo tomar decisiones. Hemos usado la sentencia if para la ejecución condicional. Esto ha ido de la mano con la aparición del tipo `boolean`, cuyo valor puede ser *true* o *false.*

Resumen de conceptos

- Una **llamada a un método** es una instrucción que le dice a un objeto que realice una acción. La acción se define por el método del objeto.
- Se puede pasar información adicional a algunos métodos entre los paréntesis. Los valores pasados se denominan **parámetros**.
- Se pueden ejecutar múltiples instrucciones **en secuencia**, una tras otra, en el orden en que son escritas.
- Cuando una clase se compila, el compilador comprueba si hay errores. Si encuentra un error, muestra un **mensaje de error**.
- Una subclase **hereda** todos los métodos de su superclase. Esto significa que tiene, y puede usar, todos los métodos que su superclase defina.
- Si llamamos a un método con un **tipo de vuelta void**, estamos ordenando algo. Si llamamos a un método con un **tipo de vuelta distinto de void**, estamos preguntando algo.
- Una **sentencia if** puede ser usada para escribir instrucciones que son ejecutadas sólo cuando se cumple una condición.

Términos en inglés

Inglés	Español
body	cuerpo
indentation	sangrado
inheritance	herencia
method	método
method call	llamada a método
run	ejecutar
statement	sentencia

CAPÍTULO

3

Mejorando el cangrejo –programación más sofisticada

temas:	conducta aleatoria, control del teclado, sonido
conceptos:	notación punto, números aleatorios, comentarios

En el capítulo anterior, vimos lo básico para comenzar a programar nuestro primer juego. Hubo muchas cosas nuevas que tuvimos que ver. Ahora, añadiremos conductas más interesantes. Nos resultará más fácil añadir código a partir de ahora, ya que hemos visto muchos de los conceptos fundamentales.

La primera cosa que miraremos es cómo añadir conducta aleatoria.

3.1 Añadiendo conducta aleatoria

En nuestra implementación, el cangrejo se puede mover por la pantalla, y puede girar en el borde de nuestro mundo. Pero cuando anda, anda siempre hacia delante en línea recta. Esto es lo que queremos cambiar ahora. Los cangrejos no andan siempre en línea recta, vamos a añadir un poco de conducta aleatoria: el cangrejo debería ir más o menos en línea recta, pero debería girar un poco alrededor de esa línea.

Podemos conseguir esto en Greenfoot usando números aleatorios. El entorno de Greenfoot tienen un método que nos da un número aleatorio. Este método, denominado `getRandomNumber` (*obténNúmeroAleatorio*), espera un parámetro que especifica el límite del número. Devolverá un número entre 0 (cero) y ese límite. Por ejemplo,

```
Greenfoot.getRandomNumber(20)
```

Concepto:

Cuando queremos llamar a un método que no es de nuestra clase ni heredado, tenemos que especificar la clase u objeto que define el método antes del nombre del método, seguido de un punto. Esto se conoce como la **notación punto**.

nos dará un número aleatorio entre 0 y 20. El límite —20— se excluye, por lo que el número estará en el rango 0-19.

La notación usada aquí se denomina *notación punto.* Cuando llamamos a métodos que han sido definidos en nuestra clase o heredados, basta con escribir el nombre del método y la lista de parámetros. Cuando el método se ha definido en otra clase, es necesario especificar la clase u objeto que tiene el método, seguido de un punto, y seguido del nombre del método y la lista de parámetros. Como el método `getRandomNumber` no es de las clases `Crab` ni `Animal`, sino que se define en una clase denominada `Greenfoot`, tenemos que escribir "`Greenfoot`." delante de la llamada al método.

Supongamos que queremos programar nuestro cangrejo de forma que haya un 10 por ciento de probabilidades de que en cada paso el cangrejo gire un poco de su curso. Podemos programar el esquema de esto con una sentencia if:

Concepto:

Los métodos que pertenecen a clases (en vez de a objetos) se indican con la palabra clave **static** en su signatura. Se denominan **métodos de clase**.

```
if ( algo-es-cierto)
{
   turn(5);
}
```

Nota: Métodos estáticos

Los métodos pueden pertenecer a objetos o clases. Cuando un método pertenece a una clase, escribimos

```
nombre-clase.nombre-método (parámetros);
```

para llamar al método. Cuando un método pertenece a un objeto, escribimos

```
objeto.nombre-método (parámetros);
```

para llamarlo.

Ambos tipos de métodos se definen en una clase. La signatura de un método nos dice si un método dado pertenece a los objetos de esa clase, o a la clase misma.

Los métodos que pertenecen a la clase misma se indican con la palabra clave **static** al comienzo de la signatura del método. Por ejemplo, la signatura del método `getRandomNumber` de la clase Greenfoot es

```
static int getRandomNumber(int limit);
```

Esto nos dice que debemos escribir el nombre de la clase (en este caso, Greenfoot) antes del punto en la llamada al método.

Encontraremos llamadas a métodos que pertenecen a otros objetos en un capítulo posterior.

Ahora tenemos que encontrar una expresión en lugar de *algo-es-cierto* que devuelva exactamente el 10 por ciento de los casos.

Podemos hacer esto usando un número aleatorio (usando el método `Greenfoot.getRandomNumber`) y un operador menor-que. El operador menor-que compara dos números y devuelve true si el primero es menor que el segundo. «Menor que» se escribe usando el símbolo «<». Por ejemplo:

```
2 < 33
```

es cierto, mientras que

```
162 < 42
```

es falso.

Ejercicio 3.1 Antes de seguir leyendo, intenta escribir, en papel, una expresión usando el método `getRandomNumber` y el operador menor-que, de forma que cuando se ejecute, devuelva el 10 por ciento de las veces.

Ejercicio 3.2 Escribe otra expresión que sea cierta el 7 por ciento de las veces.

Nota

Java tiene varios operadores para comparar dos valores:

<	menor que	>=	mayor o igual que
>	mayor que	= =	igual que
<=	menor o igual que	!=	no igual a

Si queremos expresar una oportunidad en porcentajes, lo más fácil es escoger números aleatorios con límite 100. Una expresión que es cierta el 10 por ciento de las veces, por ejemplo, podría ser

```
Greenfoot.getRandomNumber(100) < 10
```

Como la llamada a `Greenfoot.getRandomNumber(100)` nos da un nuevo número aleatorio entre 0 y 99 cada vez que lo llamamos, y como estos números están uniformemente distribuidos, será menor que 10 en el 10 por ciento de todos los casos.

Podemos usar esto ahora para hacer que nuestro cangrejo gire un poco en el 10 por ciento de sus pasos (Código 3.1).

Código 3.1
Cambios aleatorios en el trayecto —primer intento

```
import greenfoot.*; // (World, Actor, GreenfootImage, y Greenfoot)
/**
 *   Esta clase define un cangrejo. Los cangrejos viven en la playa.
 */
public class Crab extends Animal
{
    public void act()
    {
        if ( atWorldEdge() )
        {
            turn(17);
        }
        if ( Greenfoot.getRandomNumber(100) < 10  )
        {
            turn(5);
        }
        move();
    }
}
```

Ejercicio 3.3 Prueba los cambios en el trayecto indicados arriba en tu versión. Experimenta con diferentes probabilidades para girar.

Éste es un buen comienzo, pero no va bien todavía. En primer lugar, si el cangrejo gira, siempre gira la misma cantidad (5 grados), y, en segundo lugar, siempre gira a la derecha, nunca a la izquierda. Lo que realmente nos gustaría ver es que girara una cantidad pequeña pero aleatoria a la izquierda o a la derecha. (Veremos esto ahora, si te sientes capaz, intenta implementar esto tú primero antes de seguir leyendo.)

La forma más fácil de resolver el primer problema —girar siempre con la misma cantidad, en nuestro caso, 5 grados— es reemplazar el número 5 fijo de nuestro código con uno aleatorio, como sigue:

```
if ( Greenfoot.getRandomNumber(100) < 10  )
{
    turn( Greenfoot.getRandomNumber(45) );
}
```

En este ejemplo, el cangrejo aún giraría en el 10 por ciento de sus pasos. Y cuando gire, girará una cantidad aleatoria, entre 0 y 44 grados.

Ejercicio 3.4 Prueba el código mostrado arriba. ¿Qué observas? ¿Gira en diferentes ángulos ahora el cangrejo cuando gira?

Ejercicio 3.5 Tenemos aún el problema de que el cangrejo sólo gira a la derecha. Ésta no es la conducta normal de un cangrejo, así que vamos a intentar arreglarlo. Modifica tu código de forma que el cangrejo gire a la izquierda o a la derecha hasta 45 grados cuando gira.

Ejercicio 3.6 Prueba a ejecutar tu escenario con muchos cangrejos en el mundo. ¿Giran todos a la vez o de forma independiente? ¿Por qué?

El proyecto *little-crab-2* (incluido en los escenarios del libro) muestra una implementación de lo que hemos hecho hasta ahora, incluyendo los últimos ejercicios.

3.2 Añadiendo gusanos

Vamos a hacer nuestro mundo un poco más interesante añadiendo un nuevo tipo de animal.

A los cangrejos les gusta comer gusanos. (Bueno, esto no es cierto para todos los tipos de cangrejos del mundo real, pero a algunos les gusta. Supongamos que nuestro cangrejo es uno de los que les gusta comer gusanos.) Por tanto, añadamos ahora una clase para los gusanos.

Podemos añadir nuevas clases actor en un escenario Greenfoot seleccionando *Nueva subclase* de una de las clases actor existentes (Figure 3.1). En este caso, nuestra nueva subclase `Worm` (*gusano*) es un tipo específico de animal. Por tanto, debería ser una subclase de la clase `Animal`. (Recuerda, ser una subclase es una relación *es-un*: Un gusano *es un* animal.)

Cuando creamos una nueva subclase, se nos pide que introduzcamos el nombre de la clase, y que seleccionemos una imagen (Figura 3.2).

En nuestro caso, llamamos a la clase «Worm». Por convención, los nombres de la clase en Java deben siempre comenzar con una letra mayúscula. El nombre debe describir qué tipo de objeto representa la clase, por tanto, el nombre «Worm» es el adecuado para nuestro propósito.

A continuación, debemos asignar una imagen a la clase. Hay algunas imágenes asociadas con el escenario, y una librería[1] entera de imágenes genéricas entre las que escoger. En este caso hemos preparado una imagen de un gusano y está disponible en las imágenes del escenario, por lo que puede bastar con que selecciones la imagen llamada *worm.png*.

[1] N. del T. Se traduce *library* por librería, y no por *biblioteca* que sería su traducción correcta, dado que se ha popularizado esta traducción en informática y es la que proporciona la interfaz de Greenfoot.

Figura 3.1
Creando nuevas subclases

Figura 3.2
Creando una nueva clase

Una vez hecho, podemos pinchar en *Ok.* La clase se ha añadido ahora a nuestro escenario, y podemos compilarla y añadir luego gusanos a nuestro mundo.

Ejercicio 3.7 Añade algunos gusanos a nuestro mundo. Añade también algunos cangrejos. Ejecuta el escenario. ¿Qué observas? ¿Qué hacen los gusanos? ¿Qué ocurre si un cangrejo encuentra un gusano?

Ahora sabemos cómo añadir nuevas clases a nuestro escenario. La siguiente tarea consiste en conseguir que las clases interactúen: cuando un cangrejo encuentra un gusano, debería comérselo.

3.3 Comiendo gusanos

Queremos ahora añadir una nueva conducta al cangrejo. Cuando el cangrejo se encuentra un gusano, se lo come. De nuevo, tenemos que comprobar primero qué métodos hemos heredado de la clase `Animal`. Cuando abrimos de nuevo el editor de la clase `Animal`, y cambiamos a la vista de *documentación*, podemos ver los dos métodos siguientes:

`boolean canSee (java.lang.Class clss)`
Devuelve true si podemos ver un objeto de la clase 'clss' justo desde donde estamos. Se traduce como puede`Ver()`.

`void eat (java.lang.Class clss)`
Intenta comer un objeto de la clase 'clss'. Se traduce como `come()`

Usando estos métodos, vamos a implementar esta conducta. El primer método comprueba si el cangrejo puede comerse un gusano. (Sólo puede verlo cuando está justo delante de él —nuestros animales tienen un alcance de visión muy corto.) Este método devuelve un `booleano` —*true* o *false*, así que podemos usarlo en una sentencia if.

El segundo método se come un gusano. Ambos métodos esperan un parámetro de tipo `java.lang.Class`. Esto significa que esperamos que se especifique una de las clases de nuestro escenario. Aquí se muestra un ejemplo de código:

```
if ( canSee(Worm.class) )
{
    eat(Worm.class);
}
```

En este caso, especificamos `Worm.class` como el parámetro para ambas llamadas a métodos (el método `canSee` y el método `eat`). Esto declara qué tipo de objeto estamos buscando, y qué tipo de objeto queremos comer. Nuestro método `act` completo en este momento se muestra en el cuadro de Código 3.2.

Prueba esto ahora. Coloca varios gusanos en el mundo (recuerda, puedes utilizar el atajo de pinchar mientras mantienes la tecla mayúsculas para colocar rápidamente varios actores), coloca unos pocos cangrejos, ejecuta el escenario, y mira qué pasa.

Código 3.2
Primera versión de cómo comer un gusano

```
public void act()
{
    if ( atWorldEdge() )
    {
        turn(17);
    }
    if ( Greenfoot.getRandomNumber(100) < 10  )
    {
        turn(Greenfoot.getRandomNumber(90)-45);
    }
    move();

    if ( canSee(Worm.class) )
    {
        eat(Worm.class);
    }
}
```

Nota avanzada – Paquetes (Packages)

(Las notas etiquetadas como «avanzadas» se insertan para proporcionar información más detallada a los lectores realmente interesados en los detalles. No son cruciales para entender el resto del texto, y pueden ser saltadas sin problemas.)

En la definición de los métodos `canSee` y `eat` hemos visto un tipo de parámetro con el nombre `java.lang.Class`. ¿Qué es esto?

Muchos tipos se definen con clases. Muchas de estas clases están en la librería estándar de clases Java. Puedes ver la documentación de la librería de clases Java seleccionando *Documentación de Librerías* Java en el menú *Ayuda* de Greenfoot.

La librería de clases Java contiene miles de clases. Para que sea un poco más fácil trabajar con ellas, han sido agrupadas en paquetes (grupos de clases relacionadas lógicamente). Cuando el nombre de una clase incluye puntos, tal como en `java.lang.Class`, el nombre de la clase es sólo la última parte, y las partes anteriores son el nombre del paquete. Por tanto, estamos mirando en la clase llamada «Class» del paquete «java.lang».

Intenta encontrar la documentación de esta clase en la documentación de la librería de Java.

3.4 Creando nuevos métodos

En las secciones anteriores, hemos añadido una nueva conducta al cangrejo —girar en el borde del mundo, girar de forma aleatoria, y comer gusanos. Si continuamos haciendo esto como lo hemos hecho hasta ahora, el método `act` se haría cada vez más largo y, en algún caso, difícil de entender. Podemos mejorar esto dividiéndolo en piezas más pequeñas.

Podemos crear nuestros propios métodos adicionales en la clase `Crab` para nuestros propios propósitos. Por ejemplo, en vez de escribir simplemente algún código para buscar un gusano y comérnoslo en el método `act`, podemos añadir un nuevo método con este fin. Para hacer esto, primero tenemos que decidir un nombre para este método. Digamos que lo llamamos `lookForWorm` (buscaGusano). A continuación podemos crear un nuevo método añadiendo el siguiente código:

Concepto:

Una **definición de un método** define una nueva acción para los objetos de esta clase. La acción no se ejecuta de forma inmediata, sino que el método puede ser llamado mediante una llamada a este método más tarde para ejecutarlo.

```
/**
 *  Comprueba si descubrimos un gusano.
 *  Si lo vemos, nos lo comemos. Si no, no hacemos nada.
 */
public void lookForWorm()
{
    if ( canSee(Worm.class) )
    {
        eat(Worm.class);
    }
}
```

Las primeras cuatro líneas son un *comentario.* Un comentario es ignorado por el ordenador —está escrito para los lectores humanos. Usamos comentarios para explicar a los lectores humanos el propósito de este método.

Concepto:

Los **comentarios** se escriben en el código fuente como explicación para los lectores humanos. Son ignorados por el ordenador.

Cuando definimos este método, el código no se ejecuta de forma inmediata. De hecho, sólo con definirlo no se ejecuta en absoluto. Sólo hemos definido una nueva acción posible («buscar un gusano») que puede ser ejecutada más tarde. Sólo será ejecutada cuando este método sea llamado. Podemos añadir una llamada a este método dentro del método `act`:

```
lookForWorm();
```

Observa que la llamada tiene los paréntesis para la lista (vacía) de parámetros. El código fuente completo tras esta reestructuración se muestra en el Código 3.3.

Código 3.3
Dividiendo el código en varios métodos

```
public void act()
{
    if ( atWorldEdge() )
    {
        turn(17);
    }
    if ( Greenfoot.getRandomNumber(100) < 10  )
    {
        turn(5);
    }
    move();
    lookForWorm();
}

/**
 *  Comprueba si hemos encontrado un gusano.
 *  Si lo hemos visto, nos lo comemos. Si no, no hacemos nada.
 */
public void lookForWorm()
{
    if ( canSee(Worm.class) )
    {
```

```
            eat(Worm.class);
        }
    }
```

Observa que cambiando así el código no hemos cambiado en absoluto la conducta del cangrejo. Simplemente hemos hecho que el código sea más fácil de leer. Cuando añadimos más código a una clase, los métodos tienden a ser cada vez más largos. Cuanto más largos son los métodos, más difíciles son de entender. Al separar nuestro código en métodos más cortos, conseguimos que el código sea más fácil de entender.

Ejercicio 3.8 Crea otro método nuevo llamado `randomTurn` (giraAletoriamente(), este método no tiene parámetros y no devuelve nada). Selecciona el código que realiza el giro aleatorio, y muévelo del método `act` al método `randomTurn`. A continuación, llama a este nuevo método `randomTurn` desde tu método `act`. No olvides escribir un comentario para este método.

Ejercicio 3.9 Crea otro método llamado `turnAtEdge` (giraEnElBorde(), tampoco tiene parámetros ni devuelve nada). Mueve el código que comprueba si estamos al final del mundo (y que realiza el giro si estamos) al método `turnAtEdge`. Llama al método `turnAtEdge` desde tu método `act`. Tu método `act` debería ser como la versión mostrada en el Código 3.4.

Código 3.4
El nuevo método act después de crear métodos para las subtareas

```
public void act()
{
    turnAtEdge();
    randomTurn();
    move();
    lookForWorm();
}
```

Por convención, los nombres de los métodos en Java siempre comienzan con una letra minúscula. Los nombres de los métodos no pueden tener espacios (ni muchos otros caracteres de puntuación). Si el nombre de un método está formado por varias palabras, usa mayúsculas en medio del nombre del método para indicar el comienzo de cada palabra.

3.5 Añadiendo una langosta

Estamos ahora en un punto en que tenemos un cangrejo que anda más o menos de forma aleatoria en nuestro mundo, y come gusanos si los encuentra.

Para hacerlo un poco más interesante, vamos a añadir una nueva criatura: una langosta (*lobster*). A las langostas les gusta perseguir a los cangrejos.

Ejercicio 3.10 Añade una nueva clase Langosta a tu escenario. La clase debe ser una subclase de `Animal`, llamarse `Lobster` (con una «L» mayúscula), y debe usar la imagen preparada *lobster.png*.

Ejercicio 3.11 ¿Qué esperas que hagan las langostas cuando las pones en el mundo justo tras añadir la clase? Compila el escenario y pruébalo.

Queremos ahora programar nuestras langostas para que coman cangrejos. Esto es bastante fácil, ya que esta conducta es muy similar a la de los cangrejos. La única diferencia es que las langostas buscan cangrejos, mientras que los cangrejos buscan gusanos.

Ejercicio 3.12 Copia entero el método `act` desde la clase `Crab` a la clase `Lobster`. También copia los métodos enteros `lookForWorm`, `turnAtEdge`, y `randomTurn`.

Ejercicio 3.13 Cambia el código de la clase `Lobster` para que busque cangrejos, en vez de gusanos. Puedes hacer esto cambiando cada ocurrencia de «Worm» en el código fuente por «Crab». Por ejemplo, donde pone `Worm.class`, cámbialo a `Crab.class`. También cambia el nombre `lookForWorm` a `lookForCrab`. No olvides actualizar tus comentarios.

Ejercicio 3.14 Coloca un cangrejo, tres langostas y muchos gusanos en el mundo. Ejecuta el escenario. ¿Consigue el cangrejo comerse todos los gusanos antes de ser comido por la langosta?

Deberías tener ahora una versión de tu escenario donde los cangrejos y langostas andan de forma aleatoria, y buscan gusanos y cangrejos respectivamente.

Ahora, hagamos que este programa sea un juego.

3.6 Control del teclado

Para conseguir un comportamiento de juego, necesitamos que pueda haber un jugador. El jugador (¡tú!) debería ser capaz de controlar al cangrejo con el teclado, mientras que las langostas van a continuar ejecutándose ellas mismas como hasta ahora.

El entorno de Greenfoot tiene un método que nos permite comprobar si una tecla del teclado ha sido pulsada. Se llama `isKeyDown`, y, como el método `getRandomNumber` que vimos en la sección 3.1, es un método de la clase `Greenfoot`. Su signatura es

```
static boolean isKeyDown(String key)
```

Vemos que este método es estático (es un método de clase) y que el tipo de vuelta es `boolean`. Esto significa que el método devuelve *true* o *false*, y que puede ser usado como una condición en una sentencia if.

Vemos también que el método espera un parámetro de tipo `String`. Un `String` es una cadena de texto (tales como una palabra o una frase), escrito entre comillas. A continuación mostramos algunos ejemplos de `Strings`:

```
“Esto es un String”
“nombre”
“Un”
```

En este caso, el `String` que esperamos es el nombre de una tecla que queramos comprobar. Cada tecla del teclado tiene un nombre. Para esas teclas que producen caracteres visibles, el carácter es su nombre. Por ejemplo, la tecla-A se llama «A». Hay otras teclas que también tienen nombre. Por ejemplo, la flecha izquierda se llama «left». Por tanto, si queremos comprobar si se pulsa la flecha izquierda, podemos escribir

```
if (Greenfoot.isKeyDown("left"))
{
    ...// haz algo
}
```

Observa que necesitamos escribir «`Greenfoot.`» antes de la llamada al método `isKeyDown`, dado que este método se define en la clase `Greenfoot`.

Pista:

Greenfoot guarda automáticamente las clases y escenarios cuando cerramos sus ventanas. Para conseguir una copia interna de las etapas de los escenarios, usa la opción **Guardar una copia como** del menú Escenario.

Si, por ejemplo, quisiéramos que nuestro cangrejo girara 4 grados cada vez que se pulse la tecla cursor izquierdo, podemos escribir

```
if (Greenfoot.isKeyDown("left"))
{
    turn(-4);
}
```

La idea ahora es eliminar el código del cangrejo que realiza los giros aleatorios y también el código que gira automáticamente en el borde del mundo, y reemplazarlos con el código que nos permita controlar los giros del cangrejo desde el teclado.

Ejercicio 3.15 Borra la conducta de giro aleatorio del cangrejo.

Ejercicio 3.16 Borra el código del cangrejo que realiza el giro en el borde del mundo.

Ejercicio 3.17 Añade el código en el método `act` del cangrejo que hace que el cangrejo gire a la izquierda cuando pulses la tecla cursor izquierdo. Pruébalo.

Ejercicio 3.18 Añade otras líneas de código similares en el método `act` que hagan que el cangrejo gire a la derecha cuando se pulse la tecla cursor derecho.

Ejercicio 3.19 Si no lo has hecho en primer lugar, comprueba que el código que comprueba las teclas pulsadas y realiza el giro no está escrito directamente en el método `act`, sino en un método separado, que puede tener por nombre `checkKeypress` (compruebaTeclaPulsada()). Este método debería ser llamado desde el método `act`.

Intenta resolver estas tareas tú mismo primero. Si te bloqueas, mira en la página siguiente. El cuadro del Código 3.5 te muestra los métodos completos del cangrejo `act` y `checkKeypress` tras este cambio. La solución también está disponible en los escenarios del libro, en *little-crab-3*. Esta versión incluye todos los cambios vistos hasta ahora.

¡Estás ya preparado para probar tu juego! Coloca un cangrejo, varios gusanos, y unas pocas langostas en el mundo, y mira si puedes comerte todos los gusanos antes de que te coman las langostas. (Obviamente, cuantas más langostas pongas, más difícil será...)

Código 3.5
El método "act" del cangrejo. Controlando el cangrejo desde el teclado

```
/**
 *  Act – haz lo que quiera que el cangrejo quiere hacer
 */
public void act()
{
    checkKeypress();
    move();
    lookForWorm();
}
/**
 *  Comprueba si se pulsa una tecla de control en el teclado.
 *  Si se pulsa, reacciona como corresponda.
 */
public void checkKeypress()
{
    if (Greenfoot.isKeyDown("left"))
    {
        turn(-4);
    }
    if (Greenfoot.isKeyDown("right"))
    {
        turn(4);
    }
}
```

3.7 Terminando el juego

Podemos hacer una simple mejora para que termine la ejecución del juego cuando el cangrejo sea cazado por una langosta. Greenfoot tiene un método para hacer esto —simplemente necesitamos encontrar cómo se llama.

Concepto:

La **documentación de la API** lista todas las clases y métodos disponibles en Greenfoot. A menudo tenemos que buscar los métodos aquí.

Para encontrar los métodos disponibles de Greenfoot podemos mirar la documentación de las clases de Greenfoot.

En Greenfoot, escoge *Documentación de Clases Greenfoot* del menú *Ayuda*. Te mostrará la documentación de todas las clases de Greenfoot en un navegador web (Figura 3.3).

Esta documentación también se llama *API de Greenfoot* (de «application programmers' interface»). La API nos muestra todas las clases disponibles y, para cada clase, todos los métodos disponibles. Puedes ver que Greenfoot ofrece cinco clases: `Actor`, `Greenfoot`, `GreenfootImage`, `MouseInfo`, y `World`.

El método que estamos buscando está en la clase de `Greenfoot`.

Ejercicio 3.20 Abre la API de Greenfoot en tu navegador. Selecciona la clase `Greenfoot`. En su documentación, busca la sección titulada «Method Summary». En esta sección, intenta buscar un método que pare la ejecución del escenario en ejecución. ¿Cómo se llama el método?

Ejercicio 3.21 ¿Espera este método algún parámetro? ¿Cuál es el tipo de vuelta?

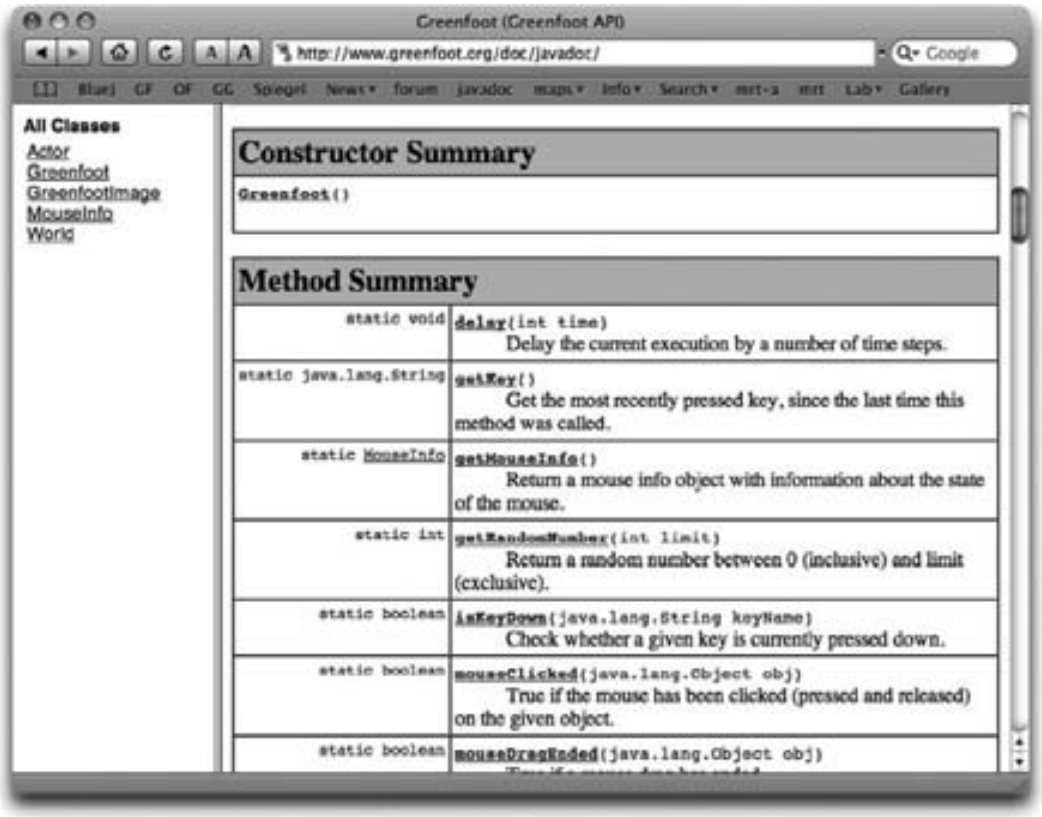


Figura 3.3
La API de Greenfoot API en la ventana de un navegador

Podemos ver la documentación de las clases de Greenfoot seleccionándolas de la lista a la izquierda. Para cada clase, el panel principal del navegador muestra un comentario general, detalles de sus constructores, y una lista de sus métodos. (Los constructores serán tratados en un capítulo posterior.)

Si navegamos por la lista de métodos disponibles de la clase `Greenfoot`, podemos encontrar un método llamado `stop` (parar). Éste es el método que usaremos para parar la ejecución cuando el cangrejo sea capturado.

Podemos usar este método escribiendo

```
Greenfoot.stop();
```

en nuestro código fuente.

Ejercicio 3.22 Añade código a tu escenario para que se pare cuando una langosta capture al cangrejo. Necesitarás decidir dónde añadir este código. Busca el lugar de tu código que se ejecuta cuando una langosta se come un cangrejo, y añade esta línea ahí.

Usaremos esta documentación de clases frecuentemente en el futuro para buscar detalles de los métodos que necesitamos usar. Sabremos algunos métodos de memoria con el tiempo, pero siempre habrá métodos que necesitamos buscar.

3.8 Añadiendo sonido

Otra forma de mejorar nuestro juego es añadirle sonidos. De nuevo, un método de la clase `Greenfoot` nos ayuda con esto.

Ejercicio 3.23 Abre la *Documentación de Clases Greenfoot* (del menú *Ayuda*), y mira la documentación de la clase `Greenfoot`. Busca los detalles del método que se usa para reproducir un sonido. ¿Cuál es su nombre? ¿Qué parámetros espera?

Mirando la documentación, podemos ver que la clase `Greenfoot` tiene un método llamado `playSound` (*reproduceSonido*). Espera el nombre de un fichero de sonido (como String) como un parámetro, y no devuelve nada.

Nota

Te puede gustar mirar la estructura de un escenario Greenfoot en tu sistema de ficheros. Si miras en la carpeta que contiene los escenarios del libro, puedes encontrar una carpeta para cada escenario de Greenfoot. Para el ejemplo del cangrejo, hay varias versiones diferentes (*little-crab*, *little-clab-2*, *little-crab-3*, etc.). Dentro de cada carpeta hay varios ficheros para cada clase del escenario, y otros ficheros auxiliares. Hay también dos carpetas de archivos multimedia: *images* guarda las imágenes y *sounds* guarda los ficheros de sonido.

Puedes ver los sonidos disponibles en esta carpeta, y puedes poner más sonidos disponibles guardando los archivos de sonido aquí.

En nuestro escenario del cangrejo, hay dos ficheros de sonido, llamados *slurp.wav* y *au.wav.*

Podemos reproducir fácilmente uno de estos sonidos con la siguiente llamada al método:

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

¡Inténtalo!

Ejercicio 3.24 Añade la reproducción de sonidos en tu escenario. Cuando un cangrejo se come un gusano, reproduce el sonido «slurp.wav». Cuando una langosta se come un cangrejo, reproduce el sonido «au.wav». Para hacer esto, tienes que colocar tu código donde esto sucede.

Sobre la grabación de sonido

Puedes también crear tus propios sonidos. Los sonidos incluidos se han grabado hablando en el micrófono del ordenador. Usa alguno de los programas de grabación de sonido gratuitos [1], graba tu propio sonido, y guárdalo (o expórtalo) a un fichero de sonido, en formatos WAV, AIFF, o AU. Cómo hacer tus propios sonidos se describe más tarde en el Capítulo 8.

[1] Usando un buscador de Internet, deberías poder encontrar varios programas gratuitos que permitan grabar y guardar sonidos. Uno que es bastante bueno es *Audacity* (http://audacity.sourceforge.net), pero hay muchos otros.

La versión *little-crab-4* de este escenario te muestra la solución. Esta versión incluye toda la funcionalidad que hemos visto hasta ahora: gusanos, langostas, control del teclado y sonido (Figura 3.4).

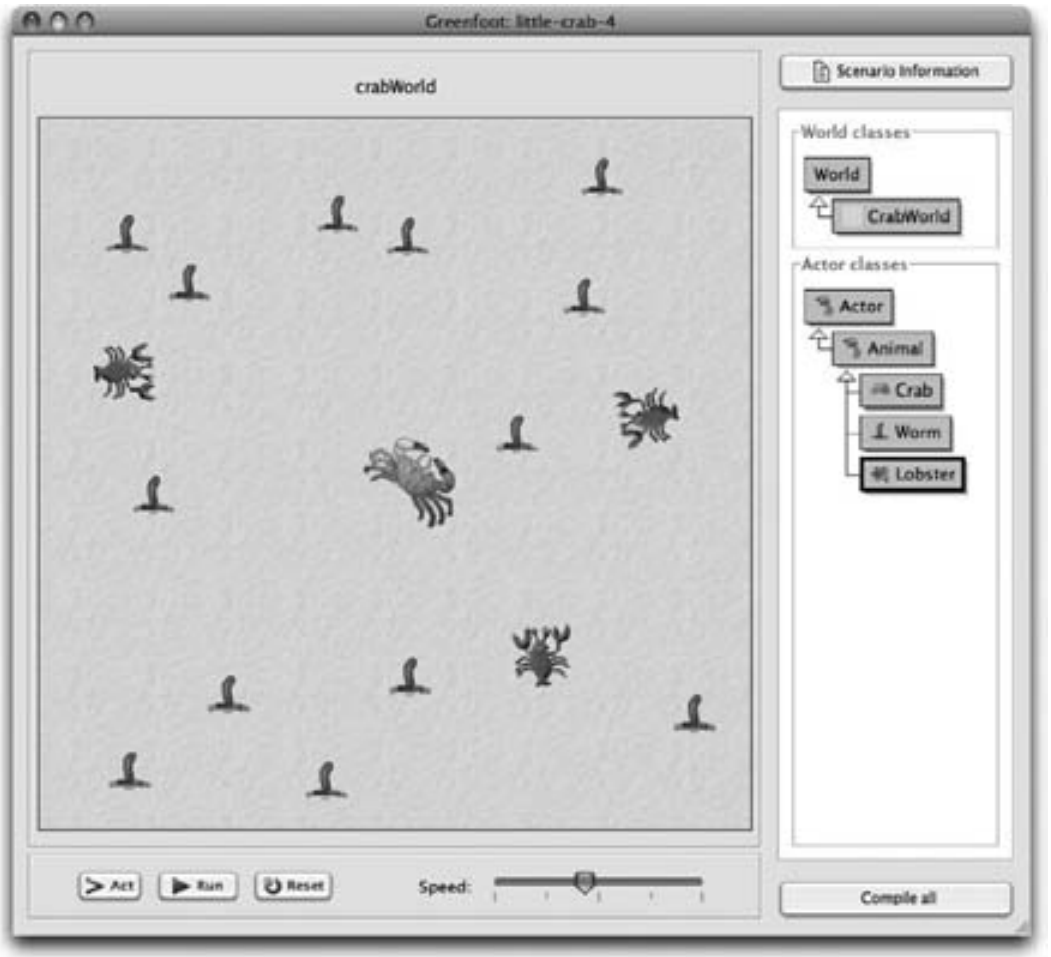


Figura 3.4
El juego del cangrejo con gusanos y langostas

Ejercicio 3.25 Si tienes un micrófono en tu ordenador, haz tus propios sonidos y úsalos cuando se come un gusano o un cangrejo. Graba los sonidos con cualquier programa de grabación de sonidos, almacénalos en la carpeta *sounds* del escenario, y úsalos en tu código.

3.9 Resumen de técnicas de programación

En este capítulo hemos visto más ejemplos de cómo usar una sentencia if —esta vez para girar sólo a veces y para reaccionar a teclas del teclado. Hemos visto también cómo llamar a métodos de otra clase, como los métodos `getRandomNumber`, `isKeyDown`, y `playSound` de la clase `Greenfoot`. Hacemos esto con la notación punto, con el nombre de la clase delante del punto.

Juntos, hemos visto ejemplos de cómo llamar a métodos desde tres sitios diferentes. Podemos llamar a métodos definidos en la clase actual (se llaman *métodos locales*), métodos definidos en una super-

clase (*métodos heredados*), y métodos estáticos de otras clases. Estos últimos métodos usan la notación punto. (Hay una versión adicional de una llamada a un método: llamar a métodos en otros objetos —lo veremos un poco más adelante.)

Otro aspecto importante que hemos explorado es cómo leer la documentación (API) de una clase existente para encontrar qué métodos tiene y cómo llamarlos.

Resumen de conceptos

- Cuando queremos llamar a un método que no está en nuestra propia clase ni ha sido heredado, necesitamos especificar la clase u objeto que ofrece el método antes del nombre del método, seguido de un punto. Esto se llama la **notación punto**.
- Los métodos que pertenecen a clases (en oposición a objetos), se señalan con la palabra clave **static** en su signatura. También se llaman **métodos de clase**.
- Una **definición de un método** define una nueva acción para los objetos de esta clase. La acción no se ejecuta inmediatamente sino que el método puede ser llamado más tarde para ejecutarlo.
- Los **comentarios** se escriben en el código fuente como explicaciones a los lectores humanos. Son ignorados por el ordenador.
- La **documentación de la API** lista todas las clases y métodos disponibles en Greenfoot. Tenemos que buscar métodos a menudo aquí.

Términos en inglés

Inglés	Español
comment	comentario
dot notation	notación punto
package	paquete
static method	método estático

CAPÍTULO

4

Terminando el juego del cangrejo

temas:	inicialización del mundo, configurar imágenes, animando imágenes
conceptos:	constructores, estado, variables de instancia (campos), asignación, new (crear objetos programáticamente)

En este capítulo, terminaremos el juego del cangrejo. «Terminar» aquí significa que es donde pararemos de discutir este proyecto en el libro. Por supuesto, un juego nunca se termina —siempre puedes pensar nuevas mejoras que puedes añadir. Te sugeriremos algunas ideas al final de este capítulo. En primer lugar, sin embargo, discutiremos varias mejoras en detalle.

4.1 Añadiendo objetos automáticamente

Estamos ya cerca de tener un pequeño juego jugable. Sin embargo, tenemos que hacer algunas cosas más. El primer problema que tenemos que resolver es el hecho de que siempre tenemos que colocar los actores (el cangrejo, langostas y gusanos) manualmente en el mundo. Sería mejor si esto pasara automáticamente.

Hay una cosa que ocurre automáticamente cada vez que compilamos con éxito: el mundo se crea. El objeto world, como vemos en la pantalla (el área cuadrada con color tierra) es una instancia de la clase `CrabWorld` (*MundoCangrejo*). Las instancias del mundo se tratan de una forma especial en Greenfoot: mientras que tenemos que crear instancias de nuestros actores nosotros mismos, el sistema de Greenfoot siempre crea automáticamente una instancia de nuestra clase mundo y la muestra en la pantalla.

Vamos a revisar el código fuente de `CrabWorld` (Código 4.1). (Si no tienes tu propio juego del cangrejo en este momento, puedes usar *little-crab-4* para este capítulo.)

Código 4.1
Código fuente de la clase `CrabWorld`

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

public class CrabWorld extends World
{
    /**
     * Crea el mundo del cangrejo (la playa). Nuestro mundo
     * tiene un tamaño de 560x560 celdas, donde cada
     * celda ocupa sólo 1 pixel.
     */
    public CrabWorld()
    {
        super(560, 560, 1);
    }
}
```

En esta clase, vemos las sentencias habituales import en la primera línea. (Veremos esta sentencia con detalle más tarde —por ahora es suficiente saber que esta línea siempre aparecerá al principio de nuestras clases en Greenfoot.)

A continuación sigue la cabecera de la clase, y un comentario (el bloque de líneas en color azulado con asteriscos —ya nos hemos encontrado con este bloque en el capítulo anterior). Los comentarios comienzan normalmente con un símbolo /** y terminan con */.

A continuación viene la parte interesante:

```
public CrabWorld()
{
    super(560, 560, 1);
}
```

Concepto:

Un **constructor** de una clase es un tipo especial de método que se ejecuta automáticamente cuando se crea un nuevo objeto.

Este método se denomina el *constructor* de esta clase. Un constructor es bastante similar a un método, pero hay algunas diferencias:

- Un constructor no tiene tipo de vuelta especificado entre la palabra clave «public» y el nombre.
- El nombre del constructor es siempre el mismo que el de la clase.

Un constructor es un tipo especial de método que se ejecuta siempre automáticamente cuando se crea una instancia de la clase. Puede hacer entonces lo que quiera hacer para inicializar la nueva instancia en su estado inicial.

En nuestro caso, el constructor fija que el tamaño del mundo sea el que queremos (560 por 560 celdas) y una *resolución* (1 pixel por celda). Hablaremos de la resolución del mundo con más detalle más adelante en el libro.

Como el constructor se ejecuta cada vez que el mundo se crea, podemos usarlo para automatizar la creación de nuestros actores. Si insertamos código en el constructor que crea un actor, este código se ejecutará también. Por ejemplo,

```
public CrabWorld()
{
    super(560, 560, 1);
    addObject( new Crab(), 150, 100 );
}
```

Este código creará automáticamente un nuevo cangrejo, y lo colocará en la posición x=150, y=100 del mundo. La ubicación 150,100 es 150 celdas del borde izquierdo del mundo, y 100 celdas del borde superior. El origen —el punto 0,0— de nuestro sistema de coordenadas está en la esquina superior izquierda del mundo (Figura 4.1).

Estamos usando dos cosas nuevas aquí: el método `addObject` (*añadeObjeto*) y la sentencia `new` (*nuevo*) para crear un nuevo cangrejo.

El método `addObject` es un método de la clase `World`. Podemos buscarlo en la documentación de la clase `World`. Allí, podemos ver la siguiente signatura:

```
void addObject(Actor object, int x, int y)
```

Si leemos la signatura de principio a fin, nos dice lo siguiente:

- El método no devuelve ningún resultado (tipo de vuelta void).

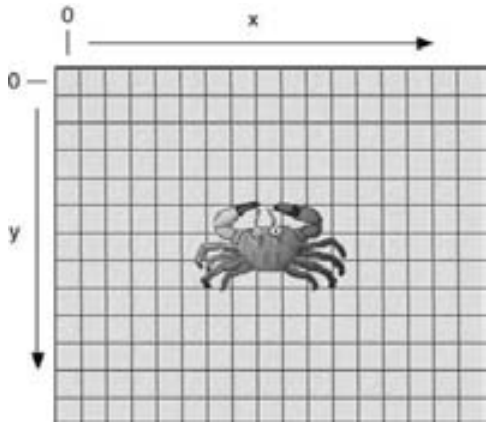


Figura 4.1
El sistema de coordenadas del mundo

- El nombre del método es `addObject`.
- El método tiene tres parámetros, denominados `object`, `x`, e `y`.
- El tipo del primer parámetro es `Actor`, el tipo de los otros dos es `int`.

Este método puede ser usado para añadir un nuevo actor a nuestro mundo. Como el método pertenece a la clase `World` y `CrabWorld` es un `World` (hereda de la clase `World`), este método está disponible en nuestra clase `CrabWorld`, y podemos llamarlo directamente.

4.2 Creando nuevos objetos

El método `addObject` nos permite añadir un objeto actor al mundo. Sin embargo, para añadir un objeto, tenemos que tener un objeto que añadir.

La palabra clave `new` de Java nos permite crear nuevos objetos de cualquiera de las clases existentes. Por ejemplo, la expresión

```
new   Crab()
```

Concepto:

Se pueden crear objetos Java programáticamente (desde tu código) usando la palabra clave **new**.

crea un nuevo ejemplar de la clase `Crab`. La expresión para crear nuevos objetos siempre comienza con la palabra clave `new`, seguida del nombre de la clase de la que queremos crear el objeto, y una lista de parámetros (que está vacía en nuestro ejemplo). La lista de parámetros nos permite pasar parámetros al constructor. Como no hemos especificado un constructor para nuestra clase `Crab`, la lista de parámetros por defecto está vacía.

Cuando creamos un nuevo objeto, tenemos que hacer algo con él. Podemos ahora usarlo en vez del parámetro actor del método `addObject` para añadir este objeto al mundo.

```
addObject( new Crab(), 150, 100);
```

Los otros dos parámetros especifican las coordenadas *x* e *y* de la posición donde deseamos añadir el objeto.

Ejercicio 4.1 Añade código al constructor de `CrabWorld` de tu propio proyecto para crear un cangrejo automáticamente, como se ha discutido anteriormente.

Ejercicio 4.2 Añade código para crear automáticamente tres langostas en `CrabWorld`. Puedes escoger posiciones arbitrarias para colocarlas.

Ejercicio 4.3 Añade código para crear 10 gusanos en posiciones arbitrarias de `CrabWorld`.

Ejercicio 4.4 Mueve todo el código que crea objetos a un método separado, denominado `populateWorld` (*poblarMundo*), en la clase `CrabWorld`. Necesitas declarar el método `populateWorld` tú mismo (no necesita ningún parámetro ni devuelve nada) y llamarlo desde el constructor. Inténtalo.

Ejercicio 4.5 Usa números aleatorios para las coordenadas de los gusanos. Puedes hacer esto reemplazando tus valores de coordenadas con llamadas para obtener números aleatorios de la clase Greenfoot.

Deberías tener ahora una versión de tu proyecto del cangrejo que coloca el cangrejo, las langostas y los gusanos en el mundo automáticamente, cada vez que compilas el escenario. (Si te da problemas, mira en el escenario *little-crab-5* de los proyectos del libro —incluye este código y el resto de cambios de este capítulo.)

4.3 Animando imágenes

La siguiente cosa que vamos a ver en este escenario es cómo animar la imagen del cangrejo. Para hacer que el movimiento del cangrejo mejore, planeamos cambiar el cangrejo de forma que mueva sus patas mientras anda.

Esta animación se consigue con un simple truco: tenemos dos imágenes diferentes del cangrejo (en nuestro escenario, se llaman *crab.png* y *crab2.png*), y simplemente vamos alternando la imagen del cangrejo entre estas dos versiones de forma rápida. La posición de las patas del cangrejo en estas imágenes es un poco diferente (Figura 4.2).

a) cangrejo con patas hacia fuera

b) cangrejo con patas hacia dentro

Figura 4.2
Dos imágenes un poco diferentes para el cangrejo

El efecto de esto (alternar entre estas imágenes) será que parece que el cangrejo está moviendo sus patas.

Para hacer esto, vamos a introducir dos nuevos conceptos: variables e imágenes en Greenfoot.

4.4 Imágenes en Greenfoot

Greenfoot proporciona una clase denominada `GreenfootImage` (*ImagenGreenfoot*) que ayuda en el uso y manipulación de imágenes. Podemos obtener una imagen creando un nuevo objeto `Green-`

Concepto:

Los actores de Greenfoot mantienen su imagen visible almacenando un objeto de tipo `GreenfootImage`.

`footImage` —usando la palabra clave new de Java— con el nombre del fichero de la imagen como parámetro en el constructor. Por ejemplo, para acceder a la imagen *crab2.png*, escribimos

```
new  GreenfootImage("crab2.png")
```

El nombre del fichero que indicamos debe existir en el directorio *images* del escenario.

Todos los actores de Greenfoot tienen imágenes. Por defecto, los actores obtienen su imagen de su clase. Asignamos una imagen a una clase cuando la creamos, y cada objeto creado de esa clase recibirá, tras la creación, una copia de la misma imagen. Esto no significa, sin embargo, que todos los objetos de la misma clase deban tener siempre la misma imagen. Cada actor individual puede decidir cambiar su imagen en cualquier momento.

Ejercicio 4.6 Comprueba la documentación de la clase `Actor`. Hay dos métodos que nos permiten cambiar la imagen de un actor. ¿Cómo se llaman, y cuáles son sus parámetros? ¿Qué devuelven?

Si hiciste el ejercicio anterior, habrás visto que uno de los métodos para establecer la imagen de un actor espera un parámetro de tipo `GreenfootImage`. Éste es el método que usaremos. Podemos crear un objeto `GreenfootImage` con un fichero de imagen como se ha descrito antes, y entonces podemos usar el método `setImage` (*estableceImagen*) del actor. Aquí se muestra el trozo de código que lo hace:

```
setImage(new GreenfootImage("crab2.png"));
```

Observa que hacemos dos cosas en esta línea. Llamamos al método `setImage`, que espera una imagen como parámetro:

```
setImage( alguna-imagen );
```

Y en vez de poner el objeto imagen que queremos usar, escribimos

```
new GreenfootImage("crab2.png")
```

Esto crea un objeto imagen a partir del fichero de imagen llamado (*crab2.png*). Cuando se ejecuta la línea entera, se ejecuta primero la parte interna del código —la creación del objeto `GreenfootImage`. A continuación, se ejecuta la llamada al método `setImage`, y el objeto imagen que acabamos de crear se pasa como parámetro.

Para nuestro objetivo, es mejor separar la creación del objeto imagen del establecimiento de la imagen de la clase. La razón de esto es que queremos alternar una y otra vez las imágenes muchas veces mientras el cangrejo está andando. Esto significa que queremos establecer la imagen muchas veces, pero nos gustaría crear cada una de las dos imágenes sólo una vez.

Por tanto, primero crearemos las imágenes y las almacenaremos, y luego usaremos las imágenes almacenadas (sin crearlas de nuevo) una y otra vez para alternar la imagen mostrada.

Para almacenar las dos imágenes en nuestro objeto cangrejo, necesitamos un nuevo constructor que no hemos visto antes: una variable.

4.5 Variables de instancia (campos)

A menudo, nuestros actores necesitan recordar alguna información. En los lenguajes de programación, esto se consigue almacenando la información en una *variable*.

Java soporta varios tipos de variables. El primero que vamos a ver aquí se llama una *variable de objeto*, *variable de instancia*, o *campo*. (Estos términos son sinónimos.) Veremos otros tipos de variables más tarde.

Concepto:

Las variables de instancia (también denominadas **campos**) pueden ser utilizadas para almacenar información (objetos o valores) para su uso posterior.

Una variable de instancia es una porción de memoria que pertenece al objeto (la *instancia* de la clase, de ahí el nombre). Cualquier cosa almacenada en ella puede ser recordada mientras el objeto exista, y se puede acceder a ella más tarde.

Una variable de instancia se declara en una clase escribiendo la palabra clave `private` (privado) seguida del tipo de la variable y del nombre de la variable:

```
private tipo-variable nombre-variable;
```

El tipo de la variable define lo que queremos almacenar en ella. En nuestro caso, como queremos almacenar en nuestra variable objetos del tipo `GreenfootImage`, el tipo debe ser `GreenfootImage`. El nombre de la variable nos da la oportunidad de dar un nombre a la variable para que nos podamos referir a ella más tarde. Debe describir para qué usamos la variable.

Vamos a mover en nuestra clase `Crab` un ejemplo (Código 4.2).

Código 4.2
La clase `Crab` con dos variables de instancia

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

// comentario omitido

public class Crab extends Animal
{

    private GreenfootImage image1;
    private GreenfootImage image2;

    // métodos omitidos


}
```

En este ejemplo, hemos declarado dos variables en nuestra clase `Crab`. Ambas son del tipo `GreenfootImage`, y se llaman `image1` e `image2`.

Siempre escribiremos las declaraciones de las variables de instancia al principio de nuestra clase, antes de los constructores y los métodos. Java no nos obliga a hacerlo así, pero es una buena práctica, para que siempre podamos encontrar las declaraciones de las variables fácilmente cuando necesitemos verlas.

Ejercicio 4.7 Antes de añadir este código, pincha con el botón derecho en un objeto cangrejo de tu mundo, y selecciona inspeccionar del menú emergente del cangrejo. Apunta todas las variables que se muestran en el objeto cangrejo.

Ejercicio 4.8 ¿Por qué piensas que el cangrejo no tiene ninguna variable, incluso no las hemos declarado en nuestra clase `Crab`?

Ejercicio 4.9 Añade las declaraciones de variable mostradas en el Código 4.2 en tu versión de la clase `Crab`. No olvides comprobar que la clase compila bien.

Ejercicio 4.10 Después de añadir las variables, inspecciona tu objeto cangrejo de nuevo. Apunta las variables y sus valores (mostrados en cajas blancas).

Observa que la declaración de estas dos variables `GreenfootImage` no nos proporciona dos objetos `GreenfootImage`. Simplemente nos da un espacio vacío para almacenar estos dos objetos (Figura 4.3). En esta figura, las variables de instancia se muestran como dos cajas blancas.

Figura 4.3
Un objeto cangrejo con dos variables de instancia vacías

A continuación, tenemos que crear dos objetos imagen y almacenarlos en las variables. La creación de los objetos ya la hemos visto antes. Se conseguía con el siguiente trozo de código:

```
new GreenfootImage("crab2.png")
```

Para almacenar los objetos en las variables, necesitamos un constructor de Java conocido como *asignación*.

4.6 Asignación

Una asignación es una sentencia que nos permite almacenar algo en una variable. Se escribe con un símbolo igual:

```
variable = expresión;
```

Concepto:

Una **sentencia de asignación** asigna un objeto o un valor a una variable.

A la izquierda del símbolo igual se pone el nombre de la variable en que queremos almacenar algo, y a la derecha está la cosa que queremos almacenar. Como el símbolo igual se usa para asignar, también se le llama *símbolo de asignación*. Normalmente lo leemos como «toma el valor», de esta forma: «la variable toma el valor expresión».

En nuestro ejemplo del cangrejo, escribimos

```
image1 = new GreenfootImage("crab.png");
image2 = new GreenfootImage("crab2.png");
```

Estas dos líneas de código crearán las dos imágenes que queremos usar y las almacenarán en nuestras dos variables `image1` e `image2`. Tras ejecutar estas sentencias, tenemos tres objetos (un cangrejo y dos imágenes), y las variables del cangrejo contienen referencias a las imágenes. Esto se muestra en la Figura 4.4.

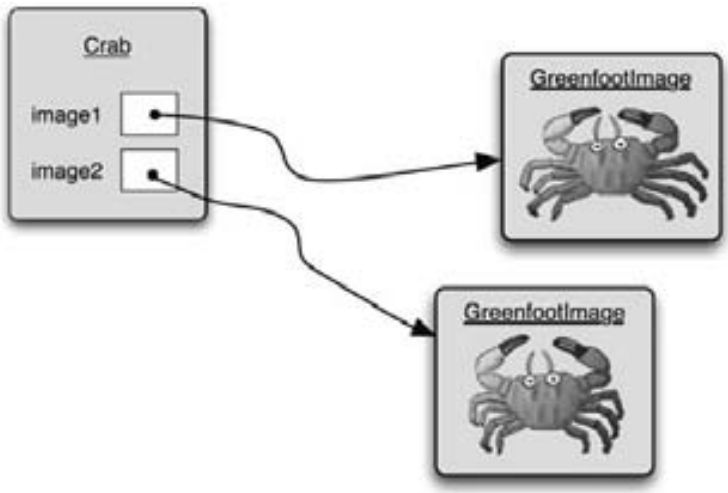


Figura 4.4
Un objeto cangrejo con dos variables, apuntando a los objetos imagen

Concepto:

Cuando asignamos un objeto a una variable, la variable contiene una **referencia** a ese objeto.

La siguiente pregunta sobre la creación de estas imágenes es dónde poner el código para crear las imágenes y almacenarlas en las variables. Como esto debe hacerse cuando se crea el objeto del cangrejo, y no cada vez que actuamos, no podemos ponerlo en el método `act`. En vez de eso, vamos a poner el código en un constructor.

4.7 Usando constructores de actor

Al principio de este capítulo, hemos visto cómo usar el constructor de la clase mundo para inicializar el mundo. De forma similar, podemos usar un constructor de una clase actor para inicializar el actor. El código del constructor se ejecuta una vez cuando se crea el actor. El Código 4.3 muestra un constructor para la clase `Crab` que inicializa las dos variables de instancia, para lo que crea los dos objetos imagen y los asigna a las variables.

Código 4.3
Inicializando las variables en el constructor

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

// comentario omitido

public class Crab extends Animal
{

    private GreenfootImage image1;
    private GreenfootImage image2;

    /**
```

```
    *   Crea un cangrejo y lo inicializa con dos imágenes.
    */
    public Crab()
    {
        image1 = new GreenfootImage("crab.png");
        image2 = new GreenfootImage("crab2.png");
        setImage(image1);
        }

    // métodos omitidos
}
```

Las mismas reglas descritas para el constructor de `World` son válidas para el constructor de `Crab`:

- La signatura de un constructor no incluye el tipo de vuelta.
- El nombre del constructor es el mismo que el nombre de la clase.
- El constructor se ejecuta automáticamente cuando se crea un objeto cangrejo.

La última regla es que el constructor se ejecuta automáticamente —se asegura que los objetos de las imágenes se crean automáticamente y se asignan cuando se crea un cangrejo—. Por tanto, después de crear el cangrejo, la situación es la mostrada en la Figura 4.4.

La última línea del constructor fija la primera de las dos imágenes creadas como la imagen actual del cangrejo:

```
setImage(image1);
```

Esto demuestra cómo el nombre de la variable (`image1`) puede ser usada ahora para referirse al objeto imagen almacenado en ella.

Ejercicio 4.11 Añade este constructor a tu clase `Crab`. No verás ningún cambio en la conducta del cangrejo, pero la clase debería compilar, y deberás ser capaz de crear cangrejos.

Ejercicio 4.12 Inspecciona tu objeto cangrejo de nuevo. Apunta de nuevo las variables y sus valores. Compara éstos con los valores que apuntaste previamente.

4.8 Alternando las imágenes

Hemos alcanzado ahora un punto en que el cangrejo tiene dos imágenes disponibles para hacer la animación. Pero no vemos que se anime todavía. Esto es ahora relativamente simple.

Para hacer la animación, necesitamos alternar entre nuestras dos imágenes. En otras palabras, en cada paso, si estamos mostrando actualmente `image1`, queremos que se muestre `image2`, y *viceversa*. El pseudocódigo para expresar esto es:

```
if (nuestra imagen actual es image1) then
    usa image2 ahora
```

```
else
    usa image1 ahora
```

El pseudocódigo, tal como lo hemos usado aquí, es una técnica para expresar una tarea en una estructura que es en parte código Java real, y en parte lenguaje natural (español en nuestro caso). A menudo nos ayuda a escribir nuestro código real. Podemos mostrar ahora lo mismo con código Java real (Código 4.4).

Código 4.4
Alternando entre dos imágenes

```
if ( getImage() == image1 )
{
    setImage(image2);
}
else
{
    setImage(image1);
}
```

Concepto:

Podemos probar si dos cosas son **iguales** usando el símbolo doble igual: **==**.

En este trozo de código, vemos varios elementos nuevos:

- El método `getImage` puede usarse para recibir la imagen actual del actor.
- El operador == (dos signos igual) se puede usar para comparar un valor con otro. El resultado es *true* o *false*.
- La sentencia if tiene una forma extendida que no hemos visto antes. Esta forma tiene una palabra clave else después del primer cuerpo de la sentencia if, seguido de otro bloque de sentencias. Veremos esta nueva forma en la siguiente sección.

Error

Es un error común mezclar el operador de asignación (=) con el operador de igualdad (==). Si quieres comprobar si dos valores o variables son iguales, debes escribir dos símbolos igual.

4.9 La sentencia if/else

Antes de continuar, vamos a ver con más detalle la sentencia if. Como acabamos de ver, una sentencia if se puede escribir con la forma

```
if ( condición )
{
    sentencias;
}
else
{
    sentencias;
}
```

Concepto:

La **sentencia if/else** ejecuta un trozo de código cuando se cumple una condición, y otro diferente cuando no se cumple.

Esta sentencia if contiene dos bloques (pares de llaves rodeando una lista de sentencias): la *cláusula-if* y la *cláusula-else* (en este orden).

Cuando se ejecuta esta sentencia if, primero se evalúa la condición. Si la condición es true, entonces la cláusula if se ejecutará, y la ejecución continúa tras la cláusula else. Si la condición es falsa, no se ejecuta la cláusula if, sino que se ejecuta la cláusula else. Por tanto, uno de los dos bloques de sentencias se ejecuta siempre, pero nunca ambos.

La parte else con el segundo bloque es opcional —si se omite queda la versión más corta de la sentencia if que hemos visto antes.

Hemos visto ya todo lo necesario para terminar esta tarea. Es hora de ponernos a teclear de nuevo para probarlo.

Ejercicio 4.13 Añade el código para alternar de imagen, como se mostraba en el Código 4.4, al método `act` de tu propia clase `Crab`. Pruébalo. (Si obtienes un error, arréglalo, te debería ir bien.). También pincha en el botón `Actuar` en vez de en el botón `Ejecutar` en `Greenfoot`— esto nos permite observar la conducta más claramente.

Ejercicio 4.14 En el Capítulo 3, hemos visto cómo usar métodos divididos para las subtareas, en vez de escribir todo el código dentro del método `act`. Haz esto con el código para intercambiar la imagen. Crea un nuevo método denominado `switchImage` (*intercambiarImagen*), mueve tu código de intercambiar imágenes a él, y llama a este método dentro de tu método `act`.

Ejercicio 4.15 Llama al método `switchImage` de forma interactiva desde el menú emergente del cangrejo. ¿Funciona?

4.10 Contando gusanos

La última cosa que vamos a ver con el cangrejo es cómo contar. Queremos añadir una nueva funcionalidad, que permita que el cangrejo cuente cuántos gusanos se ha comido, y cuando se haya comido ocho gusanos, gana el juego. Queremos también reproducir un breve «sonido de victoria» cuando esto suceda.

Para hacer esto, necesitamos añadir varias cosas a nuestro código del cangrejo. Necesitamos

- una variable de instancia para almacenar el número actual de gusanos comidos;
- una asignación que inicialice esta variable a 0 al principio;
- código para incrementar nuestra cuenta cada vez que nos comemos un gusano; y
- código que compruebe si nos hemos comido ocho gusanos, y pare el juego y reproduzca el sonido si hemos ganado.

Hagamos las tareas en el orden en que las acabamos de listar aquí.

Podemos definir una nueva variable de instancia siguiendo el patrón introducido en la sección 4.5. A continuación de las definiciones de las dos variables de instancia existentes, añadimos la línea

```
private int wormsEaten;
```

La palabra `private` se usa al principio de todas nuestras definiciones de variables de instancia. Las siguientes dos palabras son el tipo y el nombre de nuestra variable. El tipo `int` indica que queremos almacenar enteros (números enteros) en esta variable, y el nombre `wormsEaten` (*gusanosComidos*) indica lo que pensamos hacer con ella.

A continuación, añadimos la siguiente línea al final de nuestro constructor:

```
wormsEaten = 0;
```

Esto inicializa la variable `wormsEaten` a 0 cuando se crea el cangrejo. Siendo estrictos, esto es redundante, ya que las variables de instancia de tipo `int` se inicializan a 0 automáticamente. Sin embargo, como a veces queremos que el valor inicial no sea cero, es una buena práctica escribir nuestra propia inicialización.

Lo último que nos queda es contar los gusanos, y comprobar si hemos llegado a ocho. Necesitamos hacer esto cada vez que nos comemos un gusano, por lo que podemos revisar el método `lookForWorm`, donde tenemos el código que se come los gusanos. Aquí, añadimos una línea de código para incrementar la cuenta de gusanos:

```
wormsEaten = wormsEaten + 1;
```

En esta asignación, el lado derecho del símbolo de asignación se evalúa primero (`wormsEaten+1`). Por tanto, leemos el valor actual de `wormsEaten` y le sumamos 1. A continuación asignamos el resultado a la variable `wormsEaten`. Como resultado, la variable se incrementará en 1.

A continuación, necesitamos una sentencia if que compruebe si nos hemos comido ya ocho gusanos, y en este caso, que reproduzca el sonido y pare la ejecución.

El Código 4.5 muestra el método completo `lookForWorm`. El fichero de sonido usado aquí (*fanfare.wav*) se incluye en el directorio *sounds* de tu escenario, así que puede ser reproducido.

Ejercicio 4.16 Añade el código comentado a tu escenario. Pruébalo y comprueba que funciona.

Código 4.5
Contando gusanos y comprobando si hemos ganado

```
/**
 * Comprueba si nos hemos encontrado con un gusano.
 * Si lo vemos, nos lo comemos. Si no, no hacemos nada. Si nos
 * hemos comido ocho gusanos, ganamos.
 */
public void lookForWorm()
{
    if ( canSee(Worm.class) )
    {
        eat(Worm.class);
        Greenfoot.playSound("slurp.wav");

        wormsEaten = wormsEaten + 1;
        if (wormsEaten == 8)
        {
```

```
                Greenfoot.playSound("fanfare.wav");
                Greenfoot.stop();

            }
        }
    }
```

Ejercicio 4.17 Como prueba adicional, abre un inspector de objetos a tu objeto cangrejo (selecciona inspeccionar del menú emergente del cangrejo) antes de comenzar a jugar al juego. Deja abierto el inspector y mira la variable `wormsEaten` mientras juegas.

4.11 Más ideas

El escenario *little-crab-5*, de la carpeta de escenarios del libro, contiene una versión del proyecto que ya incluye todas las extensiones explicadas hasta aquí.

Vamos a dejar este escenario ahora, y vamos a empezar con un ejemplo diferente, aunque hay muchas cosas obvias (y probablemente muchas más menos obvias) que puedes hacer con este proyecto. Algunas de estas ideas son

- usar imágenes diferentes para el fondo y para los actores;
- usar más clases diferentes de actores;
- no moverse hacia delante automáticamente, sino sólo cuando se pulsa la tecla flecha hacia arriba;
- construir un juego para dos jugadores, introduciendo una segunda clase controladora del teclado, que escuche a teclas diferentes;
- hacer que surjan nuevos gusanos cuando un gusano es comido (o de forma aleatoria); y
- muchas más que se te ocurran a ti mismo.

Ejercicio 4.18 La imagen del cangrejo cambia de forma bastante rápida cuando el cangrejo anda, lo que hace que el cangrejo parezca un poco hiperactivo. Podría ser un poco más bonito si la imagen del cangrejo cambiara sólo cada segundo o cada tercio del ciclo del método `act`. Intenta implementar esto. Para hacerlo, puedes añadir un contador que se incrementa en el método `act`. Cada vez que alcance el valor 2 (o 3), la imagen cambia, y el contador se reinicia a 0.

4.12 Resumen de técnicas de programación

En este capítulo, hemos visto varios conceptos nuevos de programación. Hemos visto cómo los constructores se usan para inicializar objetos —los constructores se ejecutan siempre que se crea un nuevo objeto—. Hemos visto también cómo usar variables de instancia —también llamadas *campos*— y las sentencias de asignación para almacenar información, y cómo acceder a esta información más tarde. Hemos usado la sentencia `new` para crear programáticamente nuevos objetos y, finalmente, hemos

visto la versión completa de la sentencia if, que incluye una parte *else* que se ejecuta cuando la condición no es cierta.

Con todas estas técnicas juntas, podemos ahora escribir ya una buena cantidad de código.

Resumen de conceptos

- Un **constructor** de una clase es un tipo especial de método que se ejecuta automáticamente cuando se crea un nuevo objeto.
- Los objetos Java se pueden crear programáticamente (desde tu código) usando la palabra clave **new.**
- Los actores de Greenfoot mantienen su imagen visible conteniendo un objeto del tipo **GreenfootImage**. Se guardan en una variable de instancia heredada de la clase Actor.
- Las **variables de instancia** (también denominadas **campos**) pueden usarse para almacenar información (objetos o valores) para su uso posterior.
- Una **sentencia de asignación** asigna un objeto o un valor a una variable.
- Cuando se asigna un objeto a una variable, la variable contiene una **referencia** a ese objeto.
- Podemos comprobar si dos cosas son **iguales** usando el símbolo doble igual: **==**.
- La **sentencia if/else** ejecuta un trozo de código cuando se cumple una condición, y ejecuta un trozo diferente si no se cumple.

Términos en inglés

Inglés	Español
assignment	asignación
code snippet	trozo de código
field	campo
instance variable	variable de instancia
pseudo-code	pseudocódigo
reference	referencia
test	prueba, probar
variable	variable

INTERMEDIO

1 Compartiendo tus escenarios

En esta sección, no introduciremos nuevas técnicas de programación, sino que daremos un pequeño rodeo para ver cómo puedes compartir con otros lo que has creado. Con «otros» queremos decir desde tu amigo que se sienta a tu lado, o cualquier programador Greenfoot del otro lado del mundo. En estos tiempos con una Internet global, no hay ya mucha diferencia.

I1.1 Exportando tu escenario

Cuando has terminado de escribir un escenario —puede ser un juego o una simulación— puedes querer que otros usuarios lo usen. Esos otros usuarios deberían tener la oportunidad de lanzar (y relanzar) el juego, pero no necesitan acceder al diagrama de clases o al código fuente. No deberían modificar el juego, sino que simplemente deberían poder usarlo.

En Greenfoot, esto se hace exportando el escenario. Puedes exportar tu escenario seleccionando *Exportar* del menú *Escenario*. Te mostrará un diálogo que te deja escoger entre tres opciones: *Aplicación*, *Página Web*, y *Publicar*.

I1.2 Exportar a una aplicación

La primera opción de exportación es exportar una aplicación. Una aplicación es un programa independiente que los usuarios pueden ejecutar localmente en su ordenador.

Para hacer esto, escoge *Aplicación* en el diálogo de exportación. Puedes escoger a continuación un directorio y un nombre para el escenario ejecutable que vas a crear (Figura I1.1).

Para que el escenario funcione bien cuando lo exportes, es importante que crees automáticamente todos los actores que quieras ver en la pantalla al principio del juego. El usuario no podrá crear objetos interactivamente. Eso significa que normalmente tu mundo deberá tener un método «repoblar», como el que creamos en el juego del cangrejo.

Concepto:

Un **fichero jar** es un fichero con extensión *jar* que contiene todas las clases Java que pertenecen a una aplicación.

Usando esta función, se creará un *fichero jar ejecutable*. Es un fichero con extensión «*.jar*» (abreviatura de *Java Archive*), que puede ser ejecutado en muchos sistemas operativos (siempre que tengan Java instalado). Simplemente pincha dos veces en el fichero jar para ejecutarlo.

Cuando se ejecuta la aplicación, el escenario será como el que hicimos en Greenfoot, excepto que no aparecerán el diagrama de clases ni el botón `Compilar`. El usuario podrá ejecutar el escenario, pero no podrá editarlo ni compilarlo.

La opción «Bloquea escenario» desactiva que los actores se muevan por el mundo antes de lanzar la aplicación, y también quita el botón *Actuar* y la barra de control deslizante de la velocidad. Si tienes un juego, normalmente querrás bloquear el escenario, mientras que en las simulaciones u otros escenarios más experimentales, puede que quieras dejarlo desbloqueado para permitir a los usuarios que experimenten más.

Figura I1.1
Exportando un escenario a una aplicación

I1.3 Exportar a una página web

La segunda opción es exportar tu escenario a una página web (Figura I1.2). Las opciones del diálogo de exportación son como antes, pero esta función crea una página web (en formato HTML) y convierte tu escenario a un applet que se ejecuta en esa página web.

Puedes ejecutar tu escenario abriendo la página web generada en un navegador web.

Figura I1.2
Exportar a una página web

Si tienes acceso a un servidor web, puedes publicar esta página en la web. Si no tienes acceso a un servidor web, entonces la siguiente opción puede ser la más adecuada para ti.

I1.4 Publicando en la Galería de Greenfoot

La última opción para exportar que tienes es publicar tu propio escenario en la *Galería de Greenfoot*. La Galería de Greenfoot es un sitio web (su dirección es http://greenfootgallery.org) que permite que los usuarios de Greenfoot suban sus escenarios para que todo el mundo pueda verlos y jugar con ellos.

El diálogo de exportación (Figura I1.3) te muestra la dirección del sitio web en la parte superior. Pincha aquí para abrir el sitio web y mira lo que hay. Probablemente es recomendable que visites el sitio web primero.

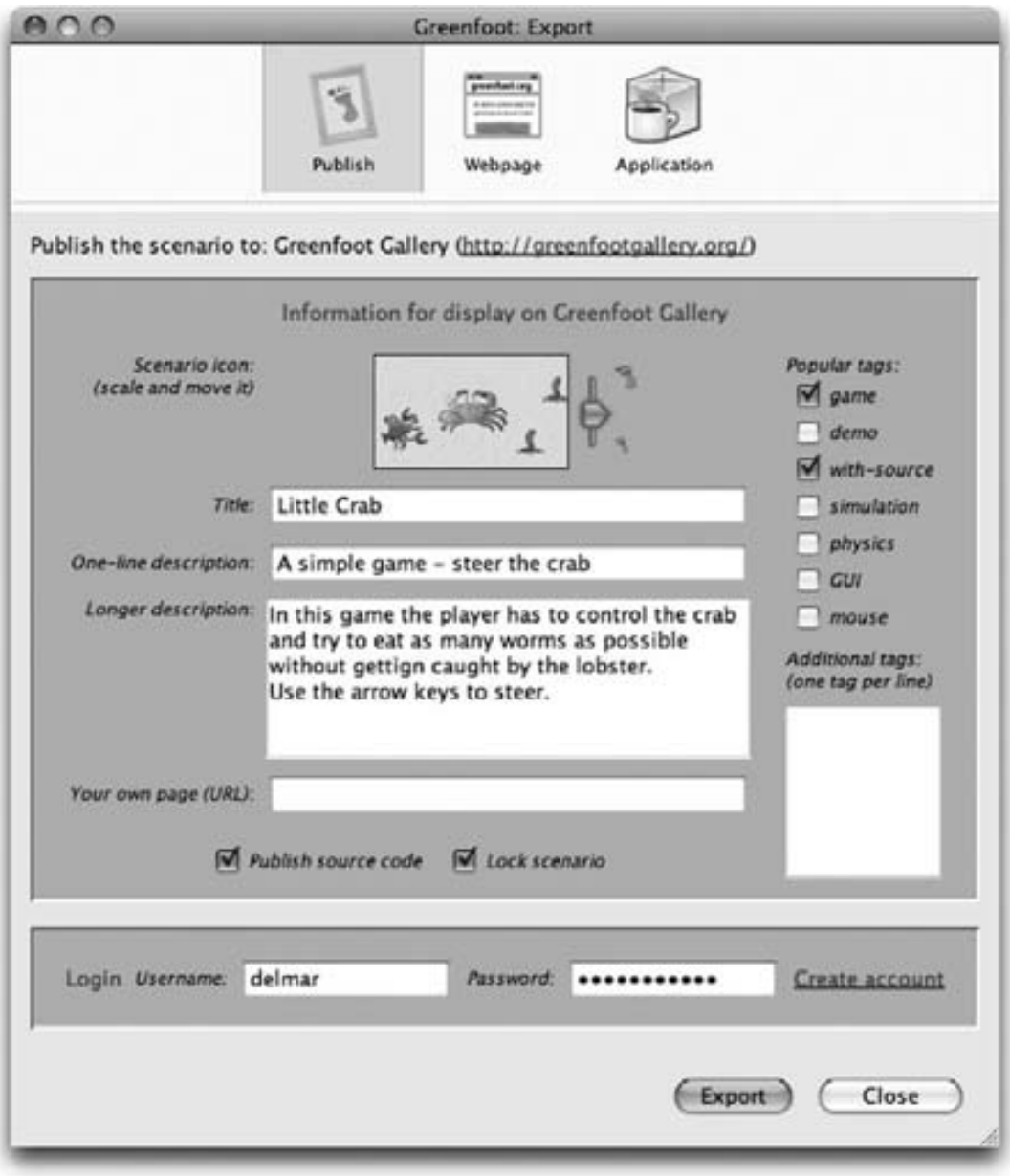


Figura I1.3
Publicando en la Galería de Greenfoot

Concepto:

Un **applet** es una versión de un programa Java que puede ejecutarse en una página web dentro de un navegador web.

En la Galería, todo el mundo puede ver y ejecutar escenarios, pero si quieres puntuarlos, hacer comentarios o subir tus propios escenarios, necesitas crearte una cuenta en el sitio. Se hace fácil y rápidamente.

Tras crear una cuenta, puedes subir fácilmente tus propios escenarios a la Galería de Greenfoot, usando el diálogo mostrado en la Figura I1.3. El diálogo te permite añadir un icono, una descripción, y etiquetas que identifiquen tu escenario.

Si escoges publicar el código fuente (seleccionando la casilla `Publicar código fuente`), todo tu código fuente completo será copiado en el sitio web de la Galería, donde todo el mundo podrá bajárselo para leerlo, y hacer su propia versión de tu escenario.

Tus escenarios publicados pueden ser cambiados y mejorados más tarde, simplemente volviendo a exportar de nuevo con el mismo título.

Publicar tus escenarios en la Galería puede ser una buena forma de obtener realimentación de otros usuarios: comentarios sobre lo que funciona y no funciona, así como sugerencias de qué podrías añadir al programa. La Galería es también un buen sitio para obtener ideas de nuevas funcionalidades, o para aprender a hacer cosas. Simplemente busca escenarios con código fuente, bájate el código, y mira cómo otros programadores han implementado sus clases.

Resumen de conceptos

- Un **fichero jar** es un único fichero con la extensión jar que contiene todas las clases Java que pertenecen a una aplicación.
- Un **applet** es una versión de un programa Java que se puede ejecutar en una página web en un navegador web.

CAPÍTULO

5

Haciendo música: un piano virtual

temas: sonido

conceptos: abstracción, bucles, *arrays*, estructura OO

En este capítulo empezaremos con un nuevo escenario: un piano en el que podemos tocar con el teclado de nuestro ordenador. La Figura 5.1 muestra cómo podría ser cuando lo terminemos.

Empezamos de nuevo abriendo un escenario de los escenarios del libro: *piano-1*. Ésta es una versión de nuestro escenario que contiene los recursos que necesitaremos (ficheros de sonido e imagen), no mucho más. Lo usaremos como escenario base para comenzar a escribir el código del piano.

Figura 5.1
El objetivo de este capítulo: un piano virtual

Ejercicio 5.1 Abre el escenario *piano-1* y examina el código de las dos clases existentes, `Piano` y `Key`. Comprueba que entiendes el código del escenario y lo que hace.

Ejercicio 5.2 Crea un objeto de la clase `Key` y colócalo en el mundo. Crea varias teclas y colócalas juntas.

5.1 Animando la tecla

Cuando examinas el código proporcionado, verás que no hay mucho por ahora: la clase `Piano` sólo especifica el tamaño y la resolución del mundo, y la clase `Key` (*Tecla*) contiene sólo los esqueletos de los métodos (métodos vacíos) para el constructor y para el método `act` (mostrados en Código 5.1).

Código 5.1
La clase inicial `Key`.

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

public class Key  extends Actor
{
    /**
    *  Crea una nueva tecla.
    */
    public Key()
    {
    }

    /**
    *  Realiza la acción para esta tecla.
    */
    public void act()
    {
    }
}
```

Podemos empezar a experimentar creando un objeto de la clase `Key` y colocándolo en el mundo. Verás que su imagen es una simple tecla blanca, y no hace nada en absoluto cuando ejecutamos el escenario.

Nuestra primera tarea será animar el teclado del piano: cuando pulsamos una tecla del piano, nos gustaría que la tecla del piano cambiara como si fuera pulsada. El escenario proporcionado contiene dos ficheros de imagen llamados *white-key.png* y *white-key-down.png*, que podemos usar para mostrar estos dos estados. (También contiene dos ficheros de imagen más, *black-key.png* y *black-key-down.png*, que usaremos más tarde para las teclas negras.) La imagen *white-key.png* es la que estamos ahora viendo cuando creamos una tecla.

Podemos crear el efecto de pulsar una tecla fácilmente, conmutando entre las dos imágenes cuando se pulsa una tecla del teclado. El Código 5.2 nos muestra un primer intento.

Código 5.2
Primera versión del método `act` intercambiando las imágenes

```
public void act()
{
    if ( Greenfoot.isKeyDown("g") ) {
    setImage ("white-key-down.png");
    }
    else {
    setImage ("white-key.png");
    }
}
```

En este código, hemos escogido una tecla arbitraria del teclado (la techa «g») a la que reaccionar. En este momento, no nos importa qué tecla usamos —al final sí que querremos asignar diferentes teclas del piano a nuestras teclas del teclado. Cuando pulsamos la tecla en el teclado, vemos la imagen de tecla pulsada; cuando no está pulsada, vemos la imagen de tecla no pulsada.

Ejercicio 5.3 Implementa esta versión del método `act` en tu propio escenario. Pruébalo —comprueba que funciona bien.

Aunque esta versión funciona, tiene un problema: la imagen se fija no sólo cuando cambia, sino continuamente. Cada vez que el método `act` se ejecuta, la imagen se fija a una de las dos imágenes, incluso cuando podría mostrar ya la imagen correcta. Por ejemplo, si no se está pulsando la tecla «g», la imagen será fijada a *white-key.png*, incluso si ya era ésta la imagen mostrada.

Este problema no parece muy grave. Después de todo, fijar la imagen cuando no es necesario es simplemente innecesario, pero no incorrecto. Hay varias razones por las que queremos arreglar esto, sin embargo. Una razón es que es una buena práctica no desperdiciar los recursos del procesador haciendo trabajo innecesario. Otra razón es que pronto añadiremos sonido a la tecla, y entonces sí importará. Cuando pulsamos una tecla, es bastante diferente si oímos el sonido de la tecla una vez, o una y otra vez.

Por tanto, vamos a mejorar nuestro código para asegurarnos de que la imagen sólo se fija cuando cambia. Para hacer esto, vamos a añadir un campo booleano a nuestra clase para que recuerde si la tecla está actualmente pulsada o no. Llamamos a este campo `isDown` (*estáAbajo*), y su declaración es como sigue:

```
private boolean isDown;
```

Almacenaremos `true` en este campo cuando la tecla del piano esté pulsada, y `false` cuando no lo esté.

Ahora podemos comprobar si una tecla del teclado acaba de ser pulsada. Si nuestro campo `isDown` es `false`, pero la tecla «g» de nuestro teclado se está pulsando, debe haber sido pulsada hace un rato. Al revés, si nuestro campo `isDown` vale `true` (creemos que la tecla está pulsada), pero la tecla «g» del teclado no está pulsada, entonces debemos acabar de liberarla. En estas dos situaciones, podemos cambiar entonces la imagen. El cuadro del Código 5.3 muestra el método `act` completo que implementa esta idea.

Código 5.3
Sólo fijamos la imagen cuando se tiene que cambiar

```
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;
    }
}
```

Concepto:

Los operadores lógicos, como **&&** (AND) y **!** (NOT), se pueden utilizar para combinar múltiples expresiones booleanas en una expresión booleana.

En ambos casos, no nos tenemos que olvidar de cambiar el valor del campo `isDown` al nuevo estado si detectamos un cambio.

Este código hace uso de dos nuevos símbolos: la exclamación (`!`) y el ampersand[1] (`&&`).

Ambos son operadores lógicos. El símbolo de fin de exclamación significa *NOT*, mientras que el doble ampersand significa *AND*.

Por tanto, las siguientes líneas del método `act`

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

se pueden leer un poco más informalmente (atención: ¡no es código Java!) como

```
if ( (not isDown) and Greenfoot.isKeyDown("g") ) ...
```

El mismo código, aún más informalmente, se puede leer como

```
if ( la-tecla-del-piano-no-está-ahora-pulsada
   and la-tecla-del-teclado-está-pulsada) {
   cambia la imagen mostrada a la imagen "pulsada";
   recuerda que la tecla del piano está pulsada ahora;
}
```

Mira de nuevo el Código 5.3, y comprueba que entiendes el código mostrado ahí. La lista completa de todos los operadores lógicos la puedes encontrar en el Apéndice D.

Ejercicio 5.4 Implementa la nueva versión del método `act` en tu propio escenario. Parece que hace lo mismo que antes, pero es un paso preparatorio para lo que vamos a hacer a continuación. No olvides que también tenemos que añadir el campo booleano `isDown` al principio de tu clase.

5.2 Reproduciendo el sonido

La siguiente cosa que vamos a hacer es conseguir que al presionar una tecla, suene. Para hacer esto, vamos a añadir un nuevo método a la clase `Key`, denominado `play` (*reproducir*). Podemos añadir este método en el editor, a continuación del método `act`. Para comenzar, podemos escribir el comentario, la signatura, y un cuerpo vacío para el nuevo método:

```
/**
*  Reproduce la nota de esta tecla.
*/
public void play()
{

}
```

[1] N. del T. Aunque este símbolo se lee en español como «et», se conoce por el término inglés *ampersand* de forma habitual.

Aunque este código no hace nada (el cuerpo del método está vacío), debería compilar.

La implementación de este método es bastante simple: queremos reproducir un único fichero de sonido. El escenario *piano-1* que usaste para comenzar este proyecto, incluye una colección de ficheros de sonido (en la subcarpeta *sounds*), cada uno de los cuales contiene el sonido de una tecla del piano. Los nombres de los ficheros de sonido son *2a.wav*, *2b.wav*, *2c.wav*, *2c#.wav*, *2d.wav*, *2d#.wav*, *2e.wav*, y así el resto. De éstas, cojamos una nota de forma aleatoria, — por ejemplo *3a.wav*, un *a* [2] medio—para probar nuestra nota de prueba.

Para reproducir esta nota, podemos usar de nuevo el método playSound (*reproduceSonido*) de la clase Greenfoot:

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

Éste es el único código que necesitamos en el método `play`. La implementación completa del método se muestra en el cuadro del Código 5.4.

Código 5.4
Reproduce la nota para la tecla

```
/**
 *  Reproduce la nota para esta tecla.
 */
public void play()
{
     Greenfoot.playSound("3a.wav");
}
```

Ejercicio 5.5 Implementa el método `play` en tu propia versión del escenario. Comprueba que el código compila.

Ejercicio 5.6 Prueba tu método. Puedes hacer esto creando un objeto de la clase `Key`, pincha con el botón derecho en el objeto e invoca el método `play` del menú emergente del objeto.

Casi hemos terminado ya. Podemos reproducir el sonido de la tecla invocando interactivamente al método `play`, y podemos ejecutar el escenario y presionar una tecla del teclado («g») para que parezca que la tecla del piano se está presionando.

Todo lo que necesitamos hacer ahora es reproducir un sonido cuando se pulsa una tecla del teclado.

Para tocar el sonido programáticamente (desde tu código), simplemente llamamos a nuestro propio método `play`, así:

```
play();
```

[2] N. del T. Se usa la notación inglesa para las notas (a, b, c, d, …) en vez de la notación del solfeo (la, si, do, re, …), y el número indica la escala.

Ejercicio 5.7 Añade código a tu clase `Key` para que la nota de la tecla se toque cuando se presiona la tecla asociada del teclado. Para hacer esto, necesitas averiguar dónde se debe añadir la llamada al método `play`. Pruébalo.

Ejercicio 5.8 ¿Qué pasa cuando creas dos teclas, ejecutas el escenario, y pulsas la tecla «g»? ¿Se te ocurre lo que necesitamos hacer para reaccionar a teclas diferentes del teclado?

Todos los cambios descritos hasta ahora están disponibles en el escenario *piano-2* de los escenarios del libro. Si has encontrado problemas que no pudiste resolver, o si simplemente quieres comparar tu solución con la nuestra, mira esta versión.

5.3 Abstracción: creando muchas teclas

Hemos alcanzado un punto en el que podemos crear un piano que reacciona a una tecla de nuestro ordenador y toca una nota. El problema ahora es obvio: cuando creamos varias teclas, todas reaccionan a la misma tecla del teclado, y todas producen la misma nota. Tenemos que cambiar esto.

La limitación actual viene del hecho de que hemos escrito a fuego (*hard-coded*) el nombre de la tecla del teclado («g») y el nombre del fichero de sonido («3a.wav») en nuestra clase. Esto significa, que hemos usado los nombres directamente, sin dejar la opción de cambiarlos a menos que cambiemos el código fuente y recompilemos.

Concepto:

La abstracción sucede de muchas formas diferentes en programación. Una de ellas es la técnica para escribir programas que puedan resolver una clase entera de problemas en vez de un único problema específico.

Cuando hacemos un programa, escribir código que puede resolver una tarea muy específica —como calcular la raíz cuadrada de 1,764 o reproducir el sonido de la tecla la-media de un piano— está bien y es bueno, pero no es increíblemente útil. Generalmente, querríamos escribir código que resuelva una *clase* entera de problemas (tales como calcular la raíz cuadrada de cualquier número, o tocar todos los sonidos del piano). Si hacemos esto, nuestro programa es mucho más útil.

Para conseguir esto, usamos una técnica denominada *abstracción*. La abstracción sucede en computación en muchas formas diferentes y contextos —éste es uno de ellos.

Usaremos abstracción para transformar nuestra clase `Key` que puede crear objetos que reproducen la nota a-media cuando se pulsa la tecla «g» del teclado, en una clase que pueda crear objetos que tocan un abanico de notas cuando se presionan diferentes teclas del teclado.

La principal idea es conseguir esto usando una variable para el nombre de la tecla del teclado a la que queremos reaccionar, y otra variable para el nombre del fichero de sonido que queremos tocar.

El cuadro del Código 5.5 muestra el comienzo de una solución para esto. Se usan dos campos adicionales —`key` (*tecla*) y `sound` (*sonido*)— para almacenar el nombre de la tecla y el fichero de sonido que queremos usar. También añadimos dos parámetros al constructor, de forma que pasemos esta información cuando se crea el objeto tecla, y asegurarnos de que almacenamos estos valores de los parámetros en los campos en el cuerpo del constructor.

Hemos hecho una abstracción de nuestra clase `Key`. Ahora, cuando creamos un nuevo objeto `Key`, podemos especificar a qué tecla de teclado debe reaccionar, y qué fichero de sonido debe reproducir. Por supuesto, no hemos escrito aún el código que usa estas variables —esto está aún por hacer.

Código 5.5
Generalizando para muchas teclas: definiendo las variables `key` y `sound`

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

     /**
       *  Crea una nueva tecla que se corresponde con una tecla del
       *  teclado, y con un sonido dado
       */
     public Key(String keyName, String soundFile)
     {
          key  = keyName;
          sound = soundFile;
     }

     // métodos omitidos.
}
```

Te dejamos esto como un ejercicio.

Ejercicio 5.9 Implementa los cambios descritos arriba. Esto es, añade los campos para la tecla y el fichero de sonido, y añade un constructor con dos parámetros que inicialice estos campos.

Ejercicio 5.10 Modifica tu código para que tu objeto tecla reaccione a la tecla especificada y reproduzca el fichero de sonido especificado en el constructor. ¡Pruébalo! (Construye varias teclas con diferentes sonidos.)

Estamos ahora en un punto en el que podemos crear un conjunto de teclas para tocar un rango de notas. (Actualmente, sólo tenemos teclas blancas, pero podemos construir ya la mitad de un teclado con esto.) Esta versión del proyecto está en el escenario *piano-3* de los escenarios del libro.

Construir todas las teclas, sin embargo, es un poco tedioso. Actualmente, tenemos que crear cada tecla del piano a mano, escribiendo todos los parámetros. Lo que es peor: cada vez que hacemos un cambio en el código fuente, tenemos que volver a empezar. Es hora de escribir algún código que cree las teclas para nosotros.

5.4 Construyendo el piano

Nos gustaría escribir algunas líneas de código en la clase `Piano` que creen y coloquen las teclas del piano para nosotros. Añadir una sola tecla (o unas pocas teclas) es bastante sencillo: añadimos la siguiente línea al constructor de `Piano`, y una tecla se crea y se coloca en el mundo cada vez que reiniciamos el escenario:

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

Recuerda que la expresión

```
new Key (“g”, “3a.wav”)
```

crea un nuevo objeto tecla (especificando una tecla y un sonido de fichero), mientras que la sentencia

```
addObject ( algún-objeto, 300, 180);
```

inserta el objeto dado en el mundo en las coordenadas *x* e *y* especificadas. Las coordenadas exactas 300 y 180 se han tomado de forma arbitraria en este punto.

Ejercicio 5.11 Añade código a tu clase `Piano` para que cree automáticamente las teclas del piano y las coloque en el mundo.

Ejercicio 5.12 Cambia la coordenada y en la que la tecla se coloca, de forma que la tecla del piano aparezca exactamente en la parte superior del piano (es decir, la parte superior de la tecla del piano debería estar alineada con la parte superior del piano). Pista: la imagen de la tecla tiene 280 pixels de alto y 63 pixels de ancho.

Ejercicio 5.13 Escribe código para crear una segunda tecla del piano que toque la nota g-media (fichero de sonido 3g.wav) cuando la tecla «f» sea pulsada en el teclado. Coloca esta tecla exactamente a la izquierda de la primera tecla (sin ningún espacio ni superposición).

Hemos visto antes en el libro que es bueno usar métodos separados para tareas separadas. Crear todas las teclas es una tarea distinta desde el punto de vista lógico, por lo que colocaremos este código en un método separado. Hará exactamente lo mismo, pero el código será más fácil de leer.

Ejercicio 5.14 Crea un nuevo método llamado `makeKeys()` en la clase `Piano`. Mueve tu código que crea las teclas a este método. Llama a este método desde el constructor de Piano. Asegúrate de escribir un comentario para tu nuevo método.

Ahora podemos avanzar e insertar una lista entera de sentencias `addObject` para crear todas las teclas que necesitamos para nuestro teclado. Esta no es, sin embargo, la mejor manera de hacer lo que queremos.

5.5 Usando bucles: el bucle while

Los lenguajes de programación ofrecen un constructor específico para hacer una tarea similar muchas veces: un *bucle*.

Un bucle es un constructor de un lenguaje de programación que nos permite expresar órdenes como: *«Haz esta sentencia 20 veces»* o *«Llama a estos dos métodos 3 millones de veces»* de forma fácil y

Concepto:

Un **bucle** es una sentencia en un lenguaje de programación que permite ejecutar una sección de código muchas veces.

concisa (sin escribir 3 millones de líneas de código). Java tiene varios tipos de bucles. Vamos a ver ahora el llamado *bucle while*.

Un bucle `while` sigue la siguiente forma:

```
while ( condición )
{
     sentencia;
     sentencia;
     ...
}
```

La palabra clave `while` de Java está seguida de una condición entre paréntesis y de un bloque (un par de llaves) conteniendo una o más sentencias. Estas sentencias serán ejecutadas una y otra vez, hasta que la condición sea cierta (true).

Un patrón muy común de un bucle es ejecutar unas sentencias un número dado de veces. Para hacer esto, usamos una *variable del bucle* como contador. Es una práctica común nombrar a las variables de bucle `i`, así que lo haremos así. Aquí sigue un ejemplo que ejecuta el cuerpo del bucle `while` 100 veces:

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

Concepto:

Una **variable local** es una variable que se declara dentro del cuerpo de un método. Se usa para almacenamiento temporal.

Hay varias cosas que podemos destacar de este código. Primero, usa un concepto que no hemos visto antes: una *variable local*.

Una variable local es una variable similar a un campo. Podemos usarla para almacenar valores en ella, tales como un número entero, o referencias a objetos.

Difiere de los campos en varios aspectos:

- Una variable local se declara dentro del cuerpo de un método, no al principio de la clase;
- No tiene ningún modificador de visibilidad (private o public) delante; y
- Sólo existe hasta que el método termina su ejecución, y es borrada cuando termina [3].

Una variable local se declara escribiendo simplemente el tipo de la variable, seguido de su nombre:

```
int i;
```

Después de declarar la variable, podemos asignarle un valor. Aquí están las dos sentencias juntas:

```
int i;
i = 0;
```

[3] Estrictamente hablando, esto no es del todo correcto. Las variables locales también se pueden declarar en otros bloques, tales como dentro de un bloque de una sentencia if o en el cuerpo de un bucle. Existen sólo hasta que la ejecución sale del bloque donde fueron declaradas. La sentencia de arriba es, sin embargo, correcta si la variable local se declara al principio de un método.

Java permite un atajo para escribir estas dos sentencias en una línea, declarando la variable y asignándole un valor:

```
int i = 0;
```

Esta línea tiene exactamente el mismo efecto que la versión en dos líneas. Ésta es la variante que hemos usado antes en el patrón de código para el bucle `while`.

Vuelve a mirar al patrón del bucle —ahora deberíamos entender por encima lo que hace. Usamos una variable `i` y la inicializamos a 0. Entonces ejecutamos repetidamente el cuerpo del bucle `while`, incrementando la `i` en cada repetición. Continuamos haciendo esto mientras `i` es menor que 100. Cuando alcanzamos 100, paramos el bucle. La ejecución continúa entonces con el código que sigue al cuerpo del bucle.

Hay dos detalles más que podemos destacar:

- Usamos la sentencia `i = i + 1;` al final del cuerpo del bucle para incrementar nuestra variable de bucle en una unidad cada vez que ejecutamos el bucle. Esto es importante. Un error común es olvidarse de incrementar el contador del bucle. La variable no cambiaría nunca entonces, la condición será siempre cierta, y el bucle continuaría siempre. Esto se llama un *bucle infinito*, y es la causa de muchos errores en los programas.
- Nuestra condición dice que ejecutamos el bucle `while` cuando `i` es menor que (<) 100, no cuando es menor o igual (<=). Por tanto, el bucle no se ejecutará cuando `i` es igual a 100. A primera vista, uno puede pensar que esto significa que el bucle se ejecuta sólo 99 veces, no 100 veces. Pero esto no es así. Como hemos comenzado en 0, y no en 1, se ejecuta 100 veces (contando de 0 a 99). Es muy común empezar a contar desde 0 en programas de ordenador —veremos pronto algunas ventajas de hacerlo así.

Ahora que conocemos el bucle `while`, vamos a usar este constructor para crear todas las teclas de nuestro piano. Nuestro piano tiene 12 teclas blancas. Podemos crear ahora las 12 teclas poniendo nuestra sentencia de creación de una tecla dentro de un bucle que se ejecute 12 veces:

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

Ejercicio 5.15 Reemplaza el código de tu propio método `makeKeys` con el bucle mostrado aquí. Pruébalo. ¿Qué observas?

Analizando este código, primero comprueba si una tecla se ha creado. Esto es engañoso, sin embargo. Tenemos realmente 12 teclas, pero como todas han sido insertadas exactamente en las mismas coordenadas, todas yacen una encima de la otra, y no podemos verlas muy bien. Intenta mover las teclas en el mundo del piano con tu ratón, y podrás ver que están todas allí.

Ejercicio 5.16 ¿Cómo puedes cambiar tu código para que no aparezcan todas las teclas en el mismo lugar? ¿Puedes cambiar tu código para que se coloquen una junto a otra?

La razón de que todas las teclas aparezcan una encima de la otra es que hemos insertado todas en la misma posición fija 300,140 del mundo. Ahora queremos insertar cada tecla en una posición diferente. Esto es bastante fácil de hacer: podemos hacer uso de nuestra variable de bucle `i` para conseguirlo.

Ejercicio 5.17 ¿Cuántas veces se ejecuta el cuerpo de nuestro bucle? ¿Cuáles son los valores de i durante cada ejecución?

Podemos reemplazar ahora la coordenada *x* fija 300 con una expresión que incluya la variable `i`:

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

(El asterisco «*» es el operador para la multiplicación. El Apéndice D lista otros operadores que puedes usar con números enteros.)

Hemos escogido `i*63`, porque sabemos que la imagen de cada tecla tiene 63 pixels de ancho. Los valores de `i`, cuando se ejecuta el bucle, son 0, 1, 2, 3, y sigue así. Por tanto, las teclas serán colocadas en las coordenadas *x* 0, 63, 126, 189, y seguiría así.

Cuando intentamos esto, nos damos cuenta de que la tecla más a la izquierda no está muy bien colocada. Como la colocación de objetos en Greenfoot usa el centro de un objeto, el centro de la primera tecla se pone en la coordenada *x* 0, lo que coloca la tecla con la mitad fuera de la pantalla. Para arreglar esto, simplemente añadimos un desplazamiento fijo a cada coordenada de una tecla. El desplazamiento se escoge de forma que el conjunto de todas las teclas aparezca en el medio de nuestro piano:

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

La coordenada *y* puede permanecer constante, ya que queremos que todas las teclas tengan la misma altura.

Nuestro código ahora coloca bien nuestras teclas blancas —esto es un buen paso hacia delante. El problema más obvio ahora es que todas las teclas del piano reaccionan a la misma tecla del piano y reproducen la misma nota. Para arreglar esto de nuevo requiere que veamos un nuevo constructor de programación: un *array*.

Ejercicio 5.18 Ejercicio de reto. (Haz este ejercicio sólo si te sientes confiado en tu programación. Si estás comenzando, puedes saltártelo.)

Normalmente no es la mejor solución usar números fijos en tu código, tales como 140 o 63 de la sentencia anterior, ya que hace tu código vulnerable a fallos cuando las cosas cambian. Por ejemplo, si reemplazamos las imágenes del teclado con unas imágenes más bonitas que tengan un tamaño diferente, nuestro código no las colocará correctamente.

Podemos evitar usar números directamente y usar en cambio los métodos `getWidth()` *(obténAncho)* y `getHeight()` *(obténAlto)* de la imagen de la tecla. Para hacer esto, asignamos el objeto tecla a una variable local de tipo `Key` cuando la creamos, y usamos entonces `key.getImage().getWidth()` en lugar de 63. Haz una cosa similar con la altura.

Para reemplazar el 54, es necesario que uses el método `getWidth()` de la imagen del piano después de hacer esto, tu código colocará siempre las teclas bien, incluso si su tamaño cambia.

5.6 Usando arrays

Actualmente, nuestras 12 teclas se crean y se colocan en las posiciones apropiadas de la pantalla, pero todas ellas reaccionan a la tecla «g», y todas tocan la misma nota. Esto sucede a pesar de que hemos preparado nuestras teclas para que acepten diferentes teclas del teclado y diferentes ficheros de sonido en el constructor. Sin embargo, todas nuestras teclas se han creado en la misma linea de código (ejecutada en un bucle), por lo que todas ellas se crean con los parámetros «g» y «3a.wav».

La solución es similar al cambio que hicimos para la coordenada *x*: deberíamos usar variables para la tecla del teclado y el fichero de sonido, y luego asignarles valores diferentes en cada ejecución del bucle.

Concepto:

Un **array** es un objeto que guarda muchas variables. Se puede acceder a estas variables usando un **índice**.

Sin embargo, esto es más problemático que en el caso de la coordenada *x*. Las teclas correctas y los nombres de los ficheros de sonido no se pueden calcular tan fácilmente. ¿De dónde sacamos los valores?

Nuestra respuesta es: los almacenaremos en un array.

Un array es un objeto que puede guardar muchas variables, y por tanto, puede almacenar muchos valores. Podemos ver esto en un diagrama. Supón que tenemos una variable llamada «nombre» de tipo `String`. Y que asignamos el `String` «Pepe» a esta variable:

```
String nombre;
name = “Pepe”;
```

La Figura 5.2 ilustra este ejemplo.

Este caso es muy simple. La variable es un contenedor que puede guardar un valor. El valor se almacena en la variable.

En el caso de un array, podemos obtener objetos separados —el objeto array— que guarda muchas variables. Podemos entonces almacenar una referencia a un objeto arrray en nuestra propia variable (Figura 5.3).

Figura 5.2
Una simple variable `String`

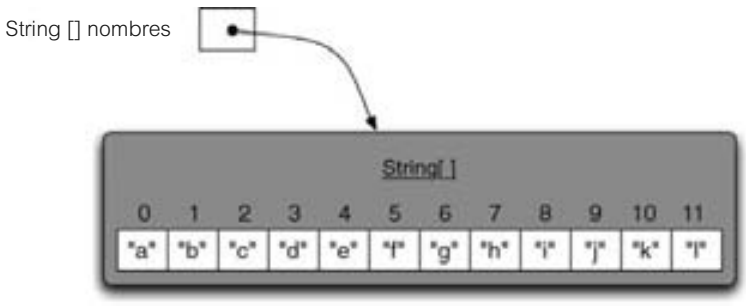


Figura 5.3
Un array de Strings

El código Java para crear esta situación es:

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

En la declaración de la variable, el par de corchetes (`[]`) indica que el tipo de la variable es un array. La palabra delante de las corchetes indica el *tipo de elemento* del array, esto es, el tipo que cada entrada debería tener. Por tanto `String[]` representa un array de Strings, mientras que `int[]` representa un array de enteros (integers).

La expresión

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

crea el objeto array y lo rellena con los Strings de la «a» a la «l». Este objeto array se asigna entonces a nuestra variable nombres. Podemos ver en el diagrama que, cuando se asigna un objeto array a una variable, la variable contiene entonces un puntero a ese objeto.

Una vez que tenemos nuestra variable array rellena, podemos acceder a los elementos individuales del array usando un *índice* —un número con la posición en el objeto array. En la Figura 5.3, se muestra el índice encima de cada elemento del array. Observa que de nuevo comenzamos contando en 0, por lo que el String «a» está en la posición 0, «b» está en la posición 1, y así sucesivamente.

Concepto:

Se accede a los **elementos** individuales de un array usando corchetes (**`[]`**) y un índice para especificar el elemento del array.

En Java, accedemos a los elementos del array añadiendo el índice entre corchetes al nombre del array. Por ejemplo,

```
nombres[3]
```

accede al elemento del array con índice 3 —el String «d».

Podemos preparar ahora dos arrays para nuestro proyecto del piano: uno con los nombres de las teclas del teclado (en orden) para nuestras teclas del piano, y otro con los nombres de los ficheros de sonido para esas teclas del piano. Podemos declarar campos en la clase `Piano` para estos array y almacenar los arrays rellenos. El cuadro del Código 5.6 lo ilustra.

Observa que los valores del array `whiteKeys` (*teclasBlancas*) son las teclas de la fila media de mi teclado del ordenador. Los teclados son un poco diferentes en sistemas diferentes y en países diferentes, por lo que puede que tengas que cambiar las teclas para que coincidan con tu teclado. La otra cosa un poco rara aquí es el String «\\». El carácter de barra inversa[4] (\) se llama un *carácter de escape* y tiene un significado especial en los Strings de Java. Para crear uno que contenga el carácter de barra inversa como un carácter normal, tienes que escribirlo dos veces. Así, escribir el String «\\» en tu código fuente Java, crea en realidad el String «\».

Código 5.6
Creando arrays para teclas y notas

```
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” };

    // constructor y métodos omitidos.

}
```

[4] N. del T. Es frecuente denominar este carácter con su término inglés *backslash*.

Ahora tenemos arrays que listan las teclas y los nombres de los ficheros de sonido que queremos usar para nuestras teclas del piano. Podemos adaptar nuestro bucle en el método `makeKeys` para hacer uso de los elementos del array y crear las teclas apropiadas. El cuadro del Código 5.7 muestra el código fuente resultante.

Código 5.7
Crea las teclas del piano con las teclas del teclado y las notas de los arrays

```
/**
*  Crea las teclas del piano y las coloca en el mundo.
*/
private void makeKeys()
{
    int i = 0;
    while (i < whiteKeys.length)
    {
        Key  key = new Key(whiteKeys[i], whiteNotes[i] + ".wav");
        addObject(key, 54  + (i*63), 140);
        i = i + 1;
    }
}
```

Podemos mencionar unas cuantas cosas:

- Hemos sacado la creación de nuevas teclas de la llamada al método `addObject` y lo hemos puesto en una línea separada, y hemos asignado el objeto tecla creado inicialmente a una variable local, denominada `key`. Esto se ha hecho por claridad: la línea de código empezaba a ser muy larga y estar muy cargada, y era difícil de leer. Al dividirla en dos pasos, es más fácil de leer.
- Los parámetros para el constructor de Key acceden a `whiteKeys[i]` y `whiteNotes[i]`. Esto es, usamos nuestra variable de bucle `i` como el índice del array para acceder por turno a todos los diferentes strings de teclas y nombres de ficheros de sonidos.

Concepto:

Cuando se usa el símbolo más (+), con Strings, realiza **concatenación de Strings**. Junta dos Strings en uno.

- Usamos el símbolo más (+) con `whiteKeys[i]` y un String («.wav»). La variable `whiteKeys[i]` también es un String, por lo que el símbolo más se está usando con dos operandos String. Cuando + se usa con Strings, realiza *concatenación de strings*. La concatenación de strings es una operación que pega juntos dos Strings y los convierte en un único String. En otras palabras, aquí, agregamos el String «.wav» al valor de `whiteNotes[i]`. Es así porque el nombre almacenado en el array tiene la forma «3c», y el nombre del fichero era «3c.wav». Podríamos haber almacenado el nombre completo del fichero en el array, pero como la extensión es la misma para todos los ficheros de notas, parecía innecesario. Simplemente basta con añadirlo ahora y nos ahorramos escribirlo en cada uno.
- Hemos reemplazado también el 12 de la condición del bucle `while` con `whiteKeys.length`. El atributo `.length` de un array devuelve el número de elementos de dicho array. En nuestro caso, tenemos 12 elementos, así que si hubiéramos dejado el 12, hubiera funcionado. Sin embargo, es más seguro usar el atributo length. Así, si decidimos más adelante tener más o menos teclas, nuestro bucle seguirá funcionando bien, sin tener que cambiar la condición.

Con estos cambios, nuestro piano debería ser jugable con la fila media de teclas de nuestro teclado, y debería producir notas diferentes para teclas diferentes.

Ejercicio 5.19 Haz los cambios descritos arriba en tu propio escenario. Asegúrate de que todas las teclas funcionan. Si tu teclado tiene una disposición diferente, adapta el array `whiteKeys` a tu teclado.

Ejercicio 5.20 La carpeta *sounds* del escenario del piano contiene más notas de la que estamos usando ahora. Cambia el piano de forma que las teclas sean una octava más baja de lo que son ahora. Esto es, usa el sonido «2c» en vez de «3c» para la primera tecla, y sigue así desde aquí.

Ejercicio 5.21 Si te apetece, puedes hacer que tus teclas produzcan sonidos completamente diferentes. Puedes grabar tus propios sonidos usando un software de grabación de sonidos, o puedes buscar archivos de sonido en Internet. Mueve los ficheros de sonido a la carpeta *sounds*, y haz que tus teclas los reproduzcan.

La versión que tenemos ahora es la versión *piano-4* de los escenarios del libro.

La parte que falta es bastante obvia: tenemos que añadir las teclas negras.

No hay nada realmente nuevo en esto. Esencialmente tenemos que hacer cosas muy similares de nuevo a las que hicimos para las teclas blancas. Te dejamos esto como un ejercicio. Sin embargo, hacer todo de golpe te llevará un buen rato. En general, cuando tengas una tarea más larga, es bueno dividirla en pasos más pequeños. Por tanto, dividiremos esta tarea en una secuencia de ejercicios que te conduzcan a la solución paso a paso.

Ejercicio 5.22 Actualmente, nuestra clase `Key` puede reproducir sólo notas blancas. Esto es porque hemos puesto a fuego los nombres de los ficheros de las imágenes de las teclas («white-key.png» y «white- key-down.png»). Usa abstracción para modificar la clase `Key` de forma que pueda mostrar tanto las teclas blancas con las negras. Se hace de forma similar a lo que hicimos con el nombre de la tecla y el nombre del fichero de sonido. Introduce dos campos y dos parámetros para los dos nombres de los ficheros de las imágenes, y luego usa las variables en vez de usar directamente los nombres de los ficheros. Pruébalo creando algunas teclas blancas y negras.

Ejercicio 5.23 Modifica tu clase `Piano` para que añada dos teclas negras en una posición arbitraria.

Ejercicio 5.24 Añade dos arrays más a la clase `Piano` para las teclas del teclado y las notas de las teclas negras.

Ejercicio 5.25 Añade otro bucle en el método `makeKeys` de la clase `Piano` que crea y coloca las teclas blancas. Esto es hacer un poco de trampa, por el hecho de que las teclas negras no se colocan alineadas con las teclas blancas —hay saltos (mira la Figura 5.1). ¿Puedes encontrar una solución a esto? Pista: Crea una entrada especial en tu array donde están los saltos, y úsalo para reconocer los saltos. (Lee la nota a continuación de los ejercicios antes de comenzar. ¡Esto es una tarea dura! Puedes querer mirar la solución de *piano-5* si no se te ocurre.)

Ejercicio 5.26 La implementación completa de este proyecto, piano-5, también incluye un método corto para mostrar una línea de texto en la pantalla. Estúdialo, y haz algunos cambios. Cambia las palabras del texto, cambia su color, y mueve el texto para que esté centrado horizontalmente.

Concepto:

El tipo **String** se define con una clase normal. Tiene muchos métodos útiles, que podemos mirar en la documentación de la librería Java.

Nota: La clase String

El tipo `String` que hemos usado varias veces antes de definir la clase. Busca esta clase en la documentación de la librería Java, y mira sus métodos. Hay muchos, algunos de los cuales son muy útiles a menudo.

Verás métodos para crear subcadenas (substrings), para averiguar la longitud de una cadena, para convertir entre mayúsculas y minúsculas, y muchos más.

Especialmente interesante para el Ejercicio 5.25 puede ser el método `equals` que permite comparar un string con otro string, y devuelve `true` si los dos strings son iguales.

Hasta aquí vamos a avanzar con este proyecto. El piano está más o menos completo ahora. Podemos tocar melodías simples, e incluso podemos tocar acordes (múltiples teclas a la vez).

Siéntete libre de extenderlo como quieras. ¿Y si añades un segundo conjunto de sonidos, y luego añades un conmutador en la pantalla que te permita conmutar de los sonidos del piano a tus sonidos alternativos?

5.7 Resumen de técnicas de programación

En este capítulo, hemos visto dos conceptos muy importantes y fundamentales de una programación más sofisticada: bucles y arrays. Los bucles nos permiten escribir código que ejecuta un bloque de sentencias muchas veces. El constructor de bucles que hemos visto se llama un *bucle while*. Java tiene otros tipos de bucles que veremos muy pronto. Usaremos bucles en muchos de nuestros programas, por lo que es esencial entenderlos.

Dentro de un bucle, solemos usar el contador del bucle para hacer cálculos o para generar diferentes valores en cada iteración del bucle.

El otro nuevo concepto importante que hemos visto es un array. Un array puede proporcionar muchas variables (todas del mismo tipo) en un único objeto. A menudo, se usan bucles para procesar un array si necesitamos hacer algo con cada uno de sus elementos. Se accede a los elementos con corchetes.

Otra técnica fundamental que hemos visto es la abstracción. En este caso, este concepto ha aparecido con el uso de los parámetros del constructor para crear el código que pudiera gestionar un tipo de problemas en vez de un problema específico.

También nos hemos encontrado con unos cuantos operadores nuevos: hemos visto los operadores lógicos AND y NOT para expresiones booleanas (`&&` y `!`), y hemos visto que el operador más (+) realiza concatenación cuando se aplica a Strings. La clase `String` se documenta en la documentación de la API de Java y tiene muchos métodos útiles.

Resumen de conceptos

- Los **operadores lógicos**, tales como && (AND) y ! (NOT), pueden usarse para combinar múltiples expresiones booleanas en una.
- La **abstracción** sucede de muchas formas en la programación. Una de ellas es la técnica para escribir código que resuelva un tipo de problemas, en vez de un problema específico.

- Un **bucle** es una sentencia en un lenguaje de programación que ejecuta un bloque de código muchas veces.
- Una **variable local** es una variable que se declara dentro del cuerpo de un método. Se usa para almacenamiento temporal.
- Un **array** es un objeto que guarda muchas variables. Podemos acceder a cada una usando un **índice**.
- Se accede a los **elementos** individuales de un array usando corchetes (**[ ]**) y un índice para especificar el elemento del array.
- Cuando se usa el símbolo más (+), con Strings, significa **concatenación de Strings**. Junta dos strings en uno.
- El tipo **String** se define con una clase normal. Tiene muchos métodos útiles, que se pueden consultar en la documentación de la librería Java.

Términos en inglés

Inglés	Español
index	índice
infinite loop	bucle infinito
loop	bucle
method stub	esqueleto de método

CAPÍTULO

6

Objetos interactuando: el laboratorio de Newton

temas:	interacción entre objetos, usando objetos de soporte, usando clases de la librería Java
conceptos:	colección, lista, bucle *for-each*, librería estándar de clases

En este capítulo, investigaremos interacciones más sofisticadas entre objetos en un mundo. Para comenzar, investigaremos una de las interacciones más universales entre objetos en cualquier sitio: la gravedad.

En este escenario, estamos tratando con cuerpos celestes (tales como estrellas y planetas). Simularemos el movimiento de estos cuerpos en el espacio, usando la ley universal de la gravitación de Newton. (Ahora sabemos que las fórmulas de Newton no son suficientemente precisas, y que la teoría de Einstein de la relatividad general describe los movimientos de los planetas con más precisión, pero Newton es aún suficientemente bueno para nuestra simulación simple. Ambos se presentan en la Figura 6.1.)

Si estás un poco preocupado de trabajar con física y fórmulas, no te preocupes. No necesitarás profundizar mucho, y la fórmula que usaremos es bastante simple. Al final, convertiremos este escenario en un experimento artístico con sonido y efectos visuales. Si estás más interesado en la parte técnica, puedes trabajar más en los aspectos físicos. Si en cambio tu interés es más artístico, concéntrate en este aspecto entonces.

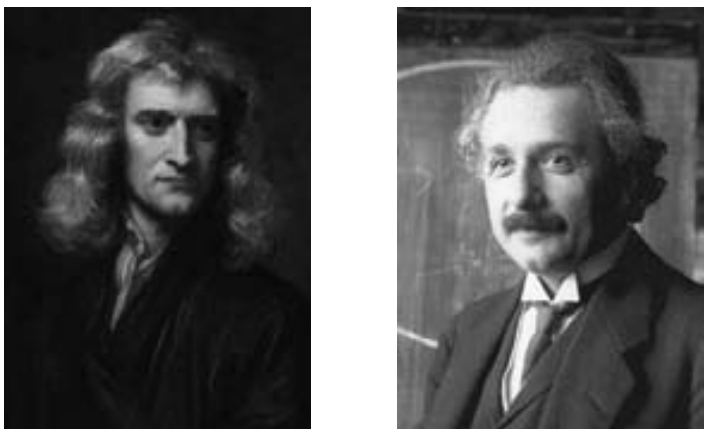


Figura 6.1
Isaac Newton y Albert Einstein

Newton: Retrato por Godfrey Kneller, 1689. Einstein: Retrato por Ferdinand Schmutzer, 1921.

6.1 El punto de partida: el laboratorio de Newton

Comenzaremos este proyecto investigando una versión parcialmente implementada de este escenario. Abre el escenario *Newtons-Lab-1* de la carpeta *book-scenarios*. Verás que ya existe una subclase de mundo (denominada `Space`). También tenemos las clases `SmoothMover`, `Body`, y `Vector` (Figura 6.2).

Ejercicio 6.1 Abre el escenario *Newtons-Lab-1*. Pruébalo (p. ej., coloca algunos cuerpos en el espacio). ¿Qué observas?

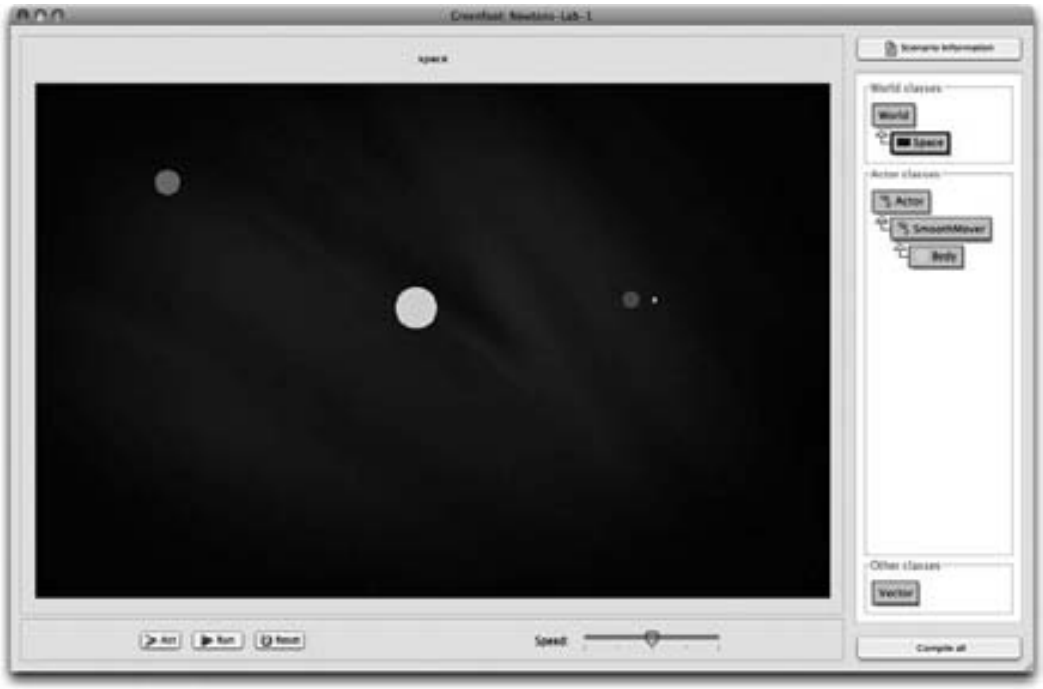


Figura 6.2
El escenario del laboratorio de Newton

Cuando intentas ejecutar este escenario, notarás que puedes colocar objetos (de tipo `Body`) en el espacio, pero estos cuerpos no se mueven, y no actúan de una forma interesante aún.

Antes de que empecemos a extender la implementación, investiguemos el escenario un poco más.

Pinchando con el botón derecho en el título del mundo (la palabra «space» justo arriba), podemos ver e invocar los métodos públicos de la clase `Space` (Espacio), (Figura 6.3).

6.2 Clases auxiliares: SmoothMover y Vector

En este escenario, usamos dos clases auxiliares (*helper classes*) de propósito general: `SmoothMover` (*MovedorSuave*) y `Vector`. Estas clases añaden funcionalidad a un escenario dado, y pueden ser

usadas en diferentes escenarios para propósitos similares. (Estas dos clases de hecho se usan en un buen número de proyectos existentes.)

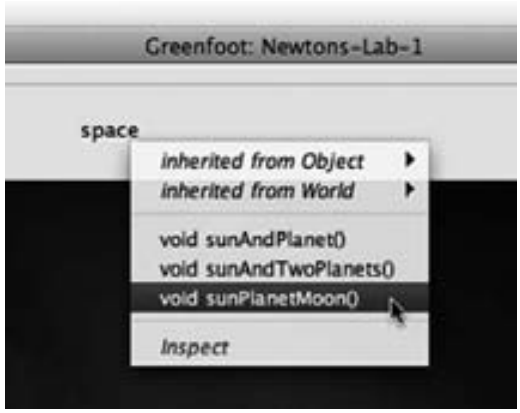


Figura 6.3
Los métodos de World en el laboratorio de Newton

Ejercicio 6.2 Invoca los diferentes métodos públicos del objeto `Space`. ¿Qué hacen?

Ejercicio 6.3 Cuando tienes una estrella o un planeta en tu mundo, pincha con el botón derecho para ver qué métodos tienen. ¿Cuáles son?

Ejercicio 6.4 Invoca el método `sunPlanetMoon` (`solPlanetaLuna`) que es uno de los métodos públicos de `Space`. Averigua y apunta la masa del Sol, el planeta, y la Luna.

Ejercicio 6.5 Mira el código fuente de la clase `Space` y mira cómo están implementados los métodos públicos.

La clase `SmoothMover` proporciona un movimiento más suave para los actores, porque almacena las coordenadas de los actores como números decimales (de tipo `double`), en vez de como números enteros. Los campos de tipo `double` pueden almacenar números con fracciones decimales (tales como 2.4567), y, por tanto, son más precisos que los enteros.

Para mostrar el actor en pantalla, las coordenadas deberán ser redondeadas a enteros, ya que la posición para pintar en la pantalla debe ser siempre un píxel entero. Internamente, sin embargo, la posición se guarda como un número decimal.

Un `SmoothMover` tiene, por ejemplo, la coordenada *x* 12.3. Si ahora movemos este actor incrementando la coordenada *x* en incrementos de 0.6, sus posiciones sucesivas serán

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

y así. Veremos el actor en la pantalla con las coordenadas *x* redondeadas. Se pintará en las siguientes coordenadas *x*

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

y así. Todo junto, incluso aunque aún se redondean a enteros para mostrarlos, el efecto es un movimiento más suave que si tratamos exclusivamente con campos `int`.

La segunda funcionalidad que añade `SmoothMover` es un vector de movimiento. Cada objeto de una subclase de `SmoothMover` mantiene un vector que indica la dirección actual y velocidad de movimiento. Podemos pensar en un vector como una flecha (invisible) con una dirección y una longitud dadas (Figura 6.4).

La clase `SmoothMover` tiene métodos para cambiar su movimiento modificando su vector de movimiento y un método `move` que mueve el actor según su vector actual.

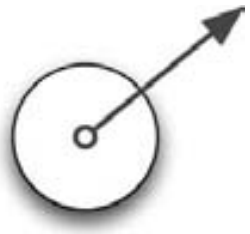


Figura 6.4
Un objeto **`SmoothMover`** con un vector de movimiento

Nota al margen: Clases abstractas

Si pinchas con el botón derecho en la clase `SmoothMover`, notarás que no puedes crear objetos de esta clase. No se muestra ningún constructor.

Cuando examinamos el código fuente de esta clase, vemos la palabra a clave `abstract` en la cabecera de la clase. Podemos declarar que las clases sean abstractas para evitar la creación de instancias de estas clases. Las clases abstractas sólo nos sirven como superclases de otras clases; y no podemos crear objetos directamente.

Ejercicio 6.6 Coloca un objeto de la clase `Body` en el mundo. Examinando el menú emergente del objeto, encuentra qué métodos ha heredado este objeto de la clase `SmoothMover`. Anótalos.

Ejercicio 6.7 ¿Qué nombre de un método aparece dos veces? ¿Cómo se diferencian las dos versiones?

Concepto:

Sobrecarga es el uso del mismo nombre de método para dos métodos o constructores diferentes.

Terminología: Sobrecarga

En Java, es perfectamente legal tener dos métodos con el mismo nombre, siempre que sus listas de parámetros sean diferentes. Esto se denomina **sobrecarga**. (El nombre del método está sobrecargado —se refiere a más de un método.)

Cuando llamamos a un método sobrecargado, el sistema en tiempo de ejecución debe averiguar a cuál de los dos métodos estamos llamando, para lo que mira los parámetros que suministramos.

También se dice que los dos métodos tienen diferentes *signaturas*.

La segunda clase auxiliar, `Vector`, implementa el vector en sí, y es usada por la clase `SmoothMover`. Observa que `Vector` no está lista en el grupo de clases `Actor`. No es un actor — nunca aparecerá en el mundo ella misma. Los objetos de esta clase sólo son creados y usados por otros objetos actor.

Los vectores pueden ser representados de dos formas diferentes: o como un par de distancias con sus coordenadas *x* e *y* (*dx*, *dy*), o como un par de valores especificando su dirección y longitud (dirección, longitud). La dirección normalmente se especifica como el ángulo con la horizontal.

La Figura 6.5 muestra el mismo vector con las dos especificaciones posibles. Vemos que tanto el par (*dx*, *dy*) como el par (dirección, longitud) pueden describir el mismo vector.

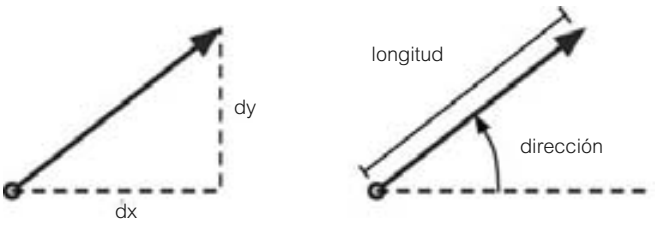


Figura 6.5
Dos formas posibles de especificar un vector

La primera representación, usando los desplazamientos *x* e *y*, se denomina una representación *cartesiana*. La segunda, usando la dirección y la longitud, se denomina una representación *polar*. Verás estos dos nombres usados en el código fuente de la clase `Vector`.

Para nuestro propósito, a veces es más fácil usar la representación cartesiana, y otras veces es más fácil usar la representación polar. Por tanto, nuestra clase vector se ha escrito de forma que podamos usar ambas. La clase realiza las conversiones necesarias internamente de forma automática.

Ejercicio 6.8 Familiarízate con los métodos de las clases `SmoothMover` y `Vector`, para lo que debes abrir el editor y estudiar su definición en la vista de *Documentación*. (Recuerda que puedes conmutar a la vista de Documentación usando el menú en la esquina superior derecha del editor.) También puedes mirar el código fuente si quieres, pero esto no es estrictamente necesario por ahora.

Ejercicio 6.9 Coloca un objeto `Body` en el mundo. ¿Cuáles de los métodos que hereda de `SmoothMover` puedes llamar interactivamente (a través del menú del objeto)? ¿Cuáles no puedes llamar en este momento?

6.3 La clase dada Body

Ejercicio 6.10 Abre el código fuente de la clase `Body`. Estúdialo.

Mirando el código de la clase `Body`, hay dos aspectos que podemos explicar con más detalle. El primero es el hecho de que la clase tenga dos constructores (Código 6.1). Éste es otro ejemplo de sobrecarga: es perfectamente legal tener dos constructores en una clase si tienen diferentes listas de parámetros.

En nuestro caso, un constructor no tiene parámetros, y el otro tiene cuatro parámetros.

Código 6.1
Constructores de la clase `Body`

```
public class Body extends SmoothMover
{

    // código omitido

    private double mass;

    /**
     * Construye un objeto Body con valores por defecto para tamaño, masa,
     * movimiento y color.
     */
    public Body()
    {
         this (20, 300, new Vector(90, 1.0), defaultColor);
    }


    /**
     * Construye un objeto Body con valores especificados para tamaño,
     * masa, movimiento y color
     */
    public Body(int size, double mass, Vector movement, Color color)
    {
         this.mass = mass;
         addForce(movement);
         GreenfootImage image = new  GreenfootImage (size, size);
         image.setColor (color);
         image.fillOval (0, 0, size-1, size-1);
         setImage (image);
    }

    // más código omitido
}
```

Terminología

Un constructor sin ningún parámetro se denomina **constructor por defecto**.

El constructor por defecto nos simplifica crear objetos body interactivamente sin tener que especificar todos los detalles. El segundo constructor permite la construcción de un objeto body especificando tamaño, masa, movimiento y color. Este último constructor se usa, por ejemplo, en la clase `Space` para crear el Sol, un planeta y la Luna.

El segundo constructor inicializa el estado de los actores con todos los valores de los parámetros que se le han pasado. El primer constructor parece un poco más misterioso. Sólo tiene una línea de código:

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

Concepto:

La palabra clave **this** se usa para llamar a un constructor desde otro, o para referirnos al objeto actual.

La línea casi parece como una llamada a un método, excepto porque usa la palabra clave `this` en vez de un nombre de método. Al usar esta llamada, el constructor ejecuta el otro constructor (el que tiene cuatro parámetros), y proporciona parámetros por defecto para los cuatro valores. El uso de la palabra clave `this` de esta forma (como un nombre de método) sólo está permitido dentro de constructores para llamar a otro constructor como parte de la inicialización.

Hay un segundo uso de la palabra clave `this`:

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

Aquí tenemos otro ejemplo de sobrecarga: el mismo nombre se usa para dos variables (un parámetro y un campo del objeto). Cuando asignamos estos valores, necesitamos especificar cuál de estas dos variables llamadas `mass` queremos usar a ambos lados de la asignación.

Cuando escribimos `mass` sin más, entonces se usa la definición más cercana de una variable con ese nombre —en este caso, el parámetro. Cuando escribimos `this.mass`, especificamos que nos referimos al campo `mass` del objeto actual. Por tanto, esta línea de código asigna el parámetro denominado `mass` al campo denominado `mass`.

Ejercicio 6.11 Borra la cadena «`this.`» antes del nombre `mass` en la línea de código mostrada arriba, de forma que diga

```
mass = mass;
```

¿Compila este código? ¿Se ejecuta? ¿Qué piensas que hace este código? ¿Cuál es su efecto? (Crea un objeto y usa la función de inspeccionar para examinar el campo `mass`. Una vez hayas terminado de experimentar, restaura el código como estaba antes.)

El segundo aspecto que merece la pena ver con más detalle son las dos líneas cerca del principio de la clase, mostradas en el cuadro de Código 6.2.

Código 6.2
Declaración de constantes

```
private static final double GRAVITY = 5.8;
private static final Color defaultColor = new Color(255, 216, 0);
```

Concepto:

Una **constante** es un valor con un nombre que se puede usar de forma similar a una variable, pero que no cambia nunca.

Estas dos declaraciones parecen similares a las declaraciones de campos, salvo que tienen las dos palabras clave `static` final tras la palabra clave `private`.

Esto es lo que llamamos una *constante*. Una constante se parece a un campo en que usamos el nombre en nuestro código para referirnos a su valor, pero el valor no cambia nunca (es *constante*). La palabra clave final es la que hace que esta declaración sea constante.

El efecto de la palabra clave `static` es que esta constante se comparta por todos los actores de esta clase, y no necesitemos hacer copias separadas para cada objeto. Ya nos encontramos con la palabra clave `static` antes (en el Capítulo 3), en el contexto de métodos de clase. Al igual que los métodos estáticos pertenecen a la clase misma (pero se pueden llamar desde los objetos de esa clase), los campos estáticos pertenecen a la clase y pueden llamarse desde sus instancias.

En este caso, las constantes declaradas son un valor para la gravedad [1] (que usaremos más tarde), y un color por defecto para los objetos body. Esto es un objeto de tipo `Color`, que veremos con más detalle a continuación.

Es una buena práctica declarar campos constante que no deberían cambiar en un programa. Al hacer que el campo sea constante, prevenimos cualquier cambio accidental del valor en el código.

6.4 Primera extensión: creando movimiento

Vale, basta ya de mirar lo que pone. Es hora de escribir algún código y ver lo que pasa. El primer experimento obvio es hacer que nuestros cuerpos celestes se muevan. Hemos mencionado que la clase `SmoothMover` proporciona un método `move()`, y como un `Body` es un `SmoothMover`, los cuerpos celestes tienen también acceso a este método.

Ejercicio 6.12 Añade una llamada al método `move()` en el método `act` de `Body`. Prúebalo. ¿Cuál es la dirección por defecto del movimiento? ¿Cuál es la velocidad por defecto?

Ejercicio 6.13 Crea varios objetos `Body`. ¿Cómo se comportan?

Ejercicio 6.14 Llama a los métodos públicos de `Space` (`sunAndPlanet()`, etc.) y ejecuta el escenario. ¿Cómo se mueven estos objetos? ¿Dónde se define la dirección y la velocidad iniciales de su movimiento?

Ejercicio 6.15 Cambia la dirección por defecto de un cuerpo celeste para que sea hacia la izquierda. Esto es, cuando se crea un cuerpo celeste usando el constructor por defecto y se ejecute su método `move()`, debería moverse a la izquierda.

Como vemos cuando realizamos estos experimentos, para que los cuerpos celestes se muevan, basta con decírselo. Sin embargo, se mueven en línea recta. Esto se debe a que el movimiento (velocidad y dirección) está especificado por su vector de movimiento y, actualmente, nada cambia este vector. Por tanto, el movimiento es constante.

6.5 Usando las clases de la librería Java

Cuando hemos leído el código de las clases `Body` y `Space`, nos hemos encontrado con la clase `Color`. El segundo constructor de la clase `Body` espera un parámetro de tipo `Color`, y el código de la clase `Space` crea objetos `Color` con una expresión como

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

Los tres parámetros del constructor de `Color` son los componentes rojo, verde y azul de un color en concreto. Cada color en una pantalla de ordenador se puede describir como la composición de estos

[1] Nuestro valor de la gravedad no tiene relación directa con ninguna unidad de la naturaleza en concreto. Es una unidad arbitraria para este escenario. Una vez que hayamos comenzado a implementar la gravedad para nuestros cuerpos celestes, puedes experimentar con valores diferentes de la gravedad cambiando este valor.

Concepto:

La **librería de clases Java** es una colección amplia de clases proporcionadas con el sistema Java. Podemos usar estas clases usando una sentencia **import**

tres colores base. (Veremos con más detalle el tema del color en el Capítulo 8. Allí, en la página 138, encontrarás también una tabla de valores de color RGB [2]. Puedes usar cualquier programa bueno de gráficos para experimentar con estos valores tú mismo.)

Para nosotros, la pregunta más acuciante ahora es ¿de dónde sale esta clase? Y ¿cómo sabes qué parámetros espera su constructor?

Encontramos una pista a esta pregunta al principio de nuestra clase, donde encontramos la línea

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

La clase `Color` es una de las muchas clases que están en la *Librería Estándar de Clases Java*. El sistema Java viene con una gran colección de clases listas para ser usadas. Con el tiempo, conoceremos muchas de ellas.

Podemos ver la documentación de todas las clases de la librería seleccionando *Documentación de la Librería Java* en el menú *Ayuda* de Greenfoot. Se abrirá la documentación de la librería Java en un navegador web (Figura 6.6).

El panel inferior izquierdo de esta ventana nos muestra una lista de todas las clases de la librería Java. (¡Hay muchas!). Podemos mirar la documentación de una clase concreta buscándola en esta lista y seleccionándola. Cuando la seleccionamos, la parte principal de la ventana muestra la documentación de esta clase.

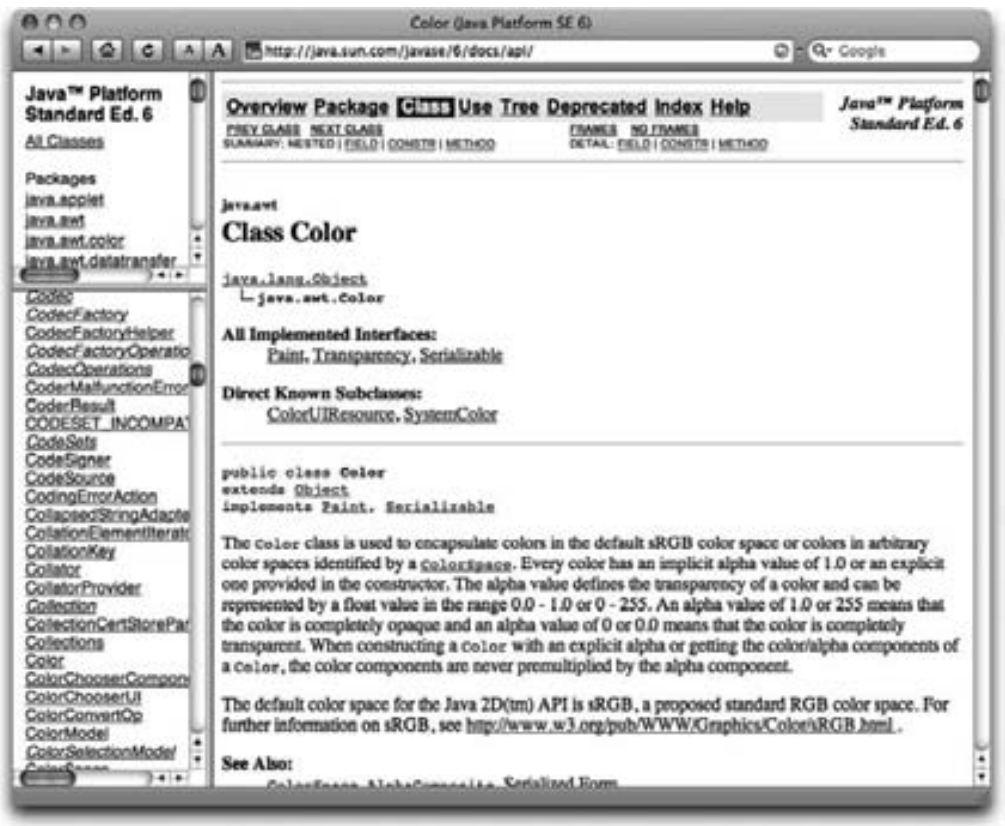


Figura 6.6
La documentación de la librería *Java*

[2] N. del T. RGB son las siglas de Red (rojo), Green (verde) y Blue (azul) en inglés.

Ejercicio 6.16 Busca la clase `Color` en la lista de clases. Selecciónala. Mira su documentación. ¿Cuántos constructores tiene?

Ejercicio 6.17 Encuentra la descripción del constructor que hemos usado (el que tiene tres enteros como parámetros). ¿Cuál es el rango válido para estos números enteros?

Puedes ver que hay literalmente miles de clases en la librería Java. Para tener algo de orden en esta lista tan larga, las clases se agrupan en *paquete*. Un paquete es un grupo de clases relacionadas lógicamente. Al principio de la documentación de cualquier clase, podemos ver a qué paquete pertenece. La clase `Color`, por ejemplo, está en un paquete llamado *java.awt*.

Cuando queramos usar cualquiera de las clases de la librería Java en nuestro propio escenario, tenemos que *importar* la clase, usando la sentencia import como hemos visto antes. La sentencia import nombra el paquete y la clase que queremos usar, con un punto entre ambos. Por tanto, para usar la clase `Color` del paquete *java.awt,* escribimos

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

Cuando importamos una clase, la podemos usar en nuestro propio escenario, igual que si fuera una de nuestras propias clases. Después de importarla, podemos crear objetos de esta clase, llamar a métodos, o hacer cualquier cosa que quisiéramos hacer con una clase.

La librería de Java es bastante intimidante al principio, ya que tiene muchas clases. No te preocupes —usaremos sólo un pequeño número de ellas, y las introduciremos una a una cuando las necesitemos.

Una de ellas, sin embargo, la necesitaremos muy pronto en la siguiente sección.

Lo que queremos hacer a continuación es añadir la gravedad a este escenario. Esto es, cuando tenemos más de un cuerpo en nuestro espacio, la fuerza gravitatoria entre estos cuerpos debería cambiar el movimiento de cada cuerpo.

6.6 Añadiendo fuerza gravitatoria

Comencemos mirando al método actual `act` de nuestra clase `Body` (Código 6.3). (Si no has hecho el Ejercicio 6.12, entonces no tendrás la llamada al método `move`, puedes añadirla ahora.)

Código 6.3
El método actual `act`

```
/**
 * Actúa. Esto es: aplica las fuerzas gravitatorias entre todos los cuerpos, y
 * después, múévelos.
 */
public void act()
{

     move();
}
```

Aunque el código actual contiene sólo la llamada al método `move`, el comentario describe correctamente lo que queremos hacer: antes de movernos, deberíamos aplicar las fuerzas causadas por la atracción gravitatoria de todos los otros cuerpos en el espacio.

Podemos dar un bosquejo de la tarea en pseudocódigo:

```
aplica fuerzas de otros cuerpos:
      obtén todos los otros cuerpos del espacio;
      para cada uno de estos cuerpos:
      {
           aplica la gravedad de ese cuerpo al nuestro propio;
}
```

Como esto no es aún una cosa fácil que hacer, comenzaremos haciendo un método separado para esta tarea (Código 6.4). El hecho de crear un método separado (inicialmente vacío) podría parecer una tarea trivial, ya que no conseguimos mucho, pero nos ayuda para descomponer nuestro problema en problemas más pequeños, y esto ayuda a estructurar nuestros pensamientos.

Código 6.4
Preparando para aplicar fuerzas gravitatorias

```
/**
 /* Actúa. Esto es: aplica las fuerzas gravitatorias entre todos los cuerpos,
 * y después, muévelos.
 */
public void act()
{
    applyForces();
    move();
}

/**
 * Aplica las fuerzas gravitatorias de todos los cuerpos celestes en
 * este universo
 */
private void applyForces()
{
    // trabajo pendiente aquí
}
```

Concepto:

Los **métodos privados** son sólo visibles dentro de la clase en que están definidos. Se usan para mejorar la estructura del código.

Nota: Métodos privados

El método que hemos creado en el cuadro del Código 6.4 tiene la palabra clave `private` al principio de su signatura, en vez de `public` como habíamos visto hasta ahora.

Los métodos pueden ser públicos (`public`) o privados (`private`). Cuando los métodos están pensados para ser llamados fuera de la clase (bien interactivamente por un usuario o por otra clase), entonces deben ser públicos. Cuando los métodos están pensados para ser llamados sólo desde métodos de la misma clase (como sucede aquí), entonces deberían ser declarados como privados.

Los métodos privados no son visibles o accesibles desde fuera de la clase. Es una buena práctica hacer métodos privados cuyo único propósito es su uso interno. Esto nos ayuda a prevenir errores y documenta el propósito del método más claramente.

A continuación, tenemos que resolver cómo podemos acceder a todos los objetos de nuestro mundo.

La clase `World` de Greenfoot tiene métodos que nos dan acceso a los objetos de un mundo.

Ejercicio 6.18 Busca la clase `World` de Greenfoot en la documentación de Greenfoot. Encuentra todos los métodos que nos dan acceso a los objetos del mundo. Anótalos.

El método más interesante de todos estos métodos es:

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

Este método nos da una lista de todos los objetos del mundo de una clase particular. El parámetro que recibe este método es del tipo `java.lang.Class` (es decir, la clase llamada `"Class"` del paquete `java.lang`[3]). Ya hemos visto parámetros de este tipo antes, en el Capítulo 4, cuando usamos los métodos `canSee` y `eat` en la clase `Crab` para comer los gusanos. Podemos usar una llamada a este método para, por ejemplo, obtener una lista de todos los objetos `Body` en nuestro mundo:

```
getObjects(Body.class)
```

Podemos también pasar `null` como un parámetro para recibir una lista de todos los objetos de cualquier clase en el mundo:

Concepto:

La palabra clave **null** quiere decir «nada» o «ningún objeto».

```
getObjects(null)
```

La palabra clave `null` es una expresión especial que significa *nada*, o *ningún objeto*. Si la usamos en una lista de parámetros, no pasamos ningún objeto como parámetro. El valor `null` podría también asignarse a variables.

El método `getObjects` (*obténObjetos*) es, sin embargo, un método de la clase `World`, por lo que debe invocarse en un objeto `World`. Antes de escribir código en nuestra clase `Body`, debemos obtener primero el objeto `World` para poder llamar a este método. Afortunadamente, hay un método en la clase `Actor` que nos da acceso al objeto mundo. Su signatura es

```
World getWorld()
```

Busca este método en la documentación de las clases de Greenfoot y lee su descripción.

Este método devuelve el objeto `World`, y podemos llamar a continuación al método `getObjects` en el objeto resultante:

```
getWorld().getObjects(Body.class)
```

Este código se puede usar desde un actor para obtener todos los objetos de la clase `Body`. Veamos más detenidamente el tipo de vuelta ahora.

El tipo de vuelta del método `getObjects` está especificado como `java.util.List`. Esto nos indica que hay un tipo denominado `List` en el paquete `java.util` de la librería estándar de clases, y que obtendremos un objeto de este tipo como resultado de este método.

El tipo `List` merece que lo veamos con más detalle.

---

[3] El paquete java.lang es especial. Contiene las clases más utilizadas, y sus clases se importan automáticamente. Por tanto, no necesitamos escribir una sentencia import para ninguna clase del paquete java.lang.

6.7 El tipo List

Concepto:

Una **colección** es un tipo de objeto que puede contener muchos otros objetos.

Tratar con colecciones de objetos es importante tanto en la programación en Greenfoot como en la programación en general. Varios métodos de Greenfoot devuelven colecciones de objetos como su resultado, normalmente en forma de una lista. El tipo de objetos devueltos entonces es del tipo `List` del paquete `java.util`.

Concepto:

Una **lista** es un ejemplo de colección. Algunos métodos de la API de Greenfoot devuelven objetos `List`.

Nota al margen: Interfaces

El tipo `List` es un poco diferente de otros tipos de objetos que hemos visto: no es una clase, sino una *interfaz*. Las interfaces son un constructor Java que proporciona abstracción sobre las diferentes clases Java que la pueden implementar. Los detalles por ahora no son importantes —es suficiente saber que podemos tratar el tipo `List` de forma similar a otros tipos—. Podemos buscarla en la Documentación de la Librería Java, y podemos llamar a los métodos existentes en el objeto. No podemos, sin embargo, crear directamente objetos de tipo `List`. Volveremos a este tema más adelante.

Ejercicio 6.19 Busca `java.util.List` en la Documentación de la Librería Java. ¿Cuáles son los nombres de los métodos usados para añadir un objeto a la lista, borrar un objeto de la lista, y encontrar cuántos objetos hay actualmente en la lista?

Ejercicio 6.20 ¿Cuál es el nombre correcto de este tipo, tal como aparece en la parte superior de la documentación?

Cuando vimos el método `getObjects` en la sección previa, observamos que devuelve un objeto de tipo `java.util.List`. Por tanto, para almacenar este objeto, necesitamos declarar una variable de este tipo. Haremos esto en nuestro método `applyForces` (*aplicaFuerzas*).

El tipo `List`, sin embargo, es diferente del resto de tipos que hemos visto hasta ahora. La documentación muestra al principio

```
Interface List<E>
```

Concepto:

Un **tipo genérico** es un tipo que recibe el nombre de un segundo tipo como parámetro.

Al margen de la palabra *interface* en lugar de *class*, vemos una notación nueva: la palabra <E> después del nombre del tipo.

Formalmente, esto se llama un *tipo genérico*. Esto significa que el tipo `List` necesita que se especifique un tipo adicional como parámetro. Este segundo tipo especifica el tipo de los elementos que se guardan en la lista.

Por ejemplo, si estamos tratando con una lista de strings, especificaríamos el tipo como

```
List<String>
```

Si en cambio estamos tratando con una lista de actores, podemos escribir

```
List<Actor>
```

En cada caso, el tipo dentro de los símbolos menor y mayor (<>) es el tipo de algún tipo conocido de objetos. En nuestro caso, esperamos una lista de cuerpos, por lo que la declaración de nuestra variable será:

```
List<Body> bodies
```

Podemos asignar ahora la lista que recuperamos del método `getObjects` a esta variable:

```
List<Body> bodies = getWorld().getObjects(Body.class);
```

Después de ejecutar esta línea, nuestra variable bodies guarda una lista de todos los cuerpos que hay ahora mismo en el mundo (mira también el cuadro del Código 6.5). (Rcuerda que debes añadir también una sentencia de importación para `java.util.List` al comienzo de tu clase.)

Código 6.5
Obteniendo una lista de todos los cuerpos en el espacio

```
private void applyForces()
{

    List<Body> bodies = getWorld().getObjects(Body.class);
    ...
}
```

6.8 El bucle *for-each*

El siguiente paso que queremos conseguir, ahora que tenemos una lista de todos los cuerpos, es aplicar la fuerza gravitatoria de cada cuerpo a nuestro movimiento.

Haremos esto recorriendo la lista de cuerpos uno a uno, y aplicando en turno la fuerza gravitatoria de cada cuerpo.

Concepto:

El **buble *for-each*** es otro tipo de bucle. Es adecuado para procesar todos los elementos de una colección.

Java tiene un bucle especializado para recorrer cada elemento de una colección, y podemos usar este bucle aquí. Se llama el bucle *for-each*, y se escribe según el patrón:

```
for (TipoElemento variable : colección)
{
    sentencias;
}
```

En este patrón, `TipoElemento` representa el tipo de cada elemento de la colección, `variable` es una variable que se declara aquí, de forma que podemos darle cualquier nombre que queramos, `colección` es el nombre de la colección que queremos procesar, y `sentencias` es una secuencia de sentencias que queremos llevar a cabo. Estará más claro con un ejemplo.

Usando nuestra lista llamada `bodies`, podemos escribir

```
for (Body body : bodies)
{
    body.move();
}
```

(Recuerda que Java es sensible a la diferencia entre mayúsculas y minúsculas: `Body` con una «B» mayúscula es diferente de `body` con una «b» minúscula. El nombre en mayúsculas se refiere a la

clase, mientras que el nombre en minúsculas se refiere a la variable conteniendo un objeto. La versión en plural —`bodies`— es otra variable que guarda la lista entera.)

Podemos leer el bucle *for-each* un poco más fácilmente si leemos la palabra clave `for` como «for each» (para cada), los dos puntos como «en», y la llave abierta como «haz». Entonces es

```
para cada body en bodies haz:...
```

Esta lectura nos da también una pista de lo que hace el bucle: ejecutará las sentencias entre llaves una vez para cada elemento de la lista `bodies`. Si, por ejemplo, hay tres elementos en esa lista, las sentencias serán ejecutadas tres veces. Cada vez, antes de que las sentencias sean ejecutadas, se asignará a la variable `body` (declarada en la cabecera del bucle) uno de los elementos de la lista. Por tanto, la secuencia de acciones será

```
body = primer elemento de'bodies';
ejecuta sentencias del bucle;
body = segundo elemento de 'bodies';
ejecuta sentencias del bucle;
body = tercer elemento de 'bodies';
ejecuta sentencias del bucle;
...
```

La variable `body` se puede usar en las sentencias del bucle para acceder al elemento actual que estamos recorriendo de la lista. Podríamos, por ejemplo, llamar a un método de este objeto, como en el ejemplo mostrado antes, o pasar el objeto a otro método para que lo procese.

Podemos ahora usar este bucle para aplicar la gravedad de todos los cuerpos a éste:

```
for (Body body : bodies)
{
    applyGravity(body);
}
```

En este código, simplemente tomamos cada elemento (almacenado en la variable `body`) y se lo pasamos a otro método llamado `applyGravity`, que escribiremos en un momento.

Deberíamos añadir una cosa más: como `bodies` es una lista de todos los cuerpos en el espacio, también incluye el objeto actual (al que queremos aplicar la fuerza gravitatoria). No necesitamos aplicar la gravedad de un objeto consigo mismo, así que podemos añadir una sentencia if que llame a `applyGravity` sólo si el elemento de la lista no es el objeto actual.

El resultado se muestra en el cuadro del Código 6.6. Observa que la palabra clave `this` se usa para referirnos al objeto actual.

Código 6.6
Aplicando la gravedad del resto de objetos del espacio

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

    for (Body body : bodies)
    {

         if (body != this)
```

```
            {
                applyGravity (body);
            }
        }
    }

    /**
     *  Aplica la fuerza gravitatoria de un cuerpo dado con éste.
     */
    private void applyGravity(Body other)
    {
        // por hacer
    }
```

6.9 Aplicando la gravedad

En el cuadro del Código 6.6, hemos resuelto la tarea de acceder a cada objeto del espacio, pero hemos retrasado la tarea de aplicar la fuerza gravitatoria. Necesitamos aún escribir el método `applyGravity` (*aplicaGravedad*) —otro ejemplo de un método privado.

Esto es ahora un poco más fácil que antes, ya que ahora este método sólo necesita tratar con dos objetos a la vez: el objeto actual, y otro objeto especificado como parámetro. Queremos aplicar la fuerza gravitatoria desde este otro objeto al primero. Aquí podemos usar la Ley de Newton.

La fórmula de la gravedad de Newton es como sigue:

$$\text{fuerza} = \frac{\text{masa1} \times \text{masa2}}{\text{distancia}^2} \text{G}$$

En otras palabras, para calcular la fuerza que necesitamos aplicar al objeto actual, necesitamos multiplicar la masa de este objeto por la masa del otro objeto, y dividir por la distancia al cuadrado entre los dos objetos. Finalmente, el valor se multiplica por la constante G —la *constante gravitatoria*. (Puede que recuerdes que ya definimos una constante para este valor en nuestra clase, llamado *GRAVITY*.)

Si estás seguro de ti mismo, o te atreves, puedes intentar implementar tú mismo el método `applyGravity`. Necesitas crear un vector en la dirección del cuerpo actual al otro cuerpo, con una longitud especificada por esta fórmula. Para el resto, podemos ver la implementación terminada de este método (Código 6.7).

Código 6.7
Calculando y aplicando la gravedad desde otro cuerpo

```
/**
 *  Aplica la fuerza de la gravedad de un cuerpo dado a éste.
 */
private void applyGravity(Body other)
{

    double dx = other.getExactX() - this.getExactX();
    double dy = other.getExactY() - this.getExactY();
    Vector force = new Vector (dx, dy);
```

```
    double distance = Math.sqrt (dx * dx  + dy * dy);
    double strength = GRAVITY * this.mass * other.mass /
                      (distance * distance);
    double acceleration = strength / this.mass;
    force.setLength (acceleration);
    addForce (force);
}
```

Este método no es tan complicado como parece. Primero, calculamos la distancia entre nuestro objeto y el otro objeto en las coordenadas *x* e *y* (*dx* y *dy*). Luego, creamos un nuevo vector con estos valores. Este vector tiene ahora la dirección correcta, pero no longitud correcta.

A continuación, calculamos la distancia entre los dos objetos usando el *teorema de Pitágoras* ($a^2 + b^2 = c^2$ para triángulos rectángulos, mira la Figura 6.7).

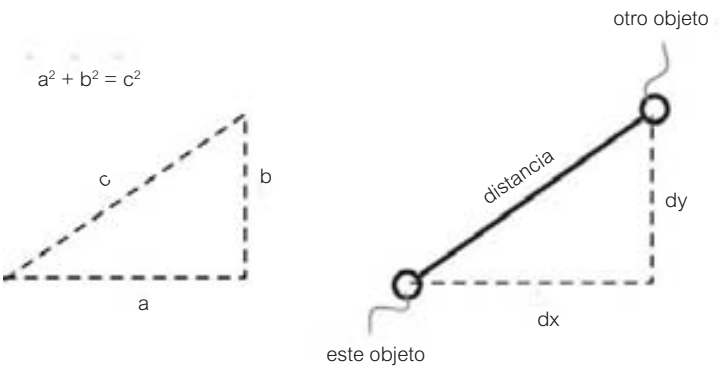


Figura 6.7
La distancia en relación a *dx* y *dy*

Esto nos dice que la distancia es la raíz cuadrada de *dx* al cuadrado más *dy* al cuadrado. En nuestro código (Código 6.7), usamos un método llamado `sqrt`[4] de la clase `Math` para calcular la raíz cuadrada. (`Math` es una clase de java.lang, y se importa por tanto automáticamente.)

Ejercicio 6.21 Busca la clase `Math` en la documentación de Java. ¿Cuántos parámetros tienen el método `sqrt`? ¿De qué tipo son? ¿Qué tipo devuelve?

Ejercicio 6.22 En la clase `Math`, busca el método que se puede usar para calcular el máximo de dos enteros. ¿Cómo se llama?

La siguiente línea de código calcula la fuerza usando la fórmula de la gravedad de Newton vista antes.

Lo que nos queda ya por hacer es calcular la aceleración, a que el movimiento actual que cambia nuestro objeto no se determina sólo por la fuerza de la gravedad, sino también por la masa de nuestro

[4] N. del T. Raíz cuadrada es *square root* en inglés, de ahí *sqrt*.

objeto. Cuanto más pesado sea el objeto, más lentamente acelerará. La aceleración se calcula con la siguiente fórmula:

$$\text{aceleración} = \frac{\text{fuerza}}{\text{masa}}$$

Una vez que hayamos calculado la aceleración, podemos fijar nuestro vector de fuerza con la longitud correcta y añadir este vector al movimiento de nuestro cuerpo. Esto es fácil de hacer usando el método `addForce` (*añadeFuerza*) que nos proporciona la clase `SmoothMover`.

Ejercicio 6.23 Busca la correspondencia entre las variables del Código 6.7 y las de la fórmula de Newton, el teorema de Pitágoras y la fórmula de la aceleración dada arriba. ¿Qué variable se corresponde con qué parte de la fórmula?

Con esto, nuestra tarea está completa. (Una implementación del código descrito hasta aquí está disponible en los escenarios del libro como *Newtons-Lab-2*.)

Esta tarea ha requerido claramente más conocimiento de matemáticas y física que las vistas hasta ahora. Si no es tu área favorita, no te preocupes, volveremos a proyectos menos matemáticos en breve. Recuerda: programando puedes hacer cualquier cosa que quieras. Puedes hacerlo muy matemático, pero también puede ser muy creativo y artístico.

6.10 Probando

Ahora que hemos terminado nuestra implementación de las fuerzas gravitatorias, es hora de probarlo. Podemos comenzar con los tres escenarios que vienen definidos en la clase `Space`.

Ejercicio 6.24 Con el código de la gravedad completo, vuelve a probar los tres métodos de inicialización del objeto `Space` (`sunAndPlanet()`, `sunAndTwoPlanets()`, y `sunPlanetMoon()`). ¿Qué observas?

Ejercicio 6.25 Experimenta con cambios en la gravedad (la constante `GRAVITY` al principio de la clase `Body`).

Ejercicio 6.26 Experimenta con cambios en la masa y/o movimiento inicial de los cuerpos (definidos en la clase `Space`).

Ejercicio 6.27 Crea algunas inicializaciones nuevas de estrellas y planetas y mira cómo interaccionan. ¿Cómo consigues un sistema que sea estable?

Verás rápidamente que es muy complicado encontrar la configuración de parámetros que consiga que el sistema permanezca estable un buen rato. La combinación de masa y gravedad a menudo hace que los objetos choquen entre sí, o salgan de la órbita. (Dado que no hemos implementado el «choque entre objetos», nuestros objetos simplemente vuelan entre sí. Sin embargo, cuando están muy próximos, su fuerza aumenta y a menudo se catapultan entre sí con trayectorias extrañas.)

Escollo

Ten cuidado cuando uses el constructor de la clase `Vector`. El constructor está sobrecargado: una versión espera un `int` y un `double` como parámetros, el otro espera dos `doubles`. Por tanto

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

llamará a un constructor, mientras que

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

llamará al otro constructor, lo que producirá un vector completamente diferente.

Algunos de estos efectos son similares a los de la naturaleza, aunque nuestra simulación es un poco inmadura debido a algunas simplificaciones que hemos hecho. El hecho, por ejemplo, de que todos los objetos actúen en secuencia, en vez de simultáneamente, tendrá un efecto en la conducta y no es una representación realista. Para hacer la simulación más precisa, podríamos calcular primero todas las fuerzas (sin realizar movimientos) y entonces ejecutar todos los movimientos según los cálculos previos. Además, nuestra simulación no modela las fuerzas con precisión cuando dos objetos están muy cerca el uno del otro, añadiendo efectos poco reales.

Podemos también preguntarnos sobre la estabilidad de nuestro propio sistema solar. Mientras que las órbitas de los planetas en nuestro sistema solar son bastante estables, los detalles precisos de su movimiento son más difíciles de predecir con precisión durante mucho tiempo en el futuro. Estamos bastante seguros de que ninguno de los planetas chocará con el Sol en los próximos billones daños, pero pequeñas variaciones de la órbita podrían ocurrir. Simulaciones como la nuestra (pero mucho más precisas y detalladas, aunque similares en esencia) se usan para intentar predecir las órbitas futuras. Hemos visto, sin embargo, que es muy difícil simular con precisión. Las simulaciones pueden mostrar que pequeñas diferencias en las condiciones similares pueden suponer grandes diferencias después de pocos billones de años [5].

Viendo lo difícil que es encontrar los parámetros para crear un sistema que sea estable durante un período limitado, nos podría sorprender que nuestro sistema solar sea estable como lo es. Sin embargo, hay una explicación: cuando se formó el sistema solar, el material de una nube de gas alrededor del Sol formó cúmulos que lentamente crecieron al colisionar con otros cúmulos de materia y se combinaron para formar objetos de crecimiento continuo. Inicialmente, hubo incontables cúmulos en órbita. Con el tiempo, algunos cayeron en el Sol, algunos se escaparon al espacio infinito. Este proceso terminó cuando sólo quedaron algunos trozos que estaban bien separados unos de otros y en órbitas estables.

Sería posible crear una simulación que modele este efecto. Si modelamos correctamente el crecimiento de los planetas partiendo de billones de pequeños y aleatorios cúmulos de materia, observaríamos el mismo efecto: quedarían algunos planetas grandes que estarían en órbitas bastante estables. Para esto, sin embargo, necesitaríamos una simulación mucho más detallada y complicada y mucho tiempo: simular este efecto nos llevaría mucho tiempo, incluso en ordenadores muy rápidos.

6.11 Gravedad y música

Antes de dejar atrás nuestro escenario del *Laboratorio de Newton*, vamos a jugar con una cosa más: añadirle música. Bueno, ruido en cualquier caso [6].

---

[5] Si estás interesado en leer más, la Wikipedia es un buena referencia inicial: http://en.wikipedia.org/wiki/Stability_of_the_solar_system.

[6] La idea de añadir sonido al proyecto de la gravedad fue inspirado por el *planetario de Kepler* (mira https://keplers-orrery. dev.java.net/ o busca «Kepler's Orrery» en YouTube).

La idea es la siguiente: añadimos un número de *Obstáculos* a nuestro mundo. Cuando los planetas tocan los obstáculos, reproducen un sonido. Vamos a crear unos pocos planetas o estrellas, dejarles flotar, y ver qué pasa.

No vamos a estudiar esta implementación con detalle. Te la dejamos como auto-estudio, y simplemente vamos a destacar algunos de los aspectos más interesantes. Puedes encontrar una implementación de esta idea en los escenarios de libro en *Newtons-Lab-3*.

Ejercicio 6.28 Abre el escenario llamado *Newtons-Lab-3* y ejecútalo. Mira su código fuente. Intenta entender qué hace.

A continuación listamos los cambios más interesantes que hemos hecho para crear esta versión a partir de la anterior:

- Hemos añadido una nueva clase `Obstacle` (*Obstáculo*). Puedes ver fácilmente objetos de esta clase en la pantalla.
- Los objetos tienen dos imágenes: el rectángulo naranja que puedes ver la mayor parte del tiempo, y una versión más ligera del rectángulo para mostrar cuando son tocados. Esto se usa para crear el efecto «iluminado». Los obstáculos también tienen asociado un fichero de sonido, igual que con las teclas del piano del escenario del piano, por eso suenan igual.
- Hemos modificado la clase `Body` para que los cuerpos reboten con los bordes de la pantalla. Es un efecto mejor para este tipo de escenario. También hemos incrementado un poco la gravedad para acelerar el movimiento y hemos modificado el código para que los cuerpos desaceleren automáticamente cuando van muy rápido. De otra forma, irían más rápido de manera indefinida.
- Finalmente, hemos añadido código en la clase `Space` para crear una fila fija de obstáculos, y para crear cinco planetas aleatorios (tamaño, masa y color aleatorios).

La implementación de estos cambios incluye algunos trozos de código interesantes que podemos destacar.

- En la clase `Obstacle`, hemos usado un método llamado `getOneIntersectingObject` (*obténUnObjetoCruzado*) para comprobar si el obstáculo está siendo golpeado por un planeta. El patrón del código es:

Concepto:

La API de Greenfoot contiene métodos para **detectar colisiones**. Éstos hacen posible detectar cuándo un actor toca a otro. (Más sobre esto se verá en el siguiente capítulo.)

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

  El método `getOneIntersectingObject` se define en la clase `Actor`, y está disponible a todos los actores. Devolverá un actor si nuestro objeto colisiona con otro, o `null` si ningún otro actor colisiona con él. La siguiente sentencia `if` que comprueba si `body` es `null`, comprueba por tanto si hay algún objeto que colisione con el actual.

  Esto es un ejemplo de *detección de colisiones*, que veremos más tarde en el siguiente capítulo.

- En la clase `Space`, hemos añadido dos métodos, `createObstacles` (*creaObstáculos*) y `randomBodies` (*cuerposAleatorios*). El primero crea los obstáculos con sus nombres de ficheros asocia-

dos, de forma similar al código de inicialización del ejemplo del piano. El segundo usa un bucle `while` para crear varios objetos `Body`. Los cuerpos se inicializan con valores aleatorios. El bucle `while` cuenta hacia atrás desde un número hasta 0, para crear el número correcto de objetos. Conviene estudiarlo como otro ejemplo de bucle.

Ejercicio 6.29 Cambia el número de cuerpos que se crean por defecto en este escenario.

Ejercicio 6.30 Juega con los parámetros de movimiento para ver si puedes crear un movimiento más bonito de los planetas. Los parámetros son: el valor de la `GRAVITY`; el valor de la aceleración cuando rebotan con un borde (actualmente 0.9); el umbral de velocidad (actualmente 7), y la aceleración (0.9) usado en el método `applyForces` para desacelerar los objetos rápidos; y la masa inicial de los planetas (en la clase `Space`).

Ejercicio 6.31 Crea una disposición diferente de obstáculos en tu escenario.

Ejercicio 6.32 Usa diferentes sonidos (diferentes ficheros de sonido) para tus obstáculos.

Ejercicio 6.33 Usa diferentes imágenes para tus obstáculos.

Ejercicio 6.34 Haz que los planetas cambien de color cada vez que chocan con el fin del universo.

Ejercicio 6.35 Haz que los planetas cambien de color cada vez que chocan con un obstáculo.

Ejercicio 6.36 Crea un tipo diferente de obstáculo que conmuta entre activo (on) e inactivo (off) cuando es colisionado. Cuando está activo, parpadea continuamente y produce un sonido en intervalos fijos.

Ejercicio 6.37 Añade algún control por teclado. Por ejemplo, presionando la tecla de flecha derecha podrás añadir un poco de fuerza a la derecha de todos los objetos `Body`.

Ejercicio 6.38 Permite que se añadan más planetas. Pinchando en el universo cuando se está ejecutando, debería crear un nuevo planeta en esa posición.

Hay incontables ideas que pueden hacer este escenario más interesante y bonito para jugar con él. ¡Inventa algunas tú mismo e intenta implementarlas!

6.12 Resumen de técnicas de programación

En este capítulo, hemos tocado un bueno número de conceptos nuevos. Hemos visto un nuevo escenario —el laboratorio de Newton— que simula estrellas y planetas en el espacio. Las simulaciones son en general un tema muy interesante, y volveremos a ellas en el Capítulo 9.

Hemos visto dos clases auxiliares útiles, `SmoothMover` y `Vector`, ambas nos han ayudado a crear un movimiento más sofisticado.

Uno de los temas nuevos más importantes de este capítulo ha sido el uso de otras clases de la Librería de Clases Estándar de Java. Hemos usado `Color`, `Math`, y `List` de la librería. Volveremos a esto con el uso de más clases en los siguientes capítulos.

Otra herramienta nueva ha sido el uso de un nuevo bucle: el bucle *for-each*. Este bucle se usa para hacer algo con cada elemento de una colección Java, como una lista. Éste es otro código que usaremos de nuevo más tarde.

Los bucles *for-each* son especialmente útiles cuando procesamos objetos de una colección. Sin embargo, no se pueden usar sin una colección, y no nos proporcionan un índice cuando procesamos los elementos. Si necesitamos un índice, o un bucle independiente de una colección, debemos entonces usar un bucle `for` o un bucle `while`.

Y finalmente, hemos visto algunos métodos útiles de la API de Greenfoot, como el método `getObjects` de la clase `World` y el método `getOneIntersectingObject` de la clase `Actor`. Este último nos lleva a un área más general de detección de colisiones, que veremos con más detalle en el siguiente capítulo, donde retomaremos el juego *Asteroids* de nuevo.

Resumen de conceptos

- **Sobrecarga** es el uso del mismo nombre de método para dos métodos diferentes o constructores.
- La palabra clave **this** se usa para llamar a un constructor desde otro, o para referirse al objeto actual.
- Una **constante** es un valor con nombre que se puede usar de forma similar a una variable, pero cuyo valor nunca cambia.
- La **librería de clases Java** es una colección amplia de clases del sistema Java. Podemos usar estas clases usando una sentencia **import**.
- Los **métodos privados** sólo son visibles desde la clase en que se declaran. Se usan para mejorar la estructura del código.
- La palabra clave **null** significa «nada» o «ningún objeto».
- Una **colección** es un tipo de objeto que puede contener otros objetos.
- Una **lista** es un ejemplo de una colección. Algunos métodos de la API de Greenfoot nos devuelven objetos List.
- Un **tipo genérico** es un tipo que recibe el nombre de un segundo tipo como un parámetro.
- El **bucle *for-each*** es otro tipo de bucle. Es adecuado para procesar todos los elementos de una colección.
- La API de Greenfoot contiene métodos para **detectar colisiones**. Esto permite detectar cuándo un actor toca a otro. (Veremos más de esto en el siguiente capítulo.)

Términos en inglés

Inglés	Español
assignment	asignación
code snippet	trozo de código
field	campo
instance variable	variable de instancia
pseudo-code	pseudocódigo
reference	referencia
test	prueba, probar
variable	variable

CAPÍTULO

7

Detección de colisiones: asteroides

temas:	más sobre movimiento, control de teclado y detección de colisiones
conceptos:	colecciones (de nuevo), bucle *for,* bucle *for-each* (de nuevo), conversión de tipos (*casting*)

En este capítulo, no introduciremos muchos conceptos nuevos, sino que volveremos a ver y profundizar en la comprensión de algunos temas que hemos visto en los últimos capítulos. Volveremos a ver un escenario que hemos visto antes, al principio del libro: los asteroides (Figura 7.1).

La versión de asteroides que usamos aquí es un poco diferente de la que vimos antes. Se han añadido algunas características (tales como una onda de protones y un contador de puntuaciones), pero no está totalmente implementada.

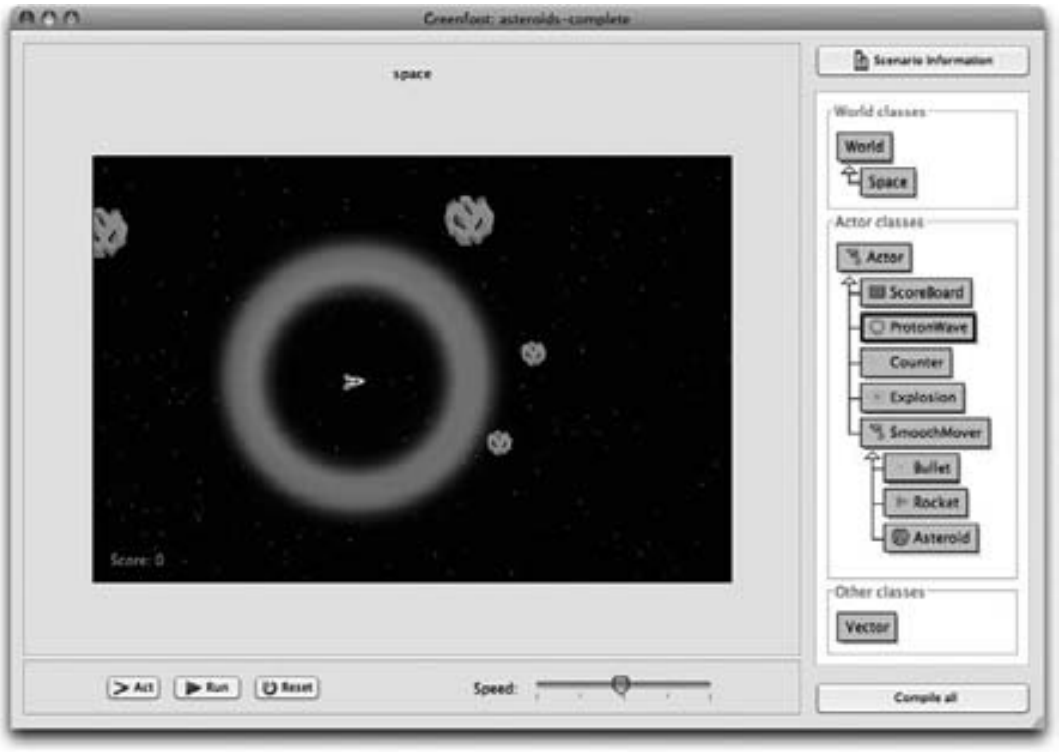


Figura 7.1
El nuevo escenario de asteroides (con la onda de protones)

Faltan partes importantes de la funcionalidad, y será nuestro trabajo en este capítulo implementarlas.

Usaremos este ejemplo para volver a ver el movimiento y la detección de colisiones. En términos de conceptos de programación en Java, usaremos el ejemplo para practicar más con bucles y colecciones.

7.1 Investigación: ¿Qué hay allí?

Debemos comenzar este proyecto examinando el código existente. Tenemos una solución parcialmente implementada, denominada *asteroids-1*, en la carpeta *chapter07* de los escenarios del libro. (Asegúrate de usar la versión de la carpeta *chapter07*, y no la versión de la carpeta *chapter01*.)

Ejercicio 7.1 Abre el escenario *asteroids-1* de la carpeta *chapter07* de los escenarios del libro. Experimenta con él para averiguar qué hace y qué no hace.

Ejercicio 7.2 Anota una lista de cosas que deberían añadirse al proyecto.

Ejercicio 7.3 ¿Qué tecla del teclado se usa para disparar una bala?

Ejercicio 7.4 Coloca una explosión en un escenario en ejecución (crea interactivamente un objeto de la clase `Explosion`). ¿Funciona? ¿Qué hace?

Ejercicio 7.5 Coloca una onda de protones en un escenario. ¿Funciona? ¿Qué hace?

Cuando experimentes con el escenario actual, notarás que falta buena parte de la funcionalidad fundamental:

- El cohete no se mueve. Ni se puede girar ni se puede mover hacia delante.
- No ocurre nada cuando un asteroide colisiona con el cohete. Vuela derecho a través de él, en vez de dañar el cohete.
- Como resultado de esto, no puedes perder. El juego nunca termina, y la puntuación final no se muestra nunca.
- Las clases `ScoreBoard`, `Explosion` y `ProtonWave`, que podemos ver en el diagrama de clases, no parece que hagan nada en el escenario.

Una cosa que podemos hacer, sin embargo, es disparar balas a los asteroides. (Si no has averiguado aún cómo, inténtalo.) Los asteroides estallan cuando son alcanzados por una bala, o desaparecen si son muy pequeños.

El objetivo de este juego sería obviamente limpiar la pantalla de asteroides sin que nuestro cohete sea alcanzado. Para hacerlo un poco más interesante, queremos también añadir otra arma —la onda de protones—. Y queremos mantener un marcador mientras jugamos. Para hacer esto, tenemos bastante trabajo que hacer.

- Tenemos que implementar el movimiento del cohete. Actualmente, puede disparar balas, pero nada más. Necesitamos poder movernos hacia delante y girar.

- Debemos asegurarnos de que el cohete explota cuando nos golpeamos con un asteroide.
- Cuando el cohete explota, queremos que salga un marcador que muestre nuestra puntación final.
- Queremos poder liberar una onda de protones. La onda de protones debería comenzar alrededor del cohete y esparcirse gradualmente a continuación, destruyendo los asteroides cuando los alcanza.

Pero antes de entrar en detalle en estas funciones, vamos a comenzar con un tema menor de tipo cosmético: pintar las estrellas en nuestro universo.

7.2 Pintando estrellas

En todos los escenarios previos, hemos usado una imagen fija para el fondo del mundo. La imagen se guardaba en un fichero de imagen en nuestro sistema de ficheros.

En este escenario, nos gustaría introducir una técnica diferente para hacer imágenes de fondo: pintarlas sobre la marcha.

El escenario de asteroides no usa un fichero de imagen para el fondo. Un mundo que no tiene una imagen de fondo asignado obtendrá, por defecto, una imagen de fondo creada automáticamente que se rellena con un fondo blanco.

Ejercicio 7.6 Investiga el constructor de la clase `Space` de tu escenario. Encuentra las líneas de código que crean el fondo negro.

Mirando el escenario *asteroids-1*, podemos ver que el fondo es negro. Cuando investigamos el constructor de la clase `Space`, podemos encontrar estas tres líneas de código:

```
GreenfootImage  background = getBackground();
background.setColor(Color.BLACK);
background.fill();
```

Consejo:

Si quieres borrar algunas líneas de código temporalmente, es más fácil «comentarlas», en vez de borrarlas. El editor de Greenfoot tiene una función para hacer esto. Simplemente selecciona las líneas en cuestión, e invoca «Comentar» (F8) o «Descomentar» (F7) del menú *Editar*.

Ejercicio 7.7 Borra estas tres líneas de tu clase. Puedes hacer esto comentándolas. ¿Qué observas? (Una vez hecho, vuelve a ponerlas.)

La primera línea recupera la imagen de fondo actual del mundo. Esta imagen (blanca) se ha generado automáticamente. Tenemos entonces una referencia al fondo del mundo almacenada en la variable `background` (*fondo*).

El objeto `background` que hemos almacenado aquí es de la clase `GreenfootImage` —esta clase ya la hemos visto antes.

Ejercicio 7.8 Busca la documentación de la clase `GreenfootImage`. ¿Cuál es el nombre del método para pintar un rectángulo? ¿Cuál es la diferencia entre `drawOval` (*pintaÓvalo*) y `fillOval` (*rellenaÓvalo*)?

La segunda línea del fragmento de código anterior fija el color de relleno a negro. El hacer esto no tiene un efecto inmediato (no cambia el color de la imagen). En cambio, determina el color que será usado en las operaciones de dibujo posteriores. El parámetro es una constante de la clase `Color`, que ya nos encontramos en el capítulo previo.

Ejercicio 7.9 Busca de nuevo la documentación de la clase `Color`. (¿Recuerdas en qué paquete está?) ¿Para cuántos colores define constantes esta clase?

La tercera línea del fragmento de código rellena ahora nuestra imagen con el color escogido. Observa que no necesitamos fijar la imagen de nuevo como fondo del mundo. Cuando tenemos la imagen (usando `getBackground()`), obtenemos una referencia a la imagen de fondo, y la misma imagen permanece aún como imagen de fondo del mundo. No se borra del mundo porque ahora tenemos una referencia a ella.

Cuando pintamos en esta imagen, estamos pintando directamente en el fondo del mundo. Nuestra tarea ahora es pintar algunas estrellas en la imagen de fondo.

Ejercicio 7.10 En la clase `Space`, crea un nuevo método denominado `createStars` (*crearEstrellas*). Este método debería tener un parámetro de tipo `int`, llamado `number` (*número*), para especificar el número de estrella que queremos crear. No devuelve nada. El cuerpo del método debería —por ahora— estar vacío.

Ejercicio 7.11 Escribe un comentario para el nuevo método. (El comentario debería describir qué hace el método, y explicar para qué se usa el parámetro.)

Ejercicio 7.12 Inserta una llamada a este nuevo método en tu constructor de `Space`. 300 estrellas puede ser una buena cantidad con la que comenzar (aunque puedes experimentar más tarde con otros números y escoger uno que pienses que queda bien).

Ejercicio 7.13 Compila la clase `Space`. En este momento, no deberías ver ningún efecto (ya que nuestro método está vacío), pero la clase debería compilar sin problemas.

Consejo:

El bucle **for** es uno de los constructores de bucles en Java. Es especialmente bueno para iterar un número fijo de veces.

En el método `createStars`, vamos a escribir ahora código para pintar algunas estrellas sobre la imagen de fondo. El número exacto de estrellas se especifica en los parámetros del método.

Usaremos otro bucle para hacer esto: el bucle *for*.

Previamente, hemos visto el bucle *while* y el bucle *for-each*. El bucle *for* usa la misma palabra clave que el bucle *for-each* (`for`), pero tiene una estructura diferente. Es

```
for (inicialización; condición-bucle; incremento)
{
    cuerpo-bucle;
}
```

Se puede ver un ejemplo de este bucle en el método `addAsteroids` de la clase `Space`.

Ejercicio 7.14 Examina el método `addAsteroids` de la clase `Space`. ¿Qué hace?

Ejercicio 7.15 Mira el bucle `for` de ese método. En la cabecera del bucle, escribe la parte de `inicialización`, la `condición-del-bucle`, y el `incremento`. (Mira la definición del bucle `for` de arriba.)

La parte de inicialización de un bucle `for` se ejecuta exactamente una vez antes de que comience el bucle. Luego se comprueba la condición del bucle: si es true, se ejecuta el cuerpo del bucle. Finalmente, después de que el cuerpo del bucle ha sido ejecutado completamente, se ejecuta la sección de incremento de la cabecera del bucle. Después de esto, el bucle vuelve a empezar: la condición se vuelve a evaluar y, si es cierta, se ejecuta el bucle de nuevo. Esto continúa hasta que la condición del bucle es falsa. La inicialización no se ejecuta nunca de nuevo.

Un bucle `for` podría ser fácilmente reemplazado por un bucle `while`. Un bucle `while` es equivalente a una estructura de bucle `for` como la vista arriba así:

```
inicialización;
while (condición-bucle)
{
     cuerpo-bucle;
     incremento;
}
```

La estructura del bucle `while` mostrado aquí y la estructura del bucle `for` mostrado arriba hacen exactamente lo mismo. La principal diferencia es que, en el bucle `for`, la inicialización y el incremento se han movido a la cabecera. Esto coloca todos los elementos que definen la conducta del bucle en un sitio, y hace que los bucles sean más fáciles de leer.

El bucle `for` es especialmente práctico si sabemos al principio del bucle cuántas veces queremos ejecutar el bucle.

El ejemplo de bucle `for` encontrado en el método `addAsteroids` decía

```
for (int i = 0; i < count; i++)
{
     int x = Greenfoot.getRandomNumber(getWidth()/2);
     int y = Greenfoot.getRandomNumber(getHeight()/2);
     addObject(new Asteroid(), x, y);
}
```

Esto nos muestra un ejemplo típico de bucle `for`:

- La parte de inicialización declara e inicializa una variable del bucle. Esta variable se denomina a menudo `i`, y a menudo se inicializa a `0`.
- La condición del bucle comprueba si nuestra variable de bucle es todavía menor que un límite dado (aquí: `count`). Si lo es, el bucle continúa.
- La sección de incremento simplemente incrementa la variable del bucle.

Hay otras variaciones diferentes del bucle `for`, pero este ejemplo nos muestra un formato muy típico.

Ejercicio 7.16 En tu clase `Space`, reescribe el bucle `for` en `addAsteroids` como un bucle `while`. Asegúrate de que hace lo mismo que antes.

Ejercicio 7.17 Reescribe este método de nuevo con un bucle `for`, como estaba antes.

Ejercicio 7.18 Implementa el cuerpo del método `createStars` que creaste antes. Este método debería incluir lo siguiente:

- Recuperar la imagen de fondo del mundo.
- Usar un bucle `for` de forma similar al de `addAsteroids`. El limite de este bucle se recibe en el parámetro del método.
- En el cuerpo del bucle, generar coordenadas aleatorias `x` e `y`. Fijar el color a blanco y entonces pintar un óvalo relleno con un ancho y alto de dos píxeles.

¡Pruébalo! ¿Ves estrellas en tu mundo? Si todo ha ido bien, deberías.

Ejercicio 7.19 Crea estrellas con luminosidad aleatoria. Puedes hacer esto creando un número aleatorio entre 0 y 255 (el rango legal de valores RGB para los colores) y creando un nuevo objeto `Color` usando el mismo valor aleatorio para los tres componentes de color (rojo, verde y azul). Al usar el mismo valor para los tres componentes de color, nos aseguramos que el color resultante será una sombra de un gris neutral. Usa este nuevo color aleatorio para pintar las estrellas. Asegúrate de generar un nuevo color con cada estrella.

Estos ejercicios son todo un desafío. Si tienes problemas, puedes mirar la solución. Tienes una implementación en la versión *asteroids-2* de este escenario. También puedes ignorar esta sección por ahora, continuar con las tareas siguientes primero, y volver aquí más tarde.

7.3 Girando

En la sección previa, hemos dedicado mucho esfuerzo a la apariencia. Usamos un bucle `for` para crear las estrellas en el fondo. Esto ha sido un gran trabajo para un efecto pequeño. Sin embargo, conocer el bucle `for` nos vendrá muy bien más tarde.

Ahora, queremos conseguir una funcionalidad real: queremos hacer que el cohete se mueva. El primer paso es hacer que gire cuando las teclas flecha izquierda o derecha sean pulsadas en el teclado.

Ejercicio 7.20 Examina la clase `Rocket`. Busca el código que maneja la entrada de teclado. ¿Cuál es el nombre del método que tiene este código?

Ejercicio 7.21 Añade una sentencia que haga que el cohete rote a la izquierda cuando se pulsa la tecla «izquierda». En cada ciclo del método `act`, el cohete debería rotar 5 grados. Puedes usar los métodos `getRotation` (*obténRotación*) y `setRotation` (*fijaRotación*) de la clase `Actor` para conseguir esto.

Ejercicio 7.22 Añade una sentencia que haga que el cohete rote a la derecha cuando se pulsa la tecla «derecha». ¡Pruébalo!

Si has conseguido completar con éxito los ejercicios, tu cohete debería ser capaz de girar ahora cuando pulsas las teclas de flechas. Como dispara en la dirección que mira, también puede disparar en todas las direcciones.

El siguiente reto es hacer que se mueva hacia delante.

7.4 Volando hacia delante

Nuestra clase `Rocket` es una subclase de la clase `SmoothMover`, que ya hemos visto en el capítulo previo. Esto significa que mantiene un vector de movimiento que determina su movimiento, y que tiene un método `move()` que hace que se mueva según este vector.

El primer paso va a ser usar este método `move()` .

Ejercicio 7.23 En el método `act` de `Rocket`, añade una llamada al método `move()` (heredado de `SmoothMover`). Pruébalo. ¿Qué observas?

Añadir una llamada a `move()` desde nuestro método `act` es un primer paso importante, pero no consigue mucho por sí mismo. Causa que el cohete se mueva según su vector de movimiento, pero como no hemos iniciado ningún movimiento, este vector tiene ahora una longitud cero, por lo que no se produce ningún movimiento.

Para cambiar esto, introduzcamos primero una pequeña deriva automática, de forma que el cohete comience con algún movimiento inicial. Esto hará que el juego sea más interesante, porque impide a los jugadores estar quietos durante mucho tiempo.

Ejercicio 7.24 Añade una pequeña deriva al movimiento inicial del cohete en su constructor. Para hacer esto, crea un nuevo vector con algún valor arbitrario para la dirección y una longitud pequeña (yo usé 0.3 para mi versión) y usa entonces el método `addForce` de `SmoothMover` con este vector como parámetro para añadir fuerza al cohete. (Asegúrate de usar un `int` como tu primer parámetro en el constructor de `Vector`, para usar el constructor correcto.)

Pruébalo. Si todo ha ido bien, el cohete debería ir a la deriva él mismo cuando el escenario comienza. No hagas que esta deriva inicial sea muy rápida. Experimenta hasta que consigas un movimiento inicial agradable y lento.

A continuación, queremos añadir controles de movimiento para el jugador. El plan es que al pulsar la tecla «flecha arriba», se encienda el propulsor del cohete y nos mueva hacia delante.

Para la otra entrada de teclado, hemos usado un código con el siguiente patrón:

```
if (Greenfoot.isKeyDown(«left»))
{
     setRotation(getRotation() - 5);
}
```

Para el movimiento hacia delante, necesitamos un patrón un poco diferente. La razón es que, para la rotación mostrada aquí, necesitamos actuar sólo si la tecla se está pulsando.

El movimiento hacia delante es diferente: cuando pulsamos la tecla «arriba» para movernos, queremos cambiar la imagen del cohete para mostrar el motor del cohete lanzando llamas. Cuando liberamos la tecla, la imagen debe volver a la imagen normal. Por tanto, necesitamos un patrón de código como el que sigue:

```
cuando tecla “arriba”  está pulsada:
     cambia la imagen para mostrar motor con llamas;
     añade movimiento;

cuando tecla “arriba” es liberada:
     vuelve a la imagen normal;
```

Mostrar las imágenes es bastante fácil. El escenario ya contiene dos imágenes del cohete para esto: *rocket.png* y *rocketWithThrust.png*. Ambas imágenes se cargan en los campos al principio de la clase `Rocket`.

Como necesitamos reaccionar en ambos casos, cuando la tecla «arriba» está pulsada y cuando no está pulsada, definiremos y llamaremos a un método separado para manejar esta funcionalidad.

En `checkKeys`, podemos insertar la siguiente llamada a método:

```
ignite(Greenfoot.isKeyDown(«up»))
```

Podemos a continuación escribir un método llamado *ignite* (*encender*) que haga lo siguiente:

- Recibe un parámetro booleano (digamos `boosterOn`, *propulsorOn*) que indica si el propulsor está encendido (on) o apagado (off).
- Si el propulsor está encendido, entonces fija la imagen a *rocketWithThrust.png* y usa `addForce` para añadir un nuevo vector. Este vector debería obtener su dirección de la rotación actual del cohete (`getRotation()`) y tener una longitud constante y pequeña (digamos, 0.3).
- Si el propulsor no está encendido, fija la imagen a *rocket.png*.

Ejercicio 7.25 Añade la llamada al método `ignite` en tu método `checkKeys`, exactamente como se ha mostrado arriba.

Ejercicio 7.26 Define un esqueleto de método (un método con un cuerpo vacío) para el método `ignite`. Este método debería tener un parámetro booleano, y un tipo de vuelta void. Asegúrate de escribir un comentario. ¡Pruébalo! El código debería compilar (pero no hará nada aún).

Ejercicio 7.27 Implementa el cuerpo del método `ignite`, como se bosquejó en las viñetas anteriores.

Para la implementación de nuestro método `ignite`, es aceptable si la imagen es fijada cada vez que el método se llama, incluso aunque esto no sea necesario (p. ej., si el propulsor está desactivado, y estaba desactivado la última vez, no necesitaríamos fijar la imagen de nuevo ya que no ha cambiado). Fijar la imagen incluso cuando no es estrictamente necesario tiene una sobrecarga muy pequeña, y por tanto no es crucial evitarlo.

Una vez que hayas completado estos ejercicios, has alcanzado una versión donde puedes volar con tu cohete y disparar a los asteroides.

Puedes encontrar una versión de proyecto que implementa los ejercicios presentados hasta ahora en *asteroids-2* dentro de los escenarios del libro.

7.5 Colisionando con asteroides

El fallo más obvio de nuestro juego de asteroides ahora mismo es que podemos volar a través de los asteroides. Esto hace que el juego no sea un gran reto, ya que no podemos perder. Arreglaremos esto ahora.

La idea es que nuestro cohete debería explotar cuando choque con un asteroide. Si hiciste los ejercicios anteriores de este capítulo, entonces ya has visto que tenemos una clase `Explosion` totalmente funcional y disponible en nuestro proyecto. Simplemente colocaremos una explosión en el mundo que ofrecerá un efecto adecuado.

Por tanto, una descripción aproximada de la tarea que tenemos que resolver es:

```
if (hemos colisionado con un asteroide) {
      quita el cohete del mundo;
      coloca una explosión en el mundo;
      muestra la puntuación final (game over);
}
```

Antes de empezar a resolver estas subtareas, vamos a preparar nuestro código fuente para implementar esta tarea, como hicimos antes con la otra funcionalidad. Seguimos la misma estrategia que antes: como esto es una tarea separada, la pondremos en un método separado, para mantener el código bien estructurado y fácilmente leíble. Normalmente comenzarás la implementación de una nueva funcionalidad de esta forma. El siguiente ejercicio lo hace.

Concepto:

Greenfoot proporciona varios métodos para **detectar colisiones**. Están en la clase Actor.

Ejercicio 7.28 Crea un nuevo esqueleto de método (un método con un cuerpo vacío) en la clase `Rocket` para comprobar colisiones con asteroides. Llámalo `checkCollision` (*compruebaColisión*). Este método puede ser `private` y no necesita devolver ningún valor ni recibir parámetros.

Ejercicio 7.29 En el método `act` de `Rocket`, añade una llamada al método `checkCollision`. Asegúrate de que tu clase compila y se ejecuta de nuevo.

La primera subtarea es comprobar si hemos colisionado con un asteroide. La clase `Actor` de `Greenfoot` contiene varios métodos para comprobar colisiones con diferente funcionalidad.

El Apéndice C presenta un resumen de los diferentes métodos de detección de colisiones y su funcionalidad. Éste podría ser un buen momento para verlos rápidamente. En algún momento, deberías familiarizarte con todos los métodos de detección de colisiones.

Para nuestros propósitos, `getIntersectingObjects` (*obténObjetosIntersecan*) podría valer. Dos objetos intersecan si cualquiera de los píxeles de sus imágenes intersecan. Esto es justo lo que necesitamos.

Concepto:

El **cuadro delimitador** de una imagen es el rectángulo que envuelve una imagen.

Hay un pequeño problema: los píxeles transparentes en las imágenes de actores.

Las imágenes en Greenfoot son siempre rectángulos. Cuando ves imágenes no rectangulares, tales como el cohete, es porque algunos píxeles de la imagen son *transparentes* (invisibles; no contienen ningún color). Para el propósito de nuestro programa, sin embargo, estos píxeles son aún parte de la imagen.

La Figura 7.2 muestra las imágenes del cohete y el asteroide con sus *cuadros delimitadores*. El cuadro delimitador es el borde de la imagen actual. (La imagen del cohete es un poco más grande que lo que parece necesario para hacerla del mismo tamaño que la segunda imagen del cohete, *rocketWithThrust*, que muestra la llama en el área actualmente vacía.)

En la Figura 7.2, las imágenes intersecan, incluso si sus partes visibles no se tocan. Los métodos de detección de colisiones reportarán esto como una intersección. Estos métodos trabajan con el cuadro delimitador, y no se fijan en las partes no transparentes de la imagen.

Como resultado, nuestro cohete «contactará» con un asteroide incluso aunque, en la pantalla, parezca que está aún a una pequeña distancia de ellos.

Para nuestro juego de asteroides, escogemos ignorar esto. En primer lugar, la distancia es pequeña, así que a menudo los jugadores no lo notarán. En segundo lugar, es bastante fácil inventarse una justificación que explique este efecto («volar demasiado cerca de un asteroide destruye tu cohete por la fuerza gravitatoria»).

Algunas veces, estaría bien comprobar si las partes actualmente visibles (no transparentes) de una imagen intersecan. Esto es posible, pero mucho más difícil. No lo discutiremos aquí.

Ahora que hemos decidido seguir con la intersección, vamos a revisar los métodos de `Actor` de nuevo. Hay dos métodos para comprobar la intersección de objetos. Sus signaturas son

```
List getIntersectingObjects(Class cls)
Actor getOneIntersectingObject(Class cls)
```

Ambos métodos aceptan un parámetro de tipo `Class` (que significa que podemos comprobar si interseca con un objeto de una clase específica si queremos). La diferencia es que un método devuelve todos los objetos con los que estamos actualmente intersecando, mientras que el otro sólo devuelve un único objeto. En caso de que intersequemos con más de un objeto, el segundo método escoge aleatoriamente uno de ellos y lo devuelve.

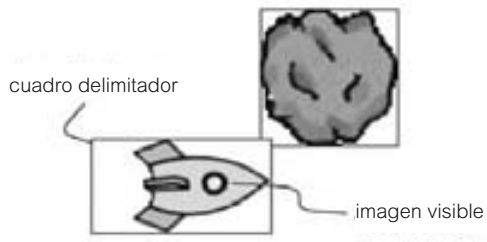


Figura 7.2
Dos imágenes de actores y sus cuadros delimitadores

Para nuestro propósito, el segundo método es suficientemente bueno. Ahora mismo es indiferente si chocamos con un asteroide, o con dos simultáneamente. El cohete explotará en ambos casos de igual forma. La única pregunta que nos importa es: ¿hemos intersecado con algún asteroide?

Por tanto, usaremos el segundo método. Como devuelve un `Actor`, en vez de una `List`, es un poco más fácil de usar. Devolverá un actor si tenemos una intersección, o null si no estamos intersecando con ningún asteroide. Podemos comprobar si devuelve un valor null para ver si hemos chocado con algo:

```
Actor a = getOneIntersectingObject(Asteroid.class);
if (a != null)
{
    ...
}
```

Ejercicio 7.30 Añade una comprobación de una intersección con un asteroide, de forma similar al mostrado aquí, en tu método `checkCollision`.

Ejercicio 7.31 Añade código al cuerpo de la sentencia `if` que añada una explosión en el mundo en la posición actual del cohete, y elimine el cohete del mundo. (Para hacer esto, necesitas usar el método `getWorld()` para acceder a sus métodos para añadir y eliminar objetos del mundo.)

Para el último ejercicio de arriba, podemos usar nuestros propios métodos `getX()` y `getY()` para recuperar nuestra posición actual. Podemos usar esto como las coordenadas para colocar la explosión.

Un intento para resolver esto podría ser algo así:

```
World world = getWorld();
world.removeObject(this); // elimina el cohete del mundo
world.addObject(new Explosion(), getX(), getY());
```

Este código parece razonable a primer vista, pero no funcionará.

Ejercicio 7.32 Prueba el código mostrado arriba. ¿Funciona? ¿En qué punto algo va mal, y cuál es el mensaje de error?

La razón por la que no funciona es que estamos llamando a los métodos `getX()` y `getY()` después de eliminar el cohete del mundo. Cuando un actor se elimina del mundo, ya no tiene coordenadas nunca más —sólo tiene coordenadas cuando está en el mundo. Por tanto, las llamadas a los métodos `getX()` y `getY()` fallan en este ejemplo.

Esto se puede arreglar fácilmente cambiando las dos últimas líneas de código: inserta primero la explosión y entonces elimina el cohete del mundo.

Ejercicio 7.33 Éste es un ejercicio muy avanzado, y puede que quieras saltártelo inicialmente, y volver a él más tarde.

La explosión usada aquí es una explosión bastante simple. Es bastante buena por ahora, pero si quieres crear juegos con mejores efectos, se puede mejorar. Una forma más sofisticada de mostrar explosiones se presenta en un video tutorial de Greenfoot, que está disponible en el sitio web de Greenfoot:

`http://www.greenfoot.org/doc/videos.html`

Crea una explosión similar para tu cohete.

7.6 Conversión de tipos

Nuestro juego es bastante jugable. Puedes haber notado que no funciona el marcador (lo veremos más tarde), y que no pasa nada cuando pierdes. A continuación, añadiremos un gran letrero de «Game Over» al final del juego, cuando el cohete explota.

Esto es fácil de hacer: ya tenemos una clase `ScoreBoard` en el proyecto que podemos usar.

Ejercicio 7.34 Crea un objeto de la clase `ScoreBoard` y colócalo en el mundo.

Ejercicio 7.35 Examina el código fuente de la clase `ScoreBoard`. ¿Cuántos constructores tiene? ¿Cuál es la diferencia entre ellos?

Ejercicio 7.36 Modifica la clase `ScoreBoard`: cambia el texto que muestra; cambia el color del texto; cambia los colores del fondo y del borde; cambia el tamaño de la fuente, de forma que tu texto quepa bien; cambia el ancho del marcador para que se ajuste a tu texto.

Como has visto, el marcador incluye un texto «Game over» y la puntación final (aunque la puntación no se cuenta todavía, pero nos preocuparemos de eso más tarde).

La clase `Space` ya tiene un método, denominado `gameOver`, cuyo propósito es crear y mostrar un marcador.

Ejercicio 7.37 Busca y examina el método `gameOver` en la clase `Space`. ¿Qué hace la implementación actual?

Ejercicio 7.38 Implementa el método `gameOver`. Debería crear un nuevo objeto `ScoreBoard`, usando el constructor que espera un parámetro `int` para la puntuación. Por ahora, usa 999 como la puntuación —arreglaremos esto más tarde—. Coloca el marcador en el mundo, exactamente en el medio.

Ejercicio 7.39 Una vez implementado, prueba tu método `gameOver`. Recuerda: puedes llamar a métodos de la clase del mundo pinchando con el botón derecho en el título del mundo (mira la Figura 6.3).

Ejercicio 7.40 ¿Cómo puedes asegurarte de que el marcador está colocado justo en el centro sin poner a fuego la posición (esto es, sin usar los números 300 y 200 directamente como coordenadas)? Pista: usa el ancho y el alto del mundo. Implementa esto en tu escenario.

Por tanto, parece que la mayor parte del trabajo ya nos la han dado preparada. Ahora sólo necesitamos llamar al método `gameOver` cuando queramos terminar el juego.

El lugar en tu código donde queremos que el juego termine es en nuestro método `checkCollision` del cohete: si detectamos una colisión, el cohete podría explotar (hemos hecho eso) y el juego terminaría.

Ejercicio 7.41 Añade una llamada al método `gameOver` en tu método `checkCollision`. Compila. ¿Qué observas? Muy probablemente, verías un error —¿cuál es el mensaje de error?

Simplemente el añadir la llamada a `gameOver` nos ha creado un problema. Esto es un problema bastante básico, y debemos analizarlo con más detalle. Veamos el código que tenemos hasta ahora, asumiendo que acabamos de añadir una llamada al método `gameOver` en nuestro método `checkCollision` (Código 7.1). Este código no compilará.

Código 7.1
Un primer intento de llamar al método `gameOver`

```
private void checkCollision()
{
     Actor a = getOneIntersectingObject(Asteroid.class);
     if (a != null)
     {
         World world = getWorld();
         world.addObject(new Explosion(), getX(), getY());
         world.removeObject(this); // elimina el cohete del mundo
         world.gameOver(); // error: esto no funcionará
     }
}
```

Cuando intentamos compilar este código, obtenemos un error que dice

cannot find symbol—method gameOver()

Este mensaje está intentando decirnos que el compilador no puede encontrar un método con este nombre. Sabemos, sin embargo, que existe este método en nuestra clase `Space`. También sabemos que la llamada `getWorld()` usada aquí nos da una referencia a nuestro objeto `Space`. ¿Cuál es entonces el problema?

El problema yace en el hecho de que la compilación no es tan lista como nos gustaría. El método `getWorld()` está definido en la clase `Actor`, y su signatura es ésta:

```
World getWorld()
```

Podemos ver que esta vez devolverá un objeto de tipo `World`. El mundo que devuelve en nuestro caso es de tipo `Space`.

Concepto:

Los objetos pueden tener **más de un tipo:** el tipo de su propia clase, y el tipo de la superclase de la clase.

Esto no es una contradicción. Nuestro objeto puede ser de tipo `World` y de tipo `Space` a la vez, porque `Space` es una subclase de `World` (`Space` *es* un `World`; también podemos decir que el tipo `Space` es un *subtipo* del tipo `World`).

El error viene de la diferencia entre los dos: `gameOver` se define en la clase `Space`, mientras que `getWorld` nos da un resultado de tipo `World`. El compilador sólo mira el tipo de vuelta del método al que llamamos (`getWorld`). Debido a esto, el compilador busca el método `gameOver` sólo allí, y no lo encuentra. Por esto obtenemos un mensaje de error.

Para resolver este problema, necesitamos decirle al compilador explícitamente que este mundo que estamos obteniendo es en realidad del tipo `Space`. Podemos hacer esto usando una conversión de tipos.

```
Space space = (Space) getWorld();
```

Concepto:

La **conversión de tipos (Casting)** es la técnica de especificar un tipo más preciso para nuestro objeto que el que el compilador conoce.

La conversión de tipos (*Casting*) es la técnica que usamos para decirle al compilador de forma más precisa el tipo de nuestro objeto que lo que puede averiguar él mismo. En nuestro caso, el compilador puede averiguar que el objeto que devuelve `getWorld` es de tipo `World`, y nosotros le decimos que en realidad es de la clase `Space`. Hacemos esto escribiendo el nombre de la clase (`Space`) entre paréntesis antes de la llamada al método. Una vez que hemos hecho esto, podemos llamar a los métodos definidos en `Space`:

```
space.gameOver();
```

Merece la pena destacar que la conversión de tipos no cambia el tipo del objeto. Nuestro mundo en realidad está formado por objetos de tipo `Space`. El problema simplemente viene de que el compilador no lo sabe, y simplemente le estamos dando esta información adicional al compilador.

De vuelta a nuestro método `checkCollision`. Una vez que hemos convertido nuestro mundo a `Space` y lo hemos almacenado en una variable de tipo `Space`, podemos llamar a todos sus métodos; a los definidos en `Space` y a los definidos en `World`. Por tanto, nuestras llamadas a `addObject` y `removeObject` deberían seguir funcionando, y la llamada a `gameOver` debería funcionar también.

Ejercicio 7.42 Implementa la llamada al método `gameOver`, usando la conversión de tipos del objeto `World` a `Space`, como hemos visto aquí. Pruébala. Debería funcionar ahora, y el marcador debería incrementarse cuando explota el cohete.

Ejercicio 7.43 ¿Qué ocurre cuando usas de forma incorrecta una conversión de tipos? Prueba a convertir el objeto mundo, digamos a `Asteroid` en vez de a `Space`. ¿Funciona? ¿Qué observas?

El trabajo que hemos hecho hasta ahora ha conseguido mostrar nuestro letrero «Game Over» (aún con un marcador incorrecto). Dejaremos la realización del marcador como un ejercicio al final de este capítulo. Si quieres arreglar esto ahora, puedes querer saltar al principio de los ejercicios finales del capítulo y ver esto primero. Aquí, veremos a continuación el tema de las ondas de protones.

7.7 Añadiendo propulsión: la onda de protones

Nuestro juego está yendo bastante bien. Lo último que tenemos que ver en detalle en este capítulo es cómo añadir una segunda arma: la onda de protones. Esto debería dar al juego un poco más de variedad. La idea es ésta: nuestra onda de protones, una vez liberada, se irradia hacia fuera de nuestra nave, dañando o destruyendo a todos asteroides que encuentra en su camino. Como funciona en todas las direcciones a la vez, es un arma mucho más potente que nuestras balas. Para el juego, probablemente deberíamos restringir con qué frecuencia se puede usar, de forma que el juego no sea demasiado fácil.

Ejercicio 7.44 Ejecuta tu escenario. Coloca una onda de protones en tu escenario —¿Qué observas?

El ejercicio muestra que tenemos un actor onda de protones, que muestra la onda a tamaño completo. Sin embargo, la onda no se mueve, ni desaparece, ni causa ningún daño a los asteroides.

Nuestra primera tarea será hacer que la onda crezca. Comenzará siendo muy pequeña, y luego crecerá hasta que alcance el tamaño completo que hemos visto.

Ejercicio 7.45 Examina el código fuente de la clase `ProtonWave`. ¿Cuáles son los métodos que ya hay?

Ejercicio 7.46 ¿Cuál es el propósito de cada método? Revisa los comentarios de cada método y expándelos para añadir una explicación más detallada.

Ejercicio 7.47 Intenta explicar qué hace el método `initializeImages` y cómo funciona. Explícalo por escrito, usando diagramas si quieres.

7.8 Creciendo la onda

Hemos visto que la clase `ProtonWave` tiene un método —`initializeImages`— que crea 30 imágenes de diferentes tamaños y los almacena en un array (Código 7.2). Este array, denominado código fuente de, guarda la imagen más pequeña en el índice 0, y la más grande en el índice 29 (mira la Figura 7.3). Las imágenes se crean cargando una imagen base (*wave.png*) y luego, en bucle, creando copias de esta imagen y escalándola a tamaños diferentes.

Código 7.2
Inicializando las imágenes para la onda de protones

```
/**
*  Crea las imágenes para expandir la onda.
*/
public static void initializeImages()
{
     if(images == null)
     {
          GreenfootImage baseImage = new GreenfootImage("wave.png");
          images = new GreenfootImage[NUMBER_IMAGES];
          int i = 0;
          while (i < NUMBER_IMAGES)
          {
              int size = (i+1) * (baseImage.getWidth() / NUMBER_IMAGES);
              images[i] = new GreenfootImage(baseImage);
              images[i].scale(size, size);
              i++;
          }
     }
}
```

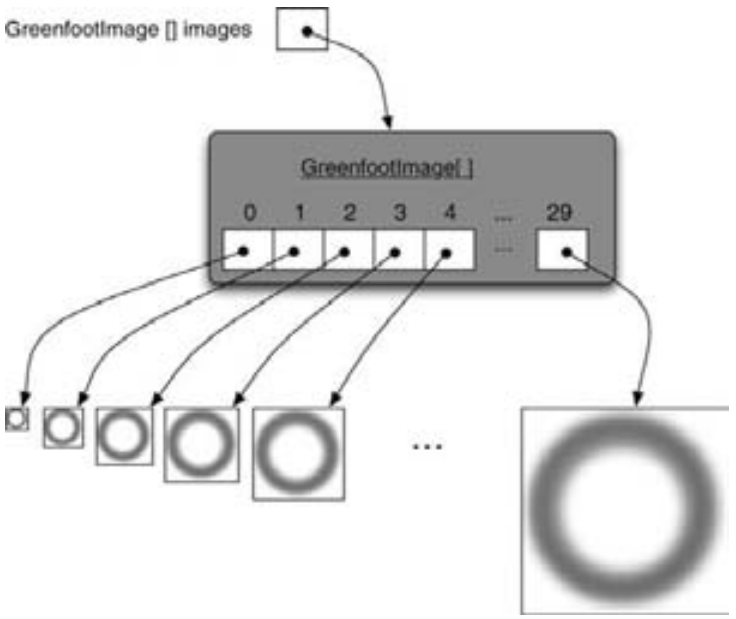


Figura 7.3
Un array de imágenes (faltan algunas por limitaciones de espacio)

Este método usa el método `scale` de la clase `GreenfootImage` para realizar el escalado. También usa un bucle `while` para iterar. Sin embargo, éste es un ejemplo donde un bucle `for`, que vimos al principio de este capítulo, podría ser adecuado.

Ejercicio 7.48 Reescribe el método `initializeImages` para que use un bucle `for` en vez de un bucle `while`.

En la práctica, no es muy importante qué bucle uses en este caso. (Lo hemos cambiado aquí principalmente para ganar más práctica en escribir bucles `for`.) Éste es, sin embargo, un caso donde un bucle `for` es una buena elección, ya que tenemos un número conocido de iteraciones (el número de imágenes) y podemos hacer buen uso del contador del bucle para calcular los tamaños de las imágenes. La ventaja sobre el bucle `while` es que el bucle `for` tiene todos los elementos del bucle (inicialización, condición e incremento) juntos en la cabecera, por lo que podemos usarlo sin peligro de que nos olvidemos alguna de las partes.

El campo `images` y el método `initializeImages` son `static` (usan la palabra clave `static` en su definición). Como mencionamos brevemente en el Capítulo 7, esto significa que el campo `images` se almacena en la clase `ProtonWave`, no en cada instancia individual. Como resultado, todos los objetos que creemos de esta clase pueden compartir este conjunto de imágenes, y no necesitamos crear un conjunto separado para cada objeto. Esto es mucho más eficiente que usar una imagen separada cada vez.

Copiar y escalar estas imágenes tarda bastante (entre una décima de segundo y medio segundo en un ordenador medio). Esto no parece mucho, pero es bastante como para que introduzcamos un retraso visible, molesto, cuando lo hacemos a la mitad de una partida del juego. Para resolver esto, el código de este método se puede incluir dentro del cuerpo de una sentencia `if`:

```
if (images == null)
{
    ...
}
```

Esta sentencia `if` asegura que la parte principal de este método (el cuerpo de la sentencia `if`) se ejecuta sólo una vez. La primera vez, `images` será `null`, y el método lo ejecuta entero. Esto inicializará el campo `images` a algo diferente de `null`. A partir de aquí, la comprobación de la sentencia `if` será todo lo que se ejecute, y el cuerpo será saltado. El método `initializeImages` se llama en realidad cada vez que una onda de protones es creada (desde el constructor), pero sólo la primera vez que es llamada tiene que hacer este trabajo pesado [1].

Ahora que tenemos una idea precisa del código y de los campos que ya existen, podemos finalmente ponernos a trabajar y conseguir que ocurra algo.

Lo que queremos hacer es lo siguiente:

- Queremos que la onda comience con la imagen más pequeña.

[1] El método es en realidad llamado la primera vez desde el constructor de `Space`, por lo que se ejecuta incluso antes de que la primera onda de protones se haya creado. Esto también evita un retraso cuando se crea la primera onda de protones. La llamada se incluye en el constructor de la onda de protones sólo por razones de seguridad: si esta clase se utiliza alguna vez en otro proyecto, y no se llama a este método antes, todo seguirá funcionando.

- En cada paso del método `act`, queremos que la onda crezca (que muestre la siguiente imagen más grande).
- Después de que hayamos mostrado la imagen más grande, la onda debería desaparecer (ser eliminada del mundo).

Los siguientes ejercicios consiguen esto.

Ejercicio 7.49 En el constructor de la clase `ProtonWave`, fija la imagen a la imagen más pequeña. (Puedes usar `images[0]` como parámetro del método `setImage`.)

Ejercicio 7.50 Crea un campo de instancia llamado `imageCount` (*cuentalmágenes*) de tipo `int`, e inicialízalo a 0. Usaremos este campo para contar las imágenes. El valor actual es el índice de la imagen actualmente mostrada.

Ejercicio 7.51 Crea el esqueleto de un método nuevo privado llamado `grow` (*crece*). Este método no tiene parámetros y no devuelve nada.

Ejercicio 7.52 Llama al método `grow` desde tu método `act`. (Incluso aunque esto no hará nada todavía.)

Casi hemos terminado. La única cosa que nos queda es implementar el método `grow`. La idea, a grandes rasgos, es ésta:

```
muestra la imagen en el índice imageCount;
incrementa imageCount;
```

También tendremos que añadir una sentencia `if` que compruebe primero si `imageCount` ha superado el número de imágenes. En ese caso, eliminaremos la onda de protones del mundo y ya habremos terminado.

Ejercicio 7.53 Implementa el método `grow` según las directrices dadas arriba.

Ejercicio 7.54 Prueba tu onda de protones. Si creas interactivamente una onda de protones y la colocas en el mundo mientras tu escenario se está ejecutando, deberías ver el efecto de expansión de la onda.

Ejercicio 7.55 Añade algún sonido. Se incluye un fichero de sonido llamado `proton.wav` en el escenario —simplemente reprodúcelo. Puedes colocar una sentencia para reproducir el sonido en el constructor de la onda de protones.

Ahora que tenemos la onda de protones funcionando, podemos equipar nuestro cohete para liberarla.

Ejercicio 7.56 En la clase `Rocket`, crea el esqueleto de un método llamado `startProtonWave` (*comienzaOndaProtones*) sin parámetros. ¿Necesita devolver algo?

Ejercicio 7.57 Implementa este método: debería colocar un objeto nuevo onda de protones en el mundo, en las coordenadas actuales del cohete.

Ejercicio 7.58 Llama a este nuevo método desde el método `checkKeys` cuando se pulsa la tecla «z». Pruébalo.

Ejercicio 7.59 Notarás rápidamente que la onda de protones se puede liberar con demasiada frecuencia ahora. Para disparar las balas, se definió un retraso en la clase `Rocket` (usando la constante `gunReloadTime` y el campo `reloadDelayCount`). Estudia este código e implementa algo parecido para la onda de protones. Prueba con diferentes valores de retardo hasta que encuentres uno que parezca razonable.

7.9 Interaccionando con objetos al alcance

Tenemos ahora una onda de protones que podemos liberar cuando pulsamos un botón. El problema que queda es: esta onda de protones no hace nada a los asteroides.

Ahora deseamos añadir código que cause daño a los asteroides cuando se golpean por la onda de protones.

Ejercicio 7.60 Prepara la nueva funcionalidad: en la clase `ProtonWave`, añade el esqueleto de un método denominado `checkCollision` (*compruebaColisión*). El método no tiene parámetros y no devuelve nada. Llama a este método desde tu método `act`.

Ejercicio 7.61 El propósito de este nuevo método es comprobar si la onda toca un asteroide, y que lo dañe si se lo toca. Escribe el comentario del método.

Esta vez no queremos usar el método `getIntersectingObjects`, dado que las áreas invisibles de la imagen en las esquinas de la imagen de la onda del protón (dentro del cuadro delimitador, pero que no son parte del círculo azulado) son bastante grandes, y los asteroides serían destruidos mucho antes de que la onda pareciera que los ha alcanzado.

Esta vez, usaremos otro método de detección de colisiones, denominado `getObjectsInRange` (*obtén ObjetosEnAlcance*).

El método `getObjectsInRange` devuelve una lista de todos los objetos dentro de un radio dado del objeto llamante (mira la Figura 7.4). Su signatura es

```
List getObjectsInRange(int radius, Class cls)
```

Cuando le llamemos, podemos especificar la clase de objetos en que estamos interesados (como antes), y que también podemos especificar un radio (en celdas). El método devolverá entonces una lista de todos los objetos de la clase pedida que se encuentran dentro del radio alrededor del objeto llamante.

Para determinar qué objetos están dentro del rango, se usan los centros de cada objeto. Por ejemplo, un asteroide estaría en un rango 20 de un cohete si la distancia al centro del cohete es menor del ancho de 20 celdas. El tamaño de la imagen no es relevante para este método.

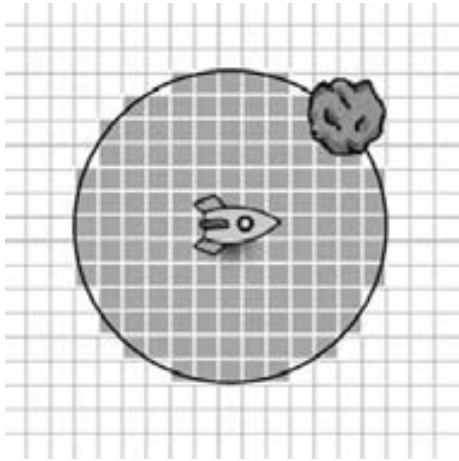


Figura 7.4
El alcance alrededor de un actor con un radio dado

Usando este método, podemos implementar nuestro método `checkCollision`.

Nuestra onda de protones tendrá imágenes de tamaño creciente. En cada ciclo del método `act`, podemos usar el tamaño de la imagen actual para determinar el rango de comprobación de nuestra colisión. Podemos averiguar el tamaño de la imagen actual haciendo las siguientes llamadas a métodos:

```
getImage().getWidth()
```

Podemos entonces usar la mitad de este tamaño como nuestro rango (ya que el rango se especifica con el radio, y no con el diámetro).

Ejercicio 7.62 En `checkCollision`, declara una variable local denominada *range (rango)* y asigna la mitad del ancho de la imagen actual a ella.

Ejercicio 7.63 Añade una llamada a `getObjectsInRange` que devuelva todos los asteroides dentro del rango calculado. Asigna el resultado a una variable de tipo `List<Asteroid>`. Recuerda que tienes también que añadir una sentencia `import` para el tipo `List`.

Estos ejercicios nos dan una lista de todos los asteroides en el rango de la onda de protones. Ahora queremos que haga algún daño a cada asteroide dentro del rango.

La clase `Asteroid` tiene un método denominado `hit` (*golpea*) que podemos usar para hacer esto. Este método ya se está usando para dañar el asteroide cuando es alcanzado por una bala, y lo podemos usar aquí de nuevo.

Podemos usar un bucle *for-each* para iterar por todos los asteroides de la lista que obtenemos de la llamada a `getObjectsInRange`. (Si no estás seguro de cómo escribir bucles *for-each*, vuelve a mirar la Sección 6.8.)

Ejercicio 7.64 Encuentra el método `hit` de la clase `Asteroid`. ¿Cuáles son sus parámetros? ¿Qué devuelve?

Ejercicio 7.65 La clase `ProtonWave` tiene una constante definida en la parte de arriba, denominada DAMAGE (*DAÑO*), que especifica cuánto daño debería causar. Encuentra la declaración de esta constante. ¿Cuál es su valor?

Ejercicio 7.66 En el método `checkCollision`, escribe un bucle *for-each* que itere por la lista de asteroides obtenidos de la llamada a `getObjectsInRange`. En el cuerpo del bucle, llama a `hit` en cada asteroide usando la constante DAMAGE para la cantidad de daño causado.

Una vez que hayas completado estos ejercicios, prueba. Si todo ha ido bien, deberías tener ahora una versión jugable de este juego que te permita disparar a los asteroides y también liberar ondas de protones para destruir muchos asteroides de una vez. Observarás que hacer que el tiempo de recarga para la onda de protones sea bastante grande, ya que el juego resulta demasiado fácil si podemos usar la onda muy a menudo.

Esta versión del juego, incluyendo todos los cambios hechos en las últimas secciones se puede encontrar en los proyectos del libro en *asteroids-3*. Puedes usar esta versión para compararla con tu propio escenario, o para buscar las soluciones si te bloqueas en uno de los ejercicios.

7.10 Otras mejoras

Estamos al final de la explicación detallada del desarrollo de este escenario en este capítulo. Hay, sin embargo, un gran número de posibles mejoras al juego. Algunas son bastante obvias, otras puede que te guste inventártelas tú mismo.

A continuación hay algunas sugerencias para trabajos avanzados, en forma de ejercicios. Muchos de ellos son independientes entre sí —no tienen que hacerse en un orden particular. Escoge aquellos que te interesen más, y realiza algunas extensiones que se te ocurran.

Ejercicio 7.67 Arregla el marcador para que cuente. Ya hemos visto antes que hay un contador en el marcador, pero no lo hemos usado aún. El contador está definido en la clase `Counter`, y un objeto contador se está creando en la clase `Space`. A grandes rasgos, tendrás que hacer lo siguiente: añade un método a la clase `Space` denominado con un nombre como `countScore` —esto debería añadir un marcador al contador, y llama a este nuevo método desde la clase `Asteroid` siempre que un asteroide sea golpeado (puede que quieras tener diferentes puntuaciones para dividir el asteroide o eliminar la última pieza).

Ejercicio 7.68 Añade nuevos asteroides cuando todos hayan sido eliminados. Puede ser que el juego tenga que volver a empezar con sólo dos asteroides, que cada vez que no haya asteroides, aparezcan nuevos, uno más cada vez. Así, en la segunda ronda, habrá tres asteroides, en la tercera ronda, cuatro, etc.

Ejercicio 7.69 Añade un contador de niveles. Cada vez que todos los asteroides son eliminados, aumentas de nivel. Puede que tengas más puntos en niveles más altos.

Ejercicio 7.70 Añade un sonido de fin de nivel. Deberías reproducirlo cada vez que se completa un nivel.

Ejercicio 7.71 Añade un indicador que muestre el estado de carga de la onda de protones, de forma que el jugador vea si está lista para usarse de nuevo. Se podría usar un contador, o algún tipo de representación gráfica.

Ejercicio 7.72 Añade un escudo. Cuando se despliega el escudo, permanece allí un tiempo breve fijo. Cuando el escudo está arriba, se le puede ver en la pantalla, y no nos dañan los asteroides que colisionen con nosotros.

Hay, por supuesto, incontables extensiones posibles. Invéntate algunas de tu cosecha, impleméntalas, y envía tus resultados a la Galería de Greenfoot.

7.11 Resumen de técnicas de programación

En este capítulo hemos trabajo para completar el juego de asteroides que recibimos inicialmente a medio hacer. Para hacer esto, nos hemos vuelto a encontrar con algunos constructores importantes que ya habíamos visto antes, incluyendo bucles, listas y detección de colisiones.

Hemos visto un nuevo estilo de bucle —el bucle `for`— y lo hemos usado para pintar estrellas, y generar las imágenes de la onda de protones. También hemos vuelto a ver el bucle *for-each* cuando hemos implementado la funcionalidad de la onda de protones.

Hemos usado varios métodos de detección de colisiones: `getOneIntersectingObject` y `getObjectsInRange`. Ambos tienen sus ventajas en ciertas situaciones. El segundo nos devolvía una lista de actores, de forma que tenemos que volver a tratar con listas.

Inicialmente es algo difícil comprender listas y bucles, pero son muy importantes en la programación, así que deberías revisar con cuidado estos aspectos si no te sientes cómodo al usarlos. Cuanto más los practiques, más fáciles serán. Después de usarlos un tiempo, te sorprenderá que los encontraras tan difíciles al principio.

Resumen de conceptos

- El **bucle for** es uno de los constructores de bucles en Java. Es especialmente bueno para iterar un número fijo de veces.
- Greenfoot proporciona varios métodos para **detectar colisiones**. Están en la clase Actor.
- El **cuadro delimitador** de una imagen es el rectángulo que delimita esa imagen.
- Los objetos pueden ser de **más de un tipo**: el tipo de su propia clase y el tipo de la superclase de la clase.
- La **conversión de tipos** es la técnica para especificar un tipo más preciso para nuestro objeto que el que el compilador conoce.

Términos en inglés

Inglés	Español
casting	conversión de tipos

INTERMEDIO

2 La competición Greeps

Nos tomaremos un poco de tiempo para avanzar en este segundo intermedio —un descanso en la secuencia de capítulos para hacer algo diferente. Esta vez, veremos «Greeps» — una competición de programación.

Los Greeps son criaturas alienígenas. Y ¡han venido a la Tierra! Una de las cosas importantes que debes saber sobre los Greeps es que les gustan los tomates. Han aterrizado con su nave espacial y pululan buscando y recogiendo tomates (Figura I2.1).

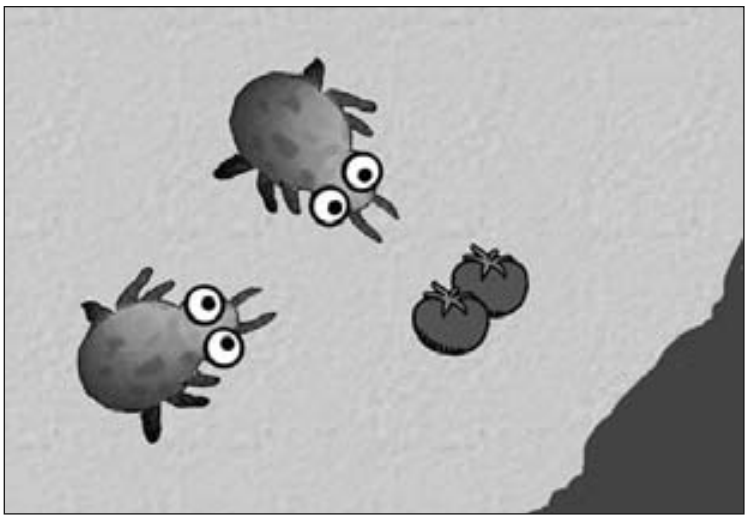


Figura I2.1
Dos Greeps
a la caza de tomates

El resto en esta competición de programación será programar tus Greeps para que encuentren y recojan tomates tan rápido como sea posible. Tendrás tiempo limitado, y cada tomate que consigas llevar a la nave espacial te dará un punto.

Puedes hacer este proyecto como una competición contra un amigo que programe su propio Greeps, o puedes hacerlo para un concurso de todo un grupo de programadores como una clase. Si lo haces solo, puedes mandar tu solución a la Galería de Greenfoot y compararla con los resultados de otros allí [1]. O podrías hacerlo tú solo simplemente para divertirte —en cualquier caso, debería ser un reto interesante.

---

[1] Si envías el escenario Greeps a la Galería de Greenfoot, por favor, no incluyas el código fuente. Queremos mantener este proyecto como un reto a futuros programadores y no queremos que sea demasiado fácil encontrar soluciones de otros.

I2.1 Cómo comenzar

Para comenzar, abre el escenario *greeps* de la carpeta *book-scenarios*. Ejecuta este escenario.

Verás que una nave espacial aterriza en un área con tierra y agua. Los Greeps abandonarán la nave espacial y comenzarán a buscar pilas de tomates (que suelen encontrarse en varios sitios en esta área). Los Greeps son animales terrestres —no pueden andar en el agua—. (De hecho, son tan sensibles al agua que se disolverían muy rápido dentro de ella, así que no lo intentes.)

Cuando pruebes este escenario, verás rápidamente que los Greeps no se comportan de forma muy inteligente. Se dirigen en una dirección aleatoria desde el barco, pero cuando llegan al borde del agua, simplemente permanecen allí, porque no saben ir hacia atrás.

Tu tarea será programar los Greeps para que usen una estrategia algo más inteligente, de forma que encuentren los tomates y los lleven al barco.

Hay algunos hechos sobre los Greeps que sería bueno que conocieras:

- Hay 20 Greeps en la nave espacial. Saldrán tras aterrizar para comenzar su trabajo. No puedes coger a ninguno de ellos.
- Los Greeps pueden transportar un tomate a sus espaldas, pero no pueden cargar tomates en su propia espalda. ¡Sólo pueden cargar un tomate en la *espalda de otro Greep*! Esto significa que dos de ellos tienen que estar en la pila de tomates al mismo tiempo para coger un tomate.

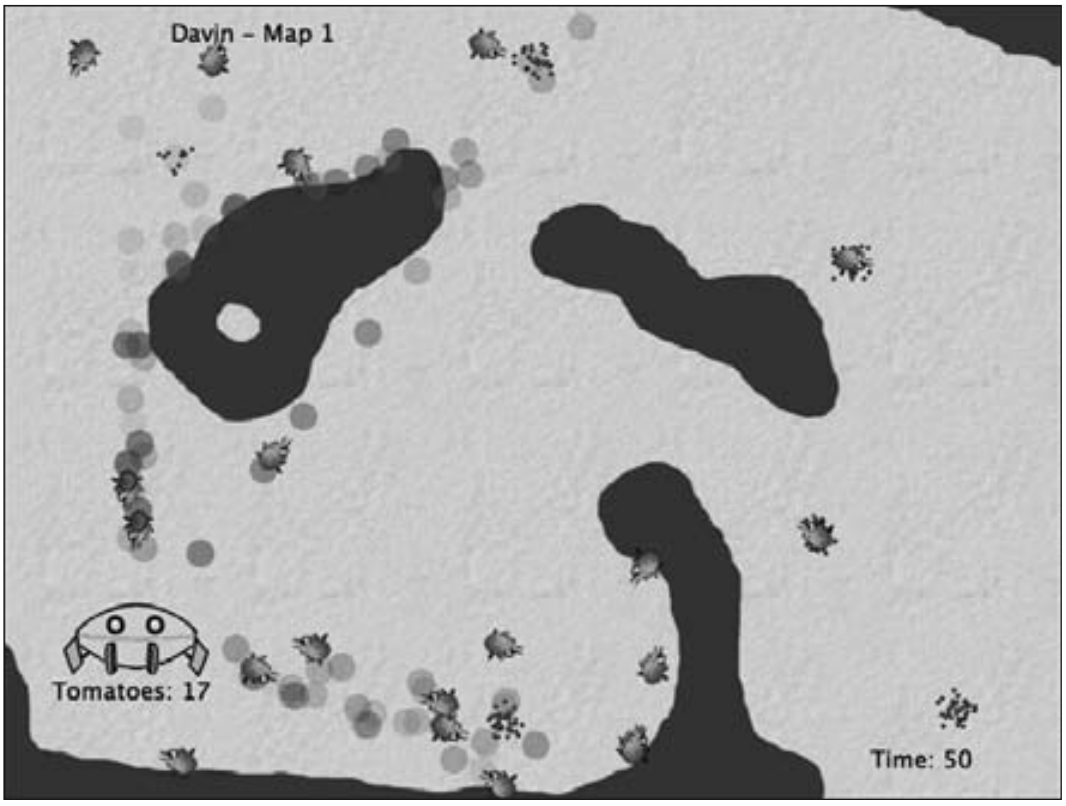


Figura I2.2
Una tribu de Greeps usando gotas de pintura

- Los Greeps no pueden hablar ni comunicarse verbalmente de ninguna forma. Pueden, sin embargo, escupir pintura en el suelo. ¡Y pueden escupir en tres colores diferentes! Hay rumores de que una vez hubo una tribu de Greeps que usó esto para transmitirse información entre ellos.
- Los Greeps son muy cortos de vista. Sólo pueden ver el suelo de su posición inmediata, y no pueden ver más allá.
- Los Greeps tienen buena memoria —nunca olvidan lo que saben—. Sin embargo —desafortunadamente— su memoria es muy limitada. Sólo pueden recordar unas pocas cosas a la vez.

Armado con este extenso conocimiento, estamos listos para programar nuestros Greeps.

I2.2 Programando tus Greeps

Para programar tus Greeps de forma que recojan tantos tomates como sea posible, deberías mejorar su conducta. La clase Greep, que se ha introducido en este escenario, ya incluye alguna conducta (aunque no muy inteligente) que puedes mirar para comenzar.

Podemos ver que `Greep` es una subclase de `Creature`. La clase `Creature` proporciona un buen número de métodos útiles que podemos usar.

Hay, sin embargo, un conjunto de reglas que debemos seguir:

Regla 1: *Cambia sólo la clase* «Greep». *No puedes crear ni modificar ninguna otra clase.*

Regla 2: *No puedes ampliar la memoria de los Greeps. Esto es, no te está permitido añadir campos a la clase (salvo campos constantes, final). Se proporciona una memoria de propósito general (un entero y dos booleanos).*

Regla 3: *No puedes moverte más de una vez por ronda de actuación (método act).*

Regla 4: *Los Greeps no se comunican directamente. No llaman a los métodos de los otros Greeps ni pueden acceder a los campos de los otros.*

Regla 5: *No puede haber visión amplia. Se te permite ver el mundo pero sólo la localización inmediata del Greep. Los Greeps son casi ciegos, y no pueden mirar más allá.*

Regla 6: *No se pueden crear objetos. No se te permite crear ningún objeto del escenario (instancias de clases definidas por el usuario, tales como* `Greep` *o* `Paint`*). Los Greeps no tienen poderes mágicos —no pueden crear cosas a partir de nada.*

Regla 7: *Sin teletransporte. Los métodos de* `Actor` *que cambian los movimientos normales (tales como setLocation) no se pueden usar como trampa.*

Es importante seguir estas reglas. Es fácil saltárselas técnicamente, pero se considera trampa.

Para programar tus Greeps, trabaja principalmente en el método `act` de los Greeps (y cualesquiera otros métodos privados que decidas crear).

Algunas pistas para comenzar:

- Lee la documentación de la clase `Creature`. (La mejor forma de hacer esto es abrir la clase en el editor y cambiar a la vista Documentación.) Aquí encontrarás algunos de los métodos más útiles para tu tarea. Mira bien qué hay aquí.
- Trabaja en pasos pequeños. Comienza haciendo pequeñas mejoras y mira qué pasa.
- Algunas de las primeras mejoras podrían ser: girar cuando estás en el agua; esperar si encuentras una pila de tomates (e intenta cargar tomates); girar si estás en el borde del mundo; . . .

Te darás cuenta pronto de muchas otras mejoras que puedes hacer. Es especialmente interesante cuando comienzas a usar las gotas de pintura del suelo para hacer marcas que vean otros Greeps.

I2.3 Ejecutando la competición

Ayuda tener un juez que ejecute la competición. En una escuela, el juez podría ser el profesor. Si lo ejecutas con tus amigos, se puede escoger a una persona (que no tome parte como concursante de la competición él mismo).

Para hacer la competición interesante, debería haber dos versiones del escenario Greeps. Una se entrega a todos los concursantes. (Esta es la que se incluye en los escenarios del libro.) Este escenario incluye tres mapas diferentes. Los Greeps aterrizan y buscan en cada uno de estos tres mapas, uno tras otro. (Por tanto, el reto para los concursantes es desarrollar algoritmos de movimiento que sean suficientemente flexibles para maniobrar en diferentes mapas, no sólo en uno conocido.)

El juez debería tener un escenario diferente que incluya más mapas. Recomendamos ejecutar la competición con 10 mapas diferentes. Los concursantes no tienen acceso a los últimos siete mapas —sólo pueden probar con los tres primeros. Entonces prueban sus Greeps con estos mapas para ver su puntuación, y el juez ejecuta entonces los Greeps con los 10 mapas (puede que en una pantalla muy grande) para obtener la puntuación oficial.

Lo mejor es que la competición dure varios días (incluso una semana o dos), ofreciendo varias oportunidades a los concursantes de enviar su trabajo para obtener la puntuación, de forma que puedan ver cómo mejoran poco a poco.

I2.4 Detalles técnicos

Para enviar una entrada al juez, la forma más fácil de hacerlo es que los concursantes envíen sólo el fichero *Greeps.java*. El juez entonces copia este fichero en su escenario completo (de 10 mapas), lo recompila y lo ejecuta. Esto asegura que no se ha modificado ninguna otra clase en el proceso.

Algunos elementos gráficos (para hacer folletos para la competición) están disponibles en

```
http://www.greenfoot.org/competition/greeps/
```

Los instructores pueden también encontrar instrucciones para obtener una versión del escenario Greeps con 10 mapas. Otra opción es que los instructores hagan más mapas ellos mismos con bastante facilidad. Una imagen de un mapa vacío se encuentra en la carpeta *images* del escenario Greeps. El agua se puede pintar simplemente sobre el mapa, y los datos del mapa (localización de pilas de tomates, etc.) se pueden especificar en la clase `Earth`.

CAPÍTULO

8

Creando imágenes y sonido

temas:	creando sonidos, creando imágenes, cambios de imágenes dinámicas, gestionando la entrada del ratón
conceptos:	formatos de sonido, parámetros de calidad del sonido, formatos de ficheros de imágenes, modelo de colores RGBA, transparencia

Muchos de los escenarios que nos hemos encontrado previamente fueron interesantes no sólo por el código de programa que definía su conducta, sino también porque hicieron un buen uso de sonido e imágenes. Hasta ahora, no hemos hablado mucho sobre la producción de ficheros multimedia, y la mayoría se basan en ficheros existentes de imágenes y sonido.

En este capítulo, trataremos algunos aspectos sobre la creación y manipulación de ficheros multimedia. Primero revisaremos algunos conocimientos básicos sobre sonido en programas de ordenador, y continuaremos con varias técnicas para crear y gestionar imágenes

Como efecto colateral, trataremos la gestión de la entrada del ratón.

8.1 Preparación

Al contrario que en capítulos anteriores, no construiremos un escenario completo en este capítulo sino que trabajaremos en ejercicios más pequeños que ilustren las diferentes técnicas que se pueden incorporar en una gran variedad de escenarios. La primera secuencia de ejercicios nos guía a través de la creación de un escenario que reproduce un sonido —que creamos nosotros mismos— cuando el usuario pincha en un actor.

Para estos ejercicios, no usaremos un escenario inicial, preparado, y parcialmente implementado, sino que crearemos uno nuevo desde cero.

Ejercicio 8.1 Como preparación para los ejercicios de este capítulo, crea un escenario. Puedes llamarlo como te guste.

Verás que el nuevo escenario incluye automáticamente las superclases `World` y `Actor`, pero ninguna más.

Ejercicio 8.2 Crea una subclase de `World`. Llámala `MyWorld`. Puedes ponerle cualquier imagen de fondo que quieras. Compila.

Ejercicio 8.3 Cambia el tamaño y la resolución de tu mundo de forma que tenga un tamaño de celda de un píxel y un tamaño de 400 celdas de ancho y 300 celdas de alto.

Ejercicio 8.4 Crea una subclase de actor en tu escenario. En este punto, no importa mucho lo que sea. Puedes querer mirar en la librería de imágenes mostradas en el diálogo *Nueva clase*, y escoge la que parezca interesante. Nombra la clase apropiadamente. (Recuerda: los nombres de las clases deben comenzar con una letra mayúscula.)

Ejercicio 8.5 Añade código a tu clase `MyWorld` para que coloque una instancia de tu actor automáticamente en el mundo.

Ejercicio 8.6 Escribe código en el método `act` de tu actor para que mueva al actor 10 píxeles a la derecha cada vez que actúe.

Deberías tener ahora un escenario con un actor que se mueve a la derecha cuando lo ejecutas. El movimiento no es, sin embargo, nuestro principal objetivo ahora. Hemos añadido el movimiento sólo para tener un efecto visual inicial con el que experimentar.

El siguiente paso de nuestra preparación será hacer que el actor reaccione al ratón.

Ejercicio 8.7 En la clase `Greenfoot`, hay varios métodos para gestionar la entrada del ratón. ¿Cuáles son? Busca en la documentación de las clases de Greenfoot y anótalos.

Ejercicio 8.8 ¿Cuál es la diferencia entre `mouseClicked` y `mousePressed`?

Cuando queremos reaccionar a los clicks del ratón, podemos usar el método `mouseClicked` (ratón-Pulsado) de la `claseGreenfoot`. Este método devuelve un booleano que se puede usar como condición de una sentencia `if`.

El parámetro del método `mouseClicked` puede especificar un objeto en el que se pulsó el ratón. Podemos pasar `null` como parámetro si no nos preocupa dónde se pinchó el ratón —el método entonces devolverá true si el ratón se pinchó en cualquier sitio.

Ejercicio 8.9 Modifica el código de tu clase actor para que se mueva a la derecha como reacción a un click del ratón. El ratón puede ser pulsado en cualquier sitio del mundo.

Ejercicio 8.10 Ahora modifica tu código para que el actor sólo se mueva cuando el usuario pincha en el actor. Para hacer esto, tienes que pasar el actor mismo (en vez de `null`) como parámetro al método `mouseClicked`. Recuerda que puedes usar la palabra clave `this` para referirte al objeto actual.

Ejercicio 8.11 Prueba tu código. Coloca instancias de tu actor en el mundo y asegúrate de que sólo se mueve en el que pinchas.

Deberías tener ahora un escenario con un actor que reacciona a los clicks del ratón. Es un buen punto de partida para nuestros experimentos con sonido e imágenes. (Si tuviste problemas en crear esto, hay un escenario llamado *soundtest* en los escenarios del libro de este capítulo que implementa este punto de partida.)

8.2 Trabajando con sonido

Como hemos visto ya antes, la clase `Greenfoot` tiene un método `playSound` que podemos usar para reproducir un fichero de sonido. Para que se pueda reproducir, el fichero de sonido debe estar dentro de la carpeta *sounds* de la carpeta del escenario.

Como comienzo, reproduzcamos un fichero de sonido existente.

Ejercicio 8.12 Selecciona un fichero de sonido de los de tus otros escenarios de Greenfoot y cópialo en la carpeta *sounds* de tu escenario actual. Entonces modifica tu actor para que reproduzca el sonido (en vez de moverse) cuando pinchas en él.

Pifia

Algunos sistemas operativos están configurados de forma que las extensiones de los ficheros no se muestran. En este caso, un fichero cuyo nombre completo sea *mysound.wav* se mostraría sólo como *mysound*. Esto es un problema, porque necesitamos usar el nombre completo, incluyendo la extensión, en nuestro código Java. Si escribes

```
Greenfoot.playSound("mysound");
```

fallaría, porque no encuentra el fichero. Sin embargo, sin saber la extensión, no sabes qué es.

La solución es cambiar la configuración del sistema operativo de forma que se muestren siempre las extensiones. Windows, por ejemplo, tiene una casilla denominada *Oculta extensiones para tipos de ficheros conocidos*, y deberías asegurarte de que no está seleccionada. En Windows Vista, puedes encontrar esta casilla buscando en los contenidos de tu carpeta y yendo a los menús*Organiza*/*Carpeta y Opciones de búsqueda*/*Ver*. En otros sistemas de Windows, los nombres de los menús pueden variar, pero también habrá una casilla así.

Podemos reproducir fácilmente un fichero de sonido existente. La tarea más interesante ahora es hacer nosotros mismos estos ficheros de sonido.

8.3 Grabación y edición de sonido

Hay varias opciones para obtener ficheros de sonido. Podemos copiar sonidos de otros proyectos de Greenfoot, o descargarlos de varias librerías de sonidos gratuitos de Internet. Si copias sonidos de Internet, fíjate bien en temas de licencias: no todo lo que está en Internet es gratis —¡respeta los derechos de autor de otras personas!— La opción más fácil para obtener ficheros de sonido es grabarlos nosotros mismos.

Para hacer esto, necesitamos un micrófono (muchos portátiles tienen un micrófono integrado, y a menudo los auriculares tienen micrófonos anexos) y software de grabación de sonidos. Si no tienes un micrófono a mano ahora mismo, puede que quieras saltarte esta sección.

Hay muchos programas de grabación de sonido disponibles, algunos de los cuales son gratis. Aquí usaremos *Audacity*[1] para los pantallazos de muestra. Audacity es una buena opción porque es potente, se ejecuta en diferentes sistemas operativos, es gratuito y es bastante fácil de usar. Como hay muchos otros programas de grabación de sonido, escoge el que tú prefieras.

La Figura 8.1 muestra una interfaz típica de un programa de grabación de sonido. Típicamente tienes controles para grabar, reproducir, etc., y un visor de la forma de onda del sonido grabado (el grafo azul).

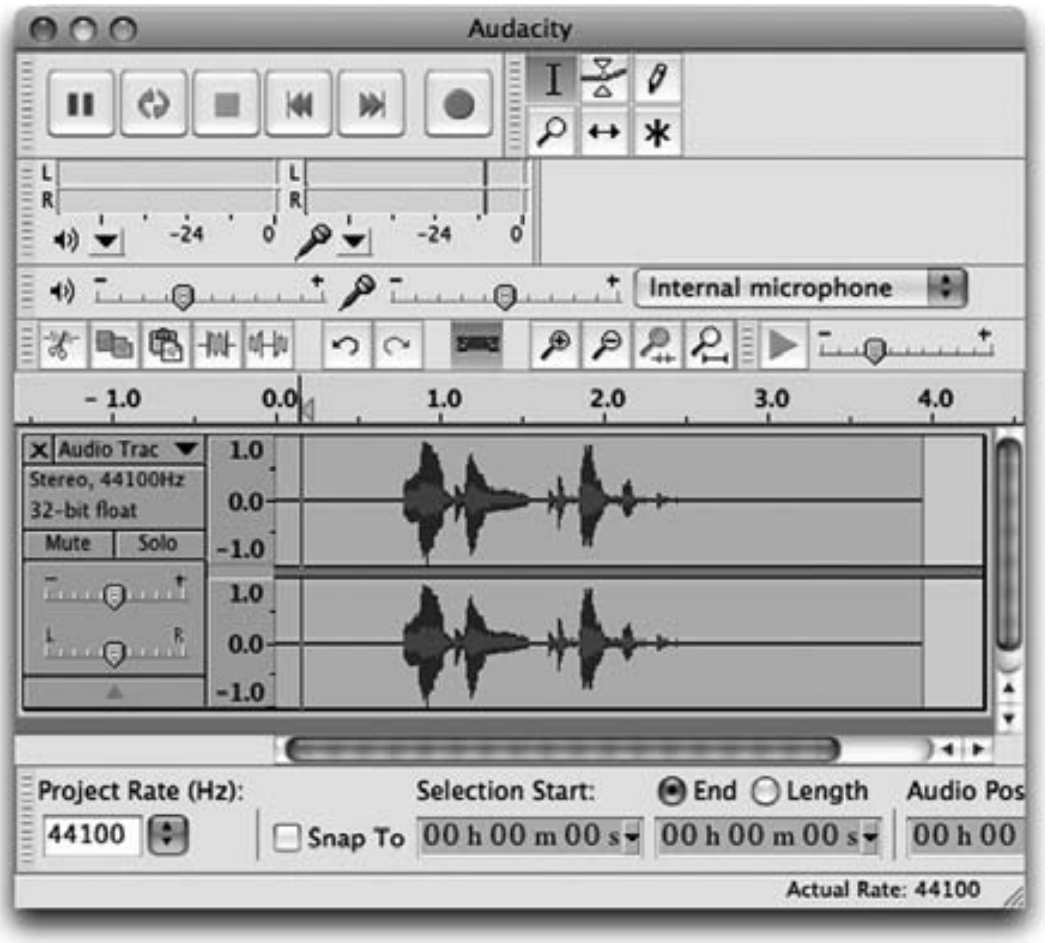


Figura 8.1
Un programa de grabación y edición de sonido (Audacity)

Grabar el sonido es muy sencillo —puedes normalmente averiguar cómo jugando con el programa un rato.

Ejercicio 8.13 Lanza un programa de grabación de sonido y graba un sonido. Reprodúcelo. ¿Ha salido como esperabas? Si no, bórralo e inténtalo de nuevo.

[1] Audacity está disponible en http://audacity.sourceforge.net

Cuando grabamos sonidos, a menudo tenemos unos pocos segundos al principio o al final que no queremos. La mayoría de los programas de grabación de sonido nos permiten editar el sonido después de guardarlo, de forma que podemos cortar lo que no queremos.

En la Figura 8.1, por ejemplo, podemos ver un silencio demasiado largo al final del fichero de sonido (las líneas rectas horizontales a la izquierda y a la derecha del final del grafo). Si guardamos el sonido tal cual, el efecto será que el sonido parece que se retrasa cuando lo reproducimos (ya que la primera parte del sonido se reproduce con un segundo de silencio).

Podemos editar el sonido seleccionando partes de él, y usar cortar, copiar y borrar para copiar o borrar las partes seleccionadas.

Ejercicio 8.14 Edita tu fichero de sonido de forma que incluya sólo el sonido exacto que quieres. Borra cualquier ruido o silencio del principio o el final, o cualquier parte entre medias que no necesites.

Muchos programas de edición de sonido también ofrecen filtros para modificar el sonido o generar nuevos elementos de sonido por completo. Usando filtros, se pueden generar muchos efectos de sonido diferentes. Al aplicar efectos, tales como amplificación, ecos, inversión, cambios de velocidad, y otros, a sonidos simples (tales como palmadas, silbidos, gritos, etc.), podemos crear una gran variedad de efectos.

Ejercicio 8.15 Si tu programa de sonido soporta filtros, aplica algunos de estos filtros al sonido que has grabado. Selecciona tres de tus filtros favoritos y describe, por escrito, lo que hacen. Describe un ejemplo donde podrías usar este efecto.

Ejercicio 8.16 Produce los siguientes sonidos: un ratón mordiendo una zanahoria; una explosión, un sonido de dos objetos duros chocando; un sonido «game over» cuando un jugador pierde; un «fin de juego» cuando el jugador ha ganado; una voz de robot; y un sonido de «saltar» (usado en juegos de saltos).

Cuando se termina la edición del sonido, estamos listos para guardarlo en un fichero.

8.4 Formatos de ficheros de sonido y tamaños de ficheros

Los ficheros de sonido se pueden guardar en muchos formatos diferentes y con diferentes codificaciones, y esto puede ser muy confuso muy rápidamente.

Greenfoot puede reproducir sonidos guardados en formatos WAV, AIFF y AU. (No ficheros MP3, me temo.) Estos formatos, sin embargo, son los que se conocen como «formatos de envoltorio» —pueden contener diferentes codificaciones, y Greenfoot puede leer sólo algunas de ellas—. Como resultado, Greenfoot no puede, por ejemplo, reproducir todos los ficheros WAV.

Cuando guardas tus propios ficheros grabados, deberías guardarlos como un fichero «signed 16 bit PCM WAV». Éste es el formato más seguro para asegurar su reproducción. En muchos programas de

Concepto:

Los sonidos se pueden guardar en una gran variedad de **formatos** y **codificaciones** diferentes. No todos los programas pueden reproducir todos los formatos de sonido. En el caso de Greenfot, normalmente usamos el formato **WAV**.

grabación de sonido, se puede hacer usando una función «Exportar», en vez de la función estándar «Guardar». Asegúrate de guardar en este formato.

Cuando encuentras ficheros de sonido que Greenfoot no puede reproducir (puede que sean descargados de Internet), normalmente puedes abrirlos en tu programa de edición de sonido y convertirlos a este formato.

Ejercicio 8.17 Guarda tu sonido grabado en un formato apropiado para Greenfoot. Mueve el sonido a la carpeta *sounds* de tu escenario. Modifica el código de tu clase actor para que reproduzca tu sonido cuando se pinche.

Ejercicio 8.18 Modifica tu código de forma que reproduzca un efecto de sonido cuando pinches con el botón izquierdo del ratón, y otro efecto de sonido cuando pinches con el botón derecho del ratón. Para hacer esto,necesitas obtener información sobre qué botón del ratón se ha pulsado. Greenfoot tiene métodos para esto —estudia la documentación de la clase Greenfoot para averiguar cómo hacer esto.

Ejercicio 8.19 Modifica tu código para que el actor reproduzca un efecto de sonido cuando pinchas en él, y se mueva además a una posición aleatoria.

Concepto:

La configuración del **formato de muestreo, tasa de muestreo**, y **estéreo/mono** de una grabación de sonido determina el tamaño del fichero y la calidad del sonido.

Los sonidos de fichero pueden ocupar mucho muy rápidamente. Esto no es un problema importante cuando usas el escenario localmente, pero si el escenario se exporta, por ejemplo a la Galería de Greenfoot, entonces el tamaño puede ser importante. Los ficheros de sonido e imagen son normalmente las partes más grandes de un escenario de Greenfoot, y los tamaños de los ficheros de sonido pueden determinar la diferencia entre un escenario que se carga rápidamente y otro en que los usuarios tienen que esperar minutos para descargar el escenario.

Para evitar ficheros de sonido muy grandes, deberíamos prestar atención a los destalles de codificación. Cuando grabamos y guardamos sonidos, podemos encontrar un compromiso entre calidad de sonido y tamaño de fichero. Podemos grabar y guardar el sonido o bien con calidad muy alta, que lleva a ficheros muy grandes, o con calidad más baja, que lleva a ficheros más pequeños. Las configuraciones que podemos variar principalmente son:

- El formato de muestreo (normalmente 16-bit, 24-bit o 32-bit).
- La tasa de muestreo, medida en hertzios (Hz), que normalmente varía entre 8.000 Hz y 96.000 Hz.
- Grabación estéreo o mono. (La grabación estéreo graba dos pistas separadas, produce por tanto el doble de datos.)

Si miras con detalle a la Figura 8.1, puedes ver que el sonido del pantallazo ha sido grabado en 32 bit, 44.100 Hz, estéreo.

Ésta es una configuración típica por defecto en los programas de grabación de sonido, pero en realidad tiene mucha más calidad de la que necesitamos para efectos de sonido simples. (Podríamos querer esta calidad para oír música que nos guste, pero la necesitamos para un efecto de sonido de un disparo corto, *¡Bang!*)

En general, deberías considerar guardar tus sonidos con calidad más baja, a menos que sientas que realmente necesitas más calidad.

Ejercicio 8.20 Busca en tu programa de grabación de sonido cómo se configura el formato de muestreo, la tasa de muestreo, y la grabación estéreo/mono. En algunos programas, puedes convertir sonidos existentes. En otros programas, puedes especificar estas configuraciones sólo para las grabaciones nuevas. Crea un sonido grabando con diferentes formatos de muestreo, con diferentes tasas de muestreo, y en estéreo y mono. Guarda todo en ficheros diferentes y compara los tamaños de los ficheros. ¿Qué cambio ha hecho que el fichero sea más pequeño?

Ejercicio 8.21 Escucha los sonidos creados en el ejercicio previo. ¿Puedes notar la diferencia? ¿Cuánto puedes reducir la calidad (y el tamaño del fichero) manteniendo una calidad aceptable?

8.5 Trabajando con imágenes

Como hemos comentado previamente en capítulos previos (p. ej., cuando creamos el fondo de asteroides en el Capítulo 7), podemos tratar con imágenes para actores de dos formas: podemos usar imágenes de ficheros, o podemos pintar una imagen al vuelo en nuestro programa.

Veremos ambos métodos con más detalle aquí.

8.6 Ficheros de imagen y formatos de ficheros

Hay varias formas de conseguir imágenes para nuestros escenarios. La más fácil es, por supuesto, usar las que vienen en la Librería de Imágenes de Greenfoot. Estas imágenes se presentan automáticamente cuando creas clases nuevas. Hay también buenas bibliotecas de imágenes e iconos gratuitos en Internet. Deberías encontrarlas tras buscarlas unos pocos minutos. (Asegúrate, no obstante, de que las imágenes que quieres usar están disponibles realmente para uso público gratuito —no todo es gratis o de dominio público simplemente porque esté en Internet. Respeta los derechos de autor de los demás y los términos de las licencias.)

La alternativa más interesante, sin embargo, si queremos hacer nuestros escenarios únicos y otorgarles su propia atmósfera, es hacer las imágenes nosotros mismos.

Hay varios programas de gráficos disponibles que podemos usar para crear imágenes. *Photoshop* es probablemente el programa comercial más conocido, y es realmente un programa muy bueno si lo tienes. Sin embargo, también hay programas de código abierto y gratuitos que proporcionan una funcionalidad similar. *Gimp*[2] es un programa gratuito excelente con muchas funcionalidades sofisticadas, que merece que lo instales. Hay también muchos programas de dibujo más sencillos que podrías usar.

Producir gráficos bonitos lleva algún tiempo para aprenderlo, y no podemos tratarlo con detalle en este libro. Juega y practica, y encontrarás muchas técnicas y trucos. Aquí, nos centraremos en los detalles técnicos de usar imágenes.

Una de las preguntas más importantes es qué formato de fichero debemos usar para guardar imágenes. Como vimos con los sonidos, hay un compromiso entre la calidad y el tamaño del fichero. Los ficheros de imagen pueden ser muy grandes (mucho más grandes que los ficheros de código de nuestros escenarios), por lo que fácilmente pueden determinar el tamaño de descarga de nuestro proyecto. De nuevo, esto es particularmente importante si vamos a exportar nuestro escenario a un servidor web,

[2] http://www.gimp.org.

como la Galería de Greenfoot. Formatos de imagen diferentes pueden conllevar tamaños de ficheros 10 veces más pequeños o más, lo que quiere decir que el escenario se descargará 10 veces más rápido (porque es sólo un décimo del tamaño) si escogemos los formatos bien.

Concepto:

El formato de imágenes **JPEG** comprime las imágenes muy bien. Es a menudo la mejor opción para fondos.

Greenfoot puede leer imágenes en formatos JPEG, PNG, GIF, BMP y TIFF. De estos, JPEG y PNG son los mejores formatos para la mayoría de los usos.

Las imágenes JPEG tienen la ventaja de que comprimen muy bien. Esto significa que se pueden guardar en ficheros muy pequeños. Esto es particularmente cierto para imágenes en color, tales como fotos y fondos (es por lo que muchas cámaras digitales usan este formato). Cuando guardas imágenes JPEG, muchos programas de gráficos te permiten escoger cuánto quieres comprimir el fichero. Cuanto más comprimamos, más pequeño será el fichero, pero también la calidad se reducirá. Gimp, por ejemplo, presenta una barra deslizante «Calidad» cuando guardamos una imagen en formato JPEG. Al reducir la calidad, crea ficheros más pequeños.

Ejercicio 8.22 Crea una imagen con tu programa de gráficos y guárdala en un fichero JPEG: guárdala con al menos cuatro configuraciones de calidad diferentes. A continuación abre estos ficheros y compáralos visualmente. Compara también sus tamaños. ¿Qué configuración de calidad ofrece un mejor compromiso entre calidad de imagen y tamaño de fichero para tu imagen?

Ejercicio 8.23 En tu programa de gráficos, crea un nuevo fondo para tu escenario que creaste en secciones anteriores de este capítulo. Guárdalo como un fichero JPEG. Úsalo en tu escenario. ¿Qué tamaño (ancho y alto) debería tener la imagen? ¿Qué ocurre si es muy grande? ¿Qué pasa si es muy pequeño?

Concepto:

Los píxeles de las imágenes tienen un valor de **transparencia** que determina si podemos ver a través de ellos. Los píxeles pueden ser parcialmente transparentes. Si son totalmente transparentes, son invisibles.

Ejercicio 8.24 ¿Cómo crees que el algoritmo JPEG hace que los ficheros sean más pequeños? ¿Cómo podría funcionar esto? Intenta pensar algunas teorías o conjeturas de cómo se podría hacer esto.

Ejercicio 8.25 ¿Cómo comprime JPEG en realidad los ficheros? Investiga en Internet para averiguarlo y escribe la respuesta.

El segundo formato de imagen que nos es muy útil es PNG.

Las imágenes PNG tienen la ventaja de que pueden manejar la transparencia muy bien. Cualquier píxel individual puede ser en parte o totalmente transparente. Esto nos permite crear imágenes no rectangulares. (Como vimos en el Capítulo 7, todas las imágenes son rectangulares, pero al tener partes transparentes podemos crear formas arbitrarias.)

Concepto:

El formato de imagen **PNG** es a menudo la mejor opción para imágenes de actores, ya que puede gestionar la transparencia y comprime muy bien.

Esta habilidad para gestionar la transparencia, combinada con una buena comprensión, hace que PNG sea un formato ideal para imágenes de actores. (Las imágenes JPEG no pueden tener píxeles transparentes, por lo que no las usaremos aquí, a menos que el actor sea rectangular. Para fondos, no es normalmente un problema, porque no tenemos normalmente transparencias en fondos.)

Es raro necesitar usar los formatos de imagen BMP, TIFF, o GIF. BMP no comprime tan bien como otros formatos, y no soporta píxeles transparentes. Las imágenes TIFF pueden conservar muy bien la calidad, pero crean ficheros más grandes. GIF es un formato propietario que ha sido reemplazado por uno mejor, PNG; que es además gratis.

Ejercicio 8.26 Crea dos nuevas imágenes para tu actor (el actor alternará entre estas dos imágenes). Guárdalas en formato PNG. Haz que una de estas imágenes sea la imagen por defecto de tu clase actor.

Ejercicio 8.27 Modifica el código de tu actor de forma que conmute entre las dos imágenes cada vez que pinchas dos veces en el actor.

Ejercicio 8.28 Modifica de nuevo el código de tu actor para que muestre la segunda imagen del actor cuando pulsas el ratón. En otras palabras, el actor comienza con la imagen por defecto; cuando pulsas el ratón sobre el actor, muestra una imagen diferente, y tan pronto como liberas el botón, vuelve a la imagen original.

8.7 Dibujando imágenes

El segundo método para conseguir imágenes para nuestros actores y fondos es dibujarlas programáticamente. Hemos visto ejemplos de esto en algunos escenarios en capítulos previos, por ejemplo, cuando pintamos las estrellas en el programa de asteroides.

Cada píxel en una imagen se define por dos valores: su color y su transparencia (también llamada *valor alfa*).

El valor del color se divide a su vez en tres componentes: rojo, verde y azul [3]. Los colores representados de esta forma se llaman normalmente como colores RGB [4].

Esto significa que podemos representar un píxel (con color y transparencia) con cuatro números. El orden normalmente es como sigue:

```
( R, G, B, A )
```

Esto es, los primeros tres valores definen las componentes roja, verde y azul (en este orden), y la última es el valor alfa.

En Java, todos estos valores están en el rango [0..255] (de 0 a 255 incluidos). Un valor de componente de color de 0 indica que no hay color en esta componente, mientras que 255 indica toda la fuerza. Un valor alfa de 0 es totalmente transparente (invisible), mientras que 255 es opaco (sólido).

La Figura 8.2 muestra una tabla de algunos de los colores posibles, todos sin transparencia (alfa = 255). La tabla de la figura 8.2 fue creada con el escenario *color-chart* de Greenfoot, que está en la carpeta *chapter08* de los escenarios del libro.

Ejercicio 8.29 ¿Qué píxeles parece que tienen un valor color/alfa de (255, 0, 0, 255)? ¿Cuál es (0, 0, 255, 128)? Y ¿cuál es (255, 0, 255, 230)?

[3] Esto es simplemente un modelo posible para representar el color. Hay otros en uso para gráficos de ordenador e imprenta. Sin embargo, éste es el más común en programación en Java, por lo que nos concentraremos en este modelo aquí.

[4] N. del T. RGB viene de los colores en inglés Red (rojo), Green (verde) y Blue (azul).

Figura 8.2
Tabla de color RGB

0,0,0	0,0,51	0,0,102	0,0,153	0,0,204	0,0,255
0,51,0	0,51,51	0,51,102	0,51,153	0,51,204	0,51,255
0,102,0	0,102,51	0,102,102	0,102,153	0,102,204	0,102,255
0,153,0	0,153,51	0,153,102	0,153,153	0,153,204	0,153,255
0,204,0	0,204,51	0,204,102	0,204,153	0,204,204	0,204,255
0,255,0	0,255,51	0,255,102	0,255,153	0,255,204	0,255,255
51,0,0	51,0,51	51,0,102	51,0,153	51,0,204	51,0,255
51,51,0	51,51,51	51,51,102	51,51,153	51,51,204	51,51,255
51,102,0	51,102,51	51,102,102	51,102,153	51,102,204	51,102,255
51,153,0	51,153,51	51,153,102	51,153,153	51,153,204	51,153,255
51,204,0	51,204,51	51,204,102	51,204,153	51,204,204	51,204,255
51,255,0	51,255,51	51,255,102	51,255,153	51,255,204	51,255,255
102,0,0	102,0,51	102,0,102	102,0,153	102,0,204	102,0,255
102,51,0	102,51,51	102,51,102	102,51,153	102,51,204	102,51,255
102,102,0	102,102,51	102,102,102	102,102,153	102,102,204	102,102,255
102,153,0	102,153,51	102,153,102	102,153,153	102,153,204	102,153,255
102,204,0	102,204,51	102,204,102	102,204,153	102,204,204	102,204,255
102,255,0	102,255,51	102,255,102	102,255,153	102,255,204	102,255,255
153,0,0	153,0,51	153,0,102	153,0,153	153,0,204	153,0,255
153,51,0	153,51,51	153,51,102	153,51,153	153,51,204	153,51,255
153,102,0	153,102,51	153,102,102	153,102,153	153,102,204	153,102,255
153,153,0	153,153,51	153,153,102	153,153,153	153,153,204	153,153,255
153,204,0	153,204,51	153,204,102	153,204,153	153,204,204	153,204,255
153,255,0	153,255,51	153,255,102	153,255,153	153,255,204	153,255,255
204,0,0	204,0,51	204,0,102	204,0,153	204,0,204	204,0,255
204,51,0	204,51,51	204,51,102	204,51,153	204,51,204	204,51,255
204,102,0	204,102,51	204,102,102	204,102,153	204,102,204	204,102,255
204,153,0	204,153,51	204,153,102	204,153,153	204,153,204	204,153,255
204,204,0	204,204,51	204,204,102	204,204,153	204,204,204	204,204,255
204,255,0	204,255,51	204,255,102	204,255,153	204,255,204	204,255,255
255,0,0	255,0,51	255,0,102	255,0,153	255,0,204	255,0,255
255,51,0	255,51,51	255,51,102	255,51,153	255,51,204	255,51,255
255,102,0	255,102,51	255,102,102	255,102,153	255,102,204	255,102,255
255,153,0	255,153,51	255,153,102	255,153,153	255,153,204	255,153,255
255,204,0	255,204,51	255,204,102	255,204,153	255,204,204	255,204,255
255,255,0	255,255,51	255,255,102	255,255,153	255,255,204	255,255,255

En Greenfoot, los colores se representan por objetos de la clase `Color` del paquete `java.awt`. Después de importar esta clase, podemos crear objetos simplemente con los valores RGB:

```
Color mycol = new  Color (255, 12, 34);
```

o con valores RGB y un valor alfa:

```
Color mycol = new  Color (255, 12, 34, 128);
```

Si no especificamos un valor alfa, el color es totalmente opaco.

Ejercicio 8.30 Crea un nuevo escenario de Greenfoot denominado *color-test*. En él, crea un mundo con un fondo que muestre un patrón. Crea una subclase de `Actor` denominada `ColorPatch`. Programa la clase `ColorPatch` para que genere una nueva imagen `GreenfootImage` de tamaño fijo, rellena con un color fijo. Usa esta imagen como imagen del actor. Experimenta con diferentes colores y valores alfa.

Ejercicio 8.31 Modifica tu código para que la imagen de color del ejercicio anterior, `ColorPatch`, cuando se cree, tenga un tamaño aleatorio, esté rellena con un color aleatorio, y tenga una transparencia aleatoria.

Ejercicio 8.32 Modifica tu código de nuevo para que la imagen no esté rellena, sino que tenga 100 pequeños puntos coloreados dentro de ella, en posiciones aleatorias dentro de la imagen del actor.

8.8 Combinando ficheros de imágenes y dibujo dinámico

Algunos de los efectos visuales más interesantes se consiguen cuando combinamos imágenes estáticas de ficheros con cambios dinámicos hechos por nuestro programa. Podemos, por ejemplo, comenzar con un fichero de imagen estática, y entonces pintar en ella con colores diferentes, o escalarla, o decolorarla cambiando su transparencia.

Para probar esto, crearemos un efecto de humo. En nuestro próximo escenario, haremos que una pelota se mueva por la pantalla, dejando un rastro de humo detrás (mira la Figura 8.3).

Ahora trabajaremos en el efecto de humo. Primero, crea una imagen que muestre una bocanada de humo (Figura 8.4).

Ejercicio 8.33 Crea un nuevo escenario denominado *smoke* (humo). Haz una imagen de fondo neutral, que sea bonita para el escenario.

Ejercicio 8.34 Crea una bola que se mueva por la pantalla a velocidad constante. Cuando choca con el borde de la pantalla, rebota. (La pantalla es, en efecto, una caja, y la pelota rebota en ella.)

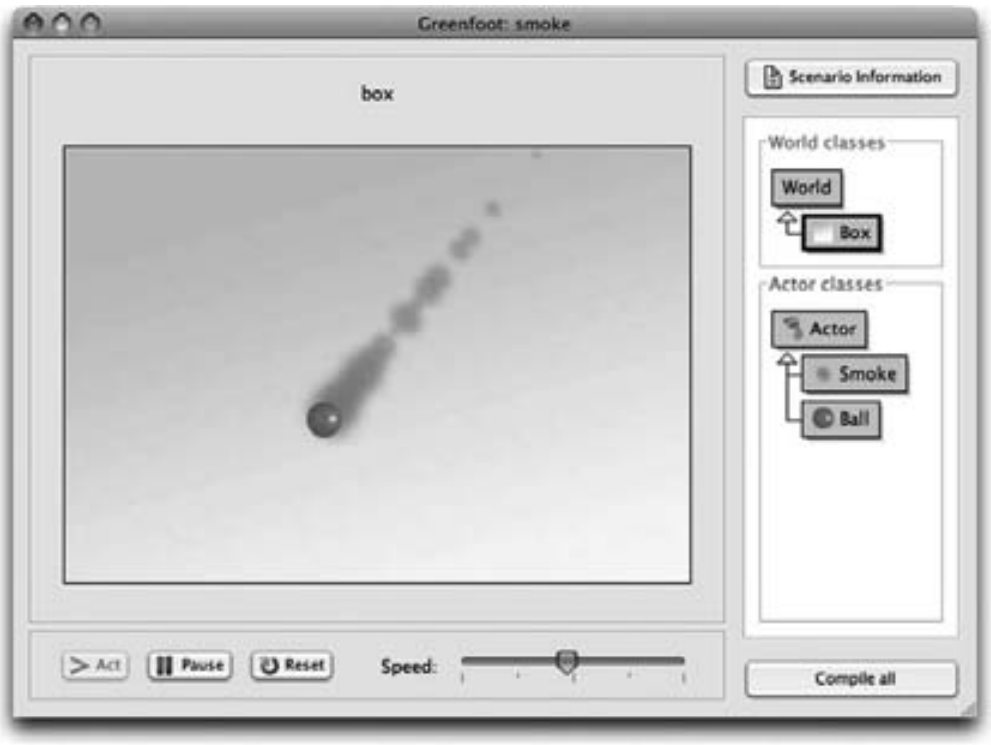


Figura 8.3
El efecto de rastro de humo

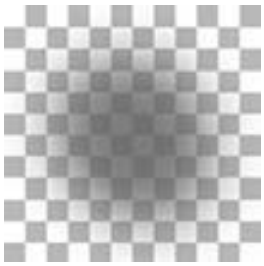


Figura 8.4
Una imagen de bocanada de humo

Observa que el fondo a cuadros no es parte de la imagen —se muestra aquí sólo para demostrar que la imagen de bocanada de humo es semi-transparente (podemos ver el fondo detrás de ella).

Tu humo no tiene que ser verde —puedes hacerlo en cualquier color que te guste— pero debería ser transparente. Puedes conseguir esto con un programa de gráficos bueno simplemente dibujando un punto con un pincel de pintura grande, suave y semi-transparente. Si tienes problemas al hacer esta imagen, puedes usar la imagen ya preparada *smoke.png* de la carpeta *chapter08* de los escenarios del libro.

Ejercicio 8.35 Crea la imagen de humo descrita antes.

Ejercicio 8.36 Crea una clase actor `Smoke` en tu escenario. Cuando la insertes en tu mundo, un actor `Smoke` debería desvanecerse rápidamente. Esto es, en cada ciclo de *act*, la imagen del actor debería ser más pequeña y más transparente. (La clase `GreenfootImage` tiene métodos para ajustar el tamaño y la transparencia.). Cuando son totalmente transparentes o muy pequeñas, deberían eliminarse del mundo.

Ejercicio 8.37 Modifica tu pelota para que deje bocanadas de humo detrás. Producir una bocanada de humo cada vez que la pelota se mueve puede ser excesivo. Intenta crear una nueva bocanada de humo sólo en cada segundo paso de la pelota. Pruébalo.

Ejercicio 8.38 Modifica tu humo para que no se desvanezca siempre con la misma velocidad. Introduce una tasa de desvanecimiento aleatoria, de forma que algunas bocanadas de humo se desvanezcan más rápidamente que otras.

Si has completado los ejercicios anteriores, entonces deberías tener un bonito rastro de humo detrás de tu pelota. Si has tenido problemas al hacer estos ejercicios, o quieres comparar tu solución con la nuestra, puedes mirar en el escenario de humo de los escenarios del libro. Implementa la tarea descrita aquí.

8.9 Resumen

Ser capaz de producir sonidos e imágenes es una habilidad de gran valor para producir juegos, simulaciones y otras aplicaciones gráficas con buen aspecto visual. Es importante conocer los formatos de fichero de sonido e imagen para escoger buenos compromisos entre tamaños de fichero y calidad.

Los sonidos se pueden grabar y editar con una gran variedad de programas de grabación de sonido, y diferentes parámetros determinan la calidad de sonido y el tamaño del fichero. Para los escenarios de Greenfoot, normalmente usamos el formato WAV con calidad bastante baja.

Las imágenes pueden también guardarse en una gran variedad de formatos. Los formatos difieren en cómo de bien comprimen los ficheros, cómo preservan la calidad de la imagen, y cómo gestionan la transparencia. JPEG y PNG son los formatos que más a menudo usamos en los escenarios de Greenfoot.

Al combinar imágenes de un fichero con operaciones de imágenes dinámicas, tales como cambios en la escala y la transparencia, podemos conseguir efectos visuales atractivos en nuestros escenarios.

Resumen de conceptos

- Los sonidos se pueden guardar en una variedad de **formatos** y **codificaciones**. No todos los programas pueden reproducir todos los formatos de sonido. Para Greenfoot, normalmente usamos el formato **WAV**.
- La configuración del **formato de muestreo, tasa de muestreo,** y configuración **estéreo/mono** de una grabación de sonido determina el tamaño del fichero y la calidad del sonido.
- El formato de imagen **JPEG** comprime las imágenes grandes muy bien. A menudo es la mejor opción para fondos.
- Los píxeles de las imágenes tienen un valor de **transparencia** que determina si podemos ver a través de ellos. Los píxeles pueden ser parcialmente transparentes. Si son totalmente transparentes, son invisibles.
- El formato de imagen **PNG** es a menudo la mejor opción para las imágenes de actores, ya que puede tratar la transparencia y comprime muy bien.

CAPÍTULO

9

Simulaciones

temas: simulaciones

conceptos: conducta emergente, experimentación

En este capítulo, veremos un tipo de software con más detalle: las simulaciones.

Las simulaciones son ejemplos fascinantes de computación, porque son altamente experimentales, y nos permiten potencialmente predecir el futuro. Muchos tipos de simulaciones pueden ser (y han sido) desarrolladas para ordenadores: simulaciones de tráfico, sistemas de previsión del tiempo, simulaciones económicas (simulaciones de la bolsa), simulaciones de reacciones químicas, explosiones nucleares, simulaciones medioambientales, y muchas más.

Concepto:

Una **simulación** es un programa de ordenador que simula algunos fenómenos del mundo real. Si las simulaciones son suficientemente precisas, podemos aprender cosas interesantes del mundo real observándolas.

Hemos visto una simulación simple en el Capítulo 6, en que simulamos parte de un sistema solar. Esa simulación fue demasiado simplista para predecir con precisión las trayectorias de planetas reales, pero hay también algunos aspectos de astrofísica que esa simulación puede ayudar a entender.

En general, las simulaciones pueden servir para dos propósitos diferentes: pueden usarse para estudiar y comprender el sistema que están simulando, o pueden usarse para predicciones.

En el primer caso, el modelado de un sistema (tales como el modelado de las estrellas y los planetas) pueden ayudarnos a entender algunos aspectos de cómo se comportan. En el segundo caso, si tenemos una simulación más precisa, podemos utilizar escenarios «qué pasaría si». Por ejemplo, podríamos tener una simulación de tráfico para una ciudad, y observamos cada mañana se desarrolla un atasco en una intersección concreta de la ciudad real. ¿Cómo podemos mejorar la situación? ¿Deberíamos construir una nueva rotonda? ¿O cambiar el sistema de luces de tráfico? ¿O puede que deberíamos construir una circunvalación?

Los efectos de cualquiera de estas intervenciones son difíciles de predecir. No podemos probar todas estas opciones en el mundo real para ver cuál es mejor, ya que sería demasiado caro y causaría mucho trastorno. Pero podemos simularlas. En nuestra simulación de tráfico, podemos probar cada opción y ver cómo mejora el tráfico.

Si la simulación es precisa, entonces el resultado que observamos en la simulación es también cierto en la vida real. Pero si esto es un gran «si»: desarrollar una simulación que sea suficientemente precisa no es fácil, y lleva un montón de trabajo. Pero para muchos sistemas, es posible hacerlo hasta un nivel útil.

Las simulaciones de previsiones del tiempo son ahora suficientemente precisas que una previsión a un día es bastante fiable. Una previsión a siete días, sin embargo, es tan buena como tirar los dados. Las simulaciones simplemente no son suficientemente buenas.

Cuando usamos simulaciones, es importante ser consciente de sus limitaciones: en primer lugar, las simulaciones siempre tienen un grado de imprecisión porque no estamos modelando los actores de forma realista por completo y porque podemos no saber con exactitud el estado exacto del sistema inicial.

Además, las simulaciones necesariamente modelan sólo parte de la realidad, y debemos ser conscientes de qué partes que dejamos fuera de nuestra simulación pueden ser relevantes en realidad.

En el ejemplo de nuestro atasco de tráfico, por ejemplo, puede que no sea la mejor solución ninguna de las opciones mencionadas antes, sino proporcionar mejor transporte público o carriles de bici para que haya menos coches en la carretera. Si nuestra simulación no incluye ese aspecto, no lo encontraríamos nunca usando la simulación, no importa lo precisa que sea.

A pesar de estas limitaciones, sin embargo, las buenas simulaciones son increíblemente útiles, e incluso puede ser fascinante jugar con simulaciones muy simples. Son tan útiles, de hecho, que casi todos los ordenadores más rápidos del mundo están ejecutando simulaciones la mayor parte del tiempo, o al menos una parte sustancial de su tiempo.

Nota al margen: Supercomputadores

Se publica regularmente una lista de los supercomputadores más rápidos del mundo en Internet en http://www.top500.org. La mayorías de ellos se describen allí con algo de detalle, y la lista incluye para muchos enlaces a sitios web donde se describe su propósito y el tipo de trabajo para el que son utilizados.

Al leer esta información, puedes ver cuántos son empleados por grandes instituciones de investigación o militares, y que casi todos se usan para ejecutar simulaciones. El uso militar es, por ejemplo, para probar explosiones nucleares en simulaciones, y las instituciones de investigación realizan diferentes experimentos científicos.

Las simulaciones también tienen un lugar especial en la historia de la programación orientada a objetos: la orientación a objetos fue inventada expresamente para ejecutar simulaciones.

El primer lenguaje de programación orientado a objetos, llamado *Simula*, se diseñó en los años sesenta por Kristen Nygaard y Ole-Johan Dahl en el Centro Noruego de Computación de Oslo. Como su nombre sugiere, su propósito era construir simulaciones. Todos los lenguajes de programación orientados a objetos de hoy en día pueden encontrar sus raíces en ese lenguaje. Dahl y Nygaard recibieron el Premio Turing de 2001 —equivalente en informática al premio Nobel— por este trabajo.

Suficiente introducción —pongámonos a teclear de nuevo y probemos algunas cosas.

En este capítulo, analizaremos una simulación de forma breve y luego trabajaremos en otra con más detalle.

9.1 Zorros y conejos

La primera simulación que investigaremos se llama *zorros-y-conejos*. Es un ejemplo típico de una clase de simulaciones llamada *simulaciones-presa-depredador* —un tipo de simulación en la que una criatura persigue (y se come) a la otra—. En nuestro caso, el depredador es un zorro y la presa es un conejo.

La idea aquí es como sigue: el escenario muestra un campo que contiene poblaciones de zorros y conejos. Los zorros se representan con cuadrados azules, y los conejos se muestran con cuadrados amarillos.

Ambas especies tienen una conducta bastante simple: los conejos se mueven aquí y allá y —si son suficientemente adultos— pueden tener descendencia. Hay una probabilidad fijada que determina en cada paso si los conejos se reproducen. Hay dos formas de morir para los conejos: mueren de viejos o son comidos por un zorro.

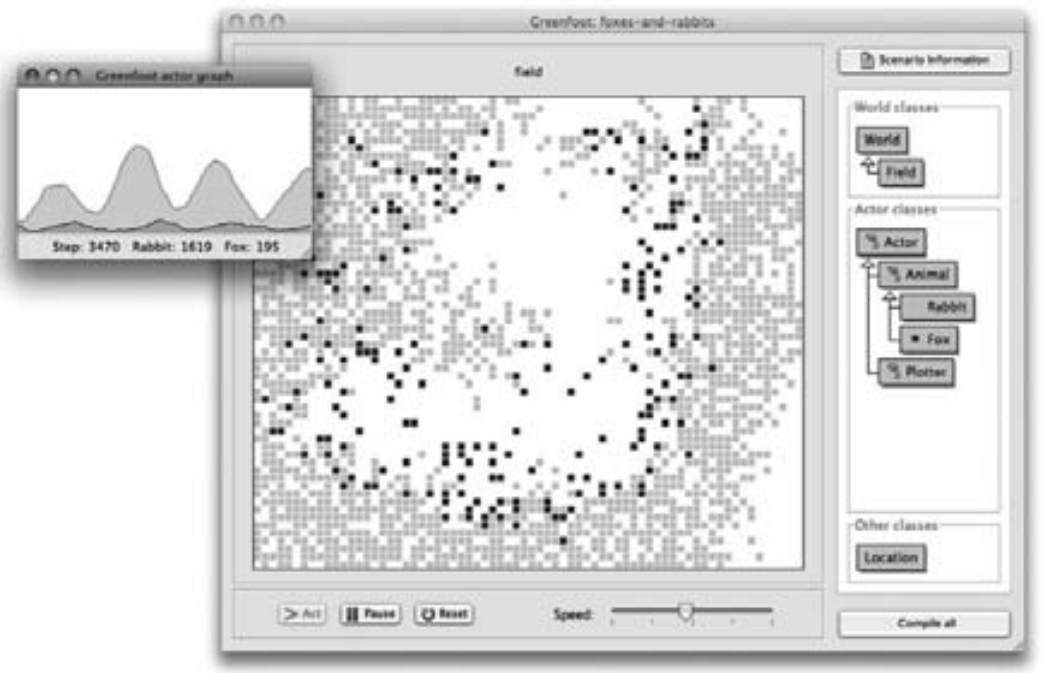


Figura 9.1
La simulación de zorros y conejos

Los zorros se mueven y se reproducen de forma similar a los conejos (aunque se reproducen con menos frecuencia y tienen menos crías). Los zorros pueden hacer una cosa adicional: cazar. Si tienen hambre, y ven un conejo próximo, se mueven para comerse el conejo.

Los zorros también pueden morir de dos formas: se mueren de viejos o se mueren de hambre. Si no encuentran un conejo que comerse durante algún tiempo, mueren. (Se asume que los conejos siempre encuentran suficiente cantidad de comida.)

Ejercicio 9.1 Abre el escenario *foxes-and-rabbits*. Ejecútalo. Explica los patrones de las poblaciones y el movimiento que ves que emerge en el campo.

Ejercicio 9.2 Notarás que este escenario muestra una segunda ventaja pequeña con un gráfico de la población. Una curva muestra el número de zorros, la otra el número de conejos. Explica la forma de estas gráficas.

Como hemos visto, la simulación es altamente simplista en varios aspectos: los animales no tienen que encontrar parejas para reproducirse (pueden hacer todo ellas mismas), las fuentes de comida para los conejos no se incluyen en la simulación, otros factores de muerte (como enfermedades) se ignoran, y muchos factores se dejan fuera. Sin embargo, los parámetros que simulamos son bastante interesantes, y podemos realizar algunos experimentos con ellos.

Ejercicio 9.3 ¿Son las poblaciones actuales estables? Esto es, ¿termina la ejecución siempre con una de las especies extinguida? Si las especies se extinguen, ¿cuál es el tiempo medio que sobreviven?

¿Importa el tamaño del campo? Por ejemplo, imagina que tenemos un parque nacional con especies en peligro de extinción. Y alguien quiere construir una autopista en el medio del parque, de forma que los animales no pueden cruzar, con lo que se divide en dos mitades iguales el parque. Los que proponen la autopista podrían argumentar que esto no importa porque el tamaño total del parque es el mismo que antes. La autoridad del parque podría exponer que esto es malo porque reduce a la mitad el tamaño de cada parque. ¿Quién tiene razón? ¿Importa el tamaño del parque? Haz algunos experimentos.

Ejercicio 9.4 La clase `Field` (*Campo*) tiene definiciones de constantes al principio del código fuente para su alto y ancho. Modifica estas constantes para cambiar el tamaño del campo. ¿Afecta el tamaño del campo a la estabilidad de las poblaciones? ¿Se extinguen las especies más fácilmente si el campo es mayor o menor? ¿O no hay diferencia?

Otros parámetros con los que podemos experimentar son las constantes definidas al principio de las clases `Rabbit` (*Conejo*) y `Fox` (*Zorro*). Los conejos tienen definiciones para su edad máxima, la edad a partir de la que pueden tener crías, la frecuencia de reproducción (definida como una probabilidad en cada paso), y el tamaño máximo de su camada cuando se reproducen. Los zorros tienen los mismos parámetros y uno adicional: el valor nutricional de un conejo cuando es comido (expresado como el número de pasos que el zorro puede sobrevivir después de comerse un conejo). El nivel de comida de un zorro decrece en cada paso y se incrementa cuando se come un conejo. Si llega a cero, el zorro se muere.

Ejercicio 9.5 Escoge un tamaño de campo en que las poblaciones sean casi estables pero ocasionalmente se extingan. Después haz cambios en los parámetros de `Rabbit` para intentar incrementar las oportunidades de supervivencia de la población. ¿Puedes encontrar la configuración que hace las poblaciones más estables? ¿Hay algún parámetro que la haga menos estable? ¿Son los efectos observados los que esperabas, o difieren de tus expectativas?

Ejercicio 9.6 Cuando la población de zorros está en peligro de extinción de vez en cuando, podríamos especular que se pueden mejorar las oportunidades de que sobrevivan los zorros incrementando el valor de comida de los conejos. Si los zorros pueden vivir más al comer un solo conejo, se morirían de hambre menos a menudo. Investiga esta hipótesis. Dobla la cantidad de la constante **RABBIT_FOOD_VALUE** y prueba. ¿Sobrevive más tiempo la población de zorros? Explica el resultado.

Ejercicio 9.7 Haz otros cambios en los parámetros de `Fox`. ¿Puedes hacer las poblaciones más estables?

Los ejercicios muestran que podemos experimentar con esta simulación cambiando algunos parámetros y observando sus efectos. En este capítulo, sin embargo, no nos contentaremos con experimentar con una simulación existente, sino que queremos desarrollar una propia.

Haremos esto en la siguiente sección con un escenario diferente: *hormigas*.

9.2 Hormigas

El escenario *ants* (*hormigas*) simula la conducta de recolección de comida de las colonias de hormigas. O, para ser más precisos, nos gustaría simular esta conducta, pero no lo hace todavía. Lo desa-

rrollaremos para que lo haga. En su estado actual, el escenario se ha preparado hasta cierto punto: tiene gráficos y parte de la implementación está terminada. Falta, sin embargo, la funcionalidad principal, y trabajaremos para completarla.

Ejercicio 9.8 Abre el escenario llamado *ants* de la carpeta *chapter09* de los escenarios del libro. Crea un hormiguero (*ant hill*) en el mundo y ejecuta el escenario. ¿Qué observas?

Ejercicio 9.9 Examina el código fuente de la clase `Ant`. ¿Qué hace actualmente?

Ejercicio 9.10 `Ant` es una subclase de la clase `Creature` (*Criatura*). ¿Qué funcionalidad tiene una criatura?

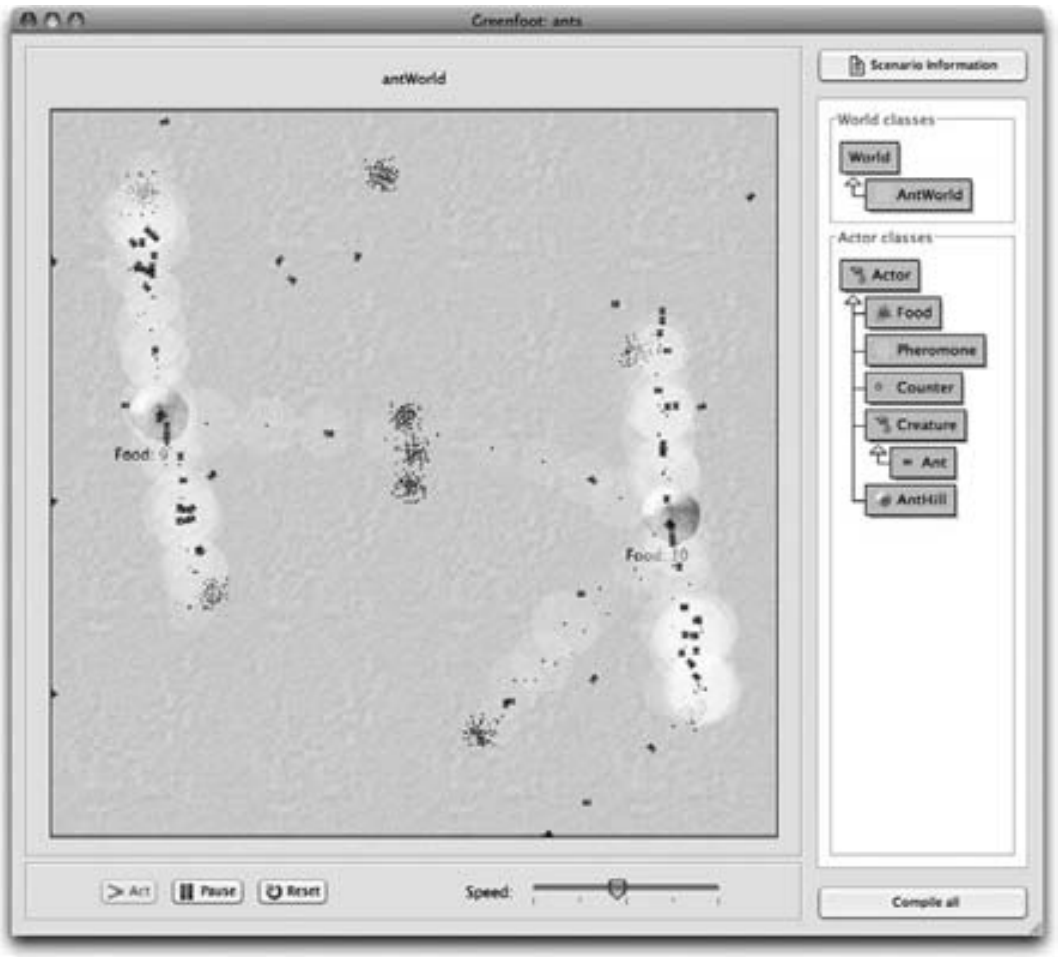


Figura 9.2
La simulación de hormigas

Ejercicio 9.11 Examina el código fuente de la clase `AntHill`. ¿Qué hace esta clase? ¿Cuántas hormigas están en un hormiguero?

La primera cosa que observamos es que las hormigas no se mueven. Se crean en el medio del hormiguero, pero como no se mueven, después de un pequeño instante, se sientan todas unas sobre otras. Por tanto, la primera cosa que debemos hacer es conseguir que las hormigas se muevan.

Ejercicio 9.12 En el método `act` de las hormigas, añade una línea de código que haga que la hormiga se mueva. Consulta la documentación de la clase `Creature` para averiguar qué métodos usar.

Iremos ahora paso a paso a través de las mejoras de este escenario. Los pasos que tenemos que dar son los que siguen:

- Introduciremos algo de comida en el escenario, para que las hormigas tengan algo que recolectar.
- Mejoraremos las hormigas, de forma que puedan encontrar la comida y llevar algo de comida a casa.
- A continuación, añadiremos una clase `Pheromone` (*Feromona*). Las feromonas son sustancias químicas que producen algunos animales para dejar mensajes a otros animales de sus especies.
- Mejoraremos las hormigas para hacer uso de las feromonas. Dejarán gotas de feromonas en el suelo cuando han encontrado comida, y otras hormigas podrán oler estas feromonas y modificar por dónde tienen que ir.

Estos pasos juntos simulan de forma aproximada la conducta de recolección de comida en colonias de hormigas. Cuando los hayamos terminado, podremos hacer algunos experimentos con la simulación.

9.3 Recogiendo comida

Nuestra primera tarea es crear algo de comida en nuestro escenario y dejar que las hormigas lo recolecten y lo lleven al hormiguero.

Ejercicio 9.13 Crea una nueva clase llamada `Food` (*Comida*). La clase no necesita una imagen fija. Dibujaremos una imagen desde la clase.

Cada objeto de la clase `Food` representa una pila de migas de comida. Nuestro plan es crear una nueva imagen dibujada dinámicamente para el objeto `Food` y dibujar un pequeño punto encima de ella para cada miga en la pila. Una pila puede comenzar con, digamos, 100 migas, y cada vez que una hormiga encuentra la pila, se lleva unas pocas migas. Esto significa que la imagen debe ser redibujada con menos migas cada vez que una hormiga coge algo de comida.

Ejercicio 9.14 En la clase `Food`, crea un campo para el número de migas que tiene actualmente la pila. Inicialízalo a 100.

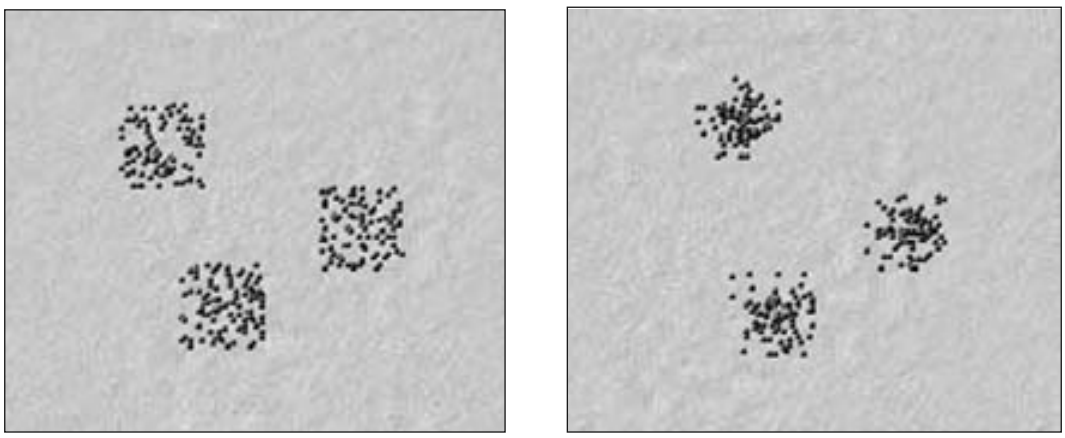
Ejercicio 9.15 Crea un nuevo método privado denominado `updateImage` (*actualizaImagen*), que cree una imagen de un tamaño fijo y pinte un punto encima de ella en una posición aleatoria para cada miga que esté en la pila. Escoge un tamaño para la imagen que pienses que queda bien. Llama a este método desde el constructor.

Si terminaste el ejercicio anterior, y has colocado migas de comida en una posición aleatoria de la imagen `Food` (usando el método `Greenfoot.getRandomNumber` para obtener las coordenadas), entonces te habrás dado cuenta de que la pila de migas tiene una forma algo cuadrada (Figura 9.3a). Esto se debe a que la imagen misma es cuadrada, y las migas se distribuyen de forma uniforme por la imagen.

Si queremos cambiar esto para que parezca más una pila (con la mayoría de migas en el medio y otras migas en forma de círculos alrededor de él, Figura 9.3b), entonces podemos usar otro método de números aleatorios para colocar las migas de forma diferente. Haremos esto en los Ejercicios 9.16 y 9.17. Observa que estos ejercicios son más avanzados y puramente cosméticos: simplemente cambian el aspecto de la pila de comida, y nada de su funcionalidad, por lo que puedes saltarlos sin repercusiones.

Ejercicio 9.16 Consulta la documentación de la API de la Librería Estándar de Clases Java. Encuentra la documentación de la clases `Random` del paquete `java.util`. Los objetos de esta clase son *generadores de números aleatorios* que son más flexibles que el método `getRandomNumber` de Greenfoot. El método que nos interesa es el que genera números aleatorios según una distribución gaussiana (también llamada «distribución normal»). ¿Cuál es el nombre del método, y qué hace?

Ejercicio 9.17 En tu clase `Food`, cambia cómo colocas las migas de comida en la imagen para hacer uso de la distribución gaussiana de números aleatorios. Para esto, tienes que usar un objeto `java.util.Random` para crear los números aleatorios en vez de `Greenfoot.getRandomNumber`.



a) migas con distribución uniforme

b) migas con distribución gaussiana

Figura 9.3
Colocación de migas de comida con diferentes algoritmos aleatorios

Nota al margen. Distribuciones aleatorias

Si necesitamos conducta aleatoria en nuestros programas, a veces es importante pensar qué tipo de distribución aleatoria necesitamos. Muchas funciones aleatorias, tales como el método `Greenfoot.getRandomNumber`, producen una *distribución uniforme*. En esta distribución, la probabilidad de ocurrencia de cada resultado posible es la misma. La función gaussiana de números aleatorios nos da una *distribución normal*. Esta distribución es la que en los resultados medios son los más probables, y los más extremos los más raros.

Si nosotros, por ejemplo, programamos un juego de dados, necesitamos una distribución uniforme. Cada cara de un dado sucede con la misma probabilidad. En otro caso, si modelamos la velocidad de coches en una simulación de tráfico, sería mejor una distribución normal. La mayoría de los coches se conducen próximos a la velocidad media, y sólo algunos van muy rápido o muy lentos.

A continuación, necesitamos añadir una funcionalidad para eliminar las migas de la pila de comida, de forma que las hormigas puedan coger algo de comida.

Ejercicio 9.18 Añade un método público a la clase `Food` que elimine unas pocas migas de la pila. Asegúrate de que la imagen se repinta con el número correcto de migas restantes. Cuando se acaben todas las migas, el objeto `Food` debería eliminarse él mismo del mundo. Prueba este método interactivamente.

Ahora que tenemos una pila de comida disponible en nuestro escenario, haremos que nuestras hormigas la recolecten. Las hormigas conmutarán entre dos conductas diferentes:

- Si no están llevando comida, buscan comida.
- Si están actualmente llevando comida, andan hacia casa.

Las hormigas conmutan entre estas dos conductas cuando alcanzan o bien la pila de comida o bien el hormiguero. Si están buscando comida, y encuentran una pila de comida, cogen algo de comida y conmutan de la primera a la segunda conducta. Si llegan al hormiguero, dejan la comida allí y vuelven a cambiar al primer patrón de conducta.

Implementaremos ahora esto en nuestra clase `Ant`. Escrito en pseudocódigo, el método `act` podría ser algo como esto:

```
if (llevando comida) {
    anda hacia casa;
    comprueba si estamos en casa;
}
else {
    busca comida;
}
```

Concepto:

Usar **métodos cortos** con un propósito específico conduce a una mejor **calidad de código**.

Vamos a implementar esto ahora cada paso.

Presta atención a la calidad de tu código: usa métodos cortos con propósitos diferentes y asegúrate de comentar bien tus métodos. No escribas demasiado código en un solo método.

Puedes encontrar una implementación de las funcionalidades vistas hasta ahora en el escenario *ants-2*. Después de completar tu implementación (o cuando te bloquees), podrías querer comparar tu solución con la nuestra.

Ejercicio 9.19 En la clase `Ant`, implementa un método `searchForFood` (*buscaComida*). Este método debería inicialmente simplemente hacer que dé un paseo aleatorio y compruebe si hemos encontrado una pila de comida. Si la encontramos, paramos la ejecución. (Esto es simplemente comprobar si hemos detectado la comida correctamente.)

Ejercicio 9.20 Añade una funcionalidad para recoger algo de comida cuando encontramos una pila de comida (en vez de parar la ejecución). Necesitamos quitar algunas migas de la pila de comida (deberíamos tener ya un método para esto), observa que estamos ahora transportando comida (necesitamos probablemente un campo para esto), y cambia la imagen de la hormiga. Ya hay una imagen preparada en el escenario, llamada *ant-with-food.png*, que puedes usar.

Ejercicio 9.21 Asegúrate de que la hormiga anda hacia casa cuando transporta comida.

Ejercicio 9.22 Implementa un método que compruebe si una hormiga ha llegado al hormiguero. Si ha llegado a casa, debería dejar su comida. Dejar la comida consiste en anotar que ya no se tiene que llevar comida (incluye volver a cambiar la imagen) y llamar al método `countFood` de `AntHill` para apuntar que se ha recolectado esta miga de comida.

9.4 Creando el mundo

Antes de que nos pongamos a añadir feromonas a nuestro escenario, hagamos primero un código de inicialización que cree algunos hormigueros y comida automáticamente, de forma que no tengamos que repetir esto manualmente cada vez que queremos hacer pruebas.

Ejercicio 9.23 Añade código a la clase `AntWorld` (*MundoHormiga*), para que cree automáticamente dos hormigueros y unas pocas pilas de comida en el mundo.

9.5 Añadiendo feromonas

Ahora que tenemos una buena configuración inicial, estamos preparados para añadir feromonas. Cada objeto feromona es una pequeña gota de una sustancia química que las hormigas dejan en el suelo. Esta gota se evaporará bastante rápidamente y entonces desaparece.

Las hormigas dejan feromonas cuando andan de vuelta a casa desde una fuente de comida. Cuando otras hormigas huelen una gota de feromona, pueden salir del hormiguero y andar hacia la comida.

Ahora tenemos una clase `Pheromone` disponible para que nuestras hormigas la puedan usar. Ahora tenemos sólo que conseguir que las hormigas la usen. La primera tarea para usar una feromona es dejarla en nuestro mundo. (La segunda tarea es darse cuenta de ella y cambiar nuestra dirección como resultado.) Hagamos la primera tarea en primer lugar.

Ejercicio 9.24 Crea una clase `Pheromone` (*Feromona*). Esta clase no necesita una imagen —dibujaremos la imagen programáticamente.

Ejercicio 9.25 Implementa un método `updateImage` (*actualizaImagen*) en la clase `Pheromone`. Inicialmente, este método debería crear una imagen con círculo blanco pintada sobre ella, y fijar ésta como la imagen del actor. El círculo blanco debería ser parcialmente transparente. Llama a este método desde el constructor.

Ejercicio 9.26 Da a las feromonas un atributo `intensity` (*intensidad*). (Esto es, añade un campo `intensity`.) La intensidad de un objeto feromona debería comenzar con una intensidad máxima definida, y decrecer en cada ciclo de actuación. Cuando la intensidad llegue a 0, elimina el objeto feromona del mundo. Una gota de feromona debería evaporarse tras 180 ciclos de actuación.

Ejercicio 9.27 Modifica tu método `updateImage`, de forma que haga uso de la intensidad de la feromona. Cuando la intensidad decrezca, el círculo blanco que la representa en la pantalla debería ser más pequeño y más transparente. Asegúrate de llamar al método `updateImage` desde el método `act` para que veamos que cambia la imagen en la pantalla.

Ejercicio 9.28 Prueba tus feromonas colocándolas manualmente y ejecuta tu simulación.

Ejercicio 9.29 Añade un método a tu hormiga que deje caer una gota de feromona en el mundo. Llama a este método repetidamente mientras tu hormiga está caminando a casa.

Si, en el ejercicio anterior, dejaste una gota de feromona en el ciclo de actuación, te darás cuenta de que se dejan caer demasiadas feromonas en el mundo. Las hormigas no pueden producir cantidades ilimitadas de feromonas. Después de dejar una gota, necesitas algún tiempo para generar más feromonas.

Ejercicio 9.30 Modifica la hormiga para que pueda dejar una gota de feromona cada 18 ciclos. Para conseguir esto, necesitarás un campo que almacene el nivel actual de feromonas de una hormiga. Cuando la hormiga deja una gota de feromona, el nivel de feromonas (feromonas que quedan en el cuerpo de la hormiga) decrece hasta 0, y entonces se incrementa lentamente hasta que la hormiga está preparada para dejar otra gota.

La Figura 9.4 muestra un rastro de feromonas dejadas por nuestra hormiga en este punto [1]. Las gotas están espaciadas (la hormiga necesita algo de tiempo para volver a generar feromonas), y las gotas de feromonas más antiguas están parcialmente evaporadas —son más pequeñas y más transparentes.

La última cosa que tenemos que añadir es que las hormigas huelan las feromonas, y cambien su dirección de movimiento cuando las huelen.

[1] Si lo miras con detalle, verás que yo he modificado mi imagen de feromona para que tenga un pequeño punto negro en el medio. Esto es para que se puedan ver mejor las feromonas incluso cuando son bastante transparentes.

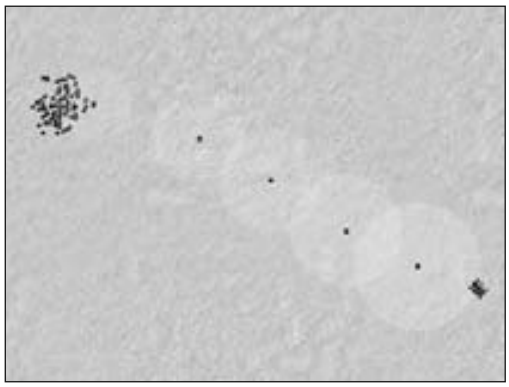


Figura 9.4
Una hormiga dejando un rastro de feromonas

Si una hormiga huele una gota de feromona, debería salir de su hormiguero durante un tiempo limitado. Si no encuentra comida o huele una nueva gota de feromona tras un tiempo, debería volver a andar de forma aleatoria. Nuestro algoritmo para buscar comida podría ser algo como esto:

```
if (hemos encontrado recientemente una gota de feromona) {
     sal de casa;
}
else if (olemos feromonas ahora) {
     anda hacia el centro de la gota de feromona;
     if (estamos en el centro de la gota de feromona) {
         nota que hemos encontrado feromonas;
     }
}
else  {
     anda de forma aleatoria;
}
comprueba si hay comida;
```

Cuando implementes esto en tu propio escenario, acuérdate de crear un método separado para cada subtarea. De esta forma, tu código quedará bien estructurado, fácil de entender, y fácil de modificar.

Ejercicio 9.31 Implementa la funcionalidad vista arriba en tu propio escenario. Cuando las hormigas huelan feromonas, saldrán de su hormiguero durante los siguientes 30 pasos, antes de volver a su conducta por defecto.

Si has completado este ejercicio, entonces tu simulación de hormigas está más o menos terminada (tanto como cualquier aplicación software esté alguna vez completa). Si ejecutas tu escenario ahora, deberías ver que las hormigas forman caminos en las fuentes de comida.

9.6 Formando un camino

Un aspecto interesante de este escenario es que no hay código en ninguna parte del proyecto que hable sobre la formación de caminos. La conducta de las hormigas individuales es bastante simple («si tienes comida, ve a casa; si hueles feromonas, sal; en otro caso, ve a cualquier sitio»). Sin embargo, juntas, las hormigas exhiben una conducta bastante sofisticada: forman caminos estables,

Concepto:

Las simulaciones de sistemas a menudo muestran **conducta emergente**. Esta conducta no está programada en actores individuales sino que emergen del resultado de la suma de todas las conductas.

refrescando las feromonas cuando se evaporan, y transportan eficientemente la comida de vuelta a su hormiguero.

Esto se conoce como una *conducta emergente*. Esta conducta no se programa en un actor individual, sino que la conducta del sistema emerge de las interacciones de los muchos (bastante simples) actores.

La mayoría de sistemas complejos muestran algún tipo de conducta emergente del sistema, ya sean sistemas de tráfico en ciudades, redes de ordenadores, o multitudes de personas. Predecir estos efectos es muy difícil, y las simulaciones de ordenador nos pueden ayudar a entender estos sistemas.

Ejercicio 9.32 ¿Cómo de realista es nuestra simulación en el uso de feromonas por las hormigas? Investiga un poco en el uso real de feromonas por las colonias de hormigas y anota qué aspectos de nuestra simulación son realistas, y dónde hemos hecho simplificaciones.

Ejercicio 9.33 Asume que la polución ha introducido una sustancia tóxica en el entorno de las hormigas. El efecto es que su producción de feromonas se reduce a la cuarta parte de la cantidad previa. (El tiempo que pasa entre la suelta de dos gotas de feromonas dura cuatro veces más.) ¿Serán aún capaces de formar caminos? Pruébalo.

Ejercicio 9.34 Asume que otro contaminante ha decrementado la habilidad de las hormigas para recordar que olieron recientemente una feromona a un tercio. En lugar de 30 pasos, sólo pueden recordar la feromona durante 10 pasos. ¿Cuál es el efecto de esto en su conducta?

Hay muchos otros experimentos que podemos hacer. El más obvio es intentar diferentes ubicaciones para hormigueros y fuentes de comida, y diferentes valores para los atributos que determinan la conducta de las hormigas.

El escenario *ants-3* en la carpeta *chapter09* muestra una implementación de las tareas descritas antes. Incluye tres métodos diferentes de inicialización en la clase del mundo que se pueden llamar interactivamente desde el menú emergente de `AntWorld`.

9.7 Resumen

En este capítulo, hemos visto dos ejemplos de simulaciones. Nos han servido para dos propósitos. Primero, tuvimos la oportunidad de practicar muchas de las técnicas de programación que hemos visto en los capítulos anteriores, y tuvimos que usar la mayoría de los constructores Java previamente introducidos. En segundo lugar, las simulaciones son un tipo interesante de aplicación con la que experimentar. Muchas simulaciones se usan en la vida real con diferentes propósitos, tales como previsión del tiempo, planificación de tráfico, estudios de impacto medioambiental, investigación física, y muchos más.

Si conseguiste resolver todos los ejercicios de este capítulo, entonces has llegado a entender gran parte de lo que este libro te intentaba enseñar, y eres competente en los fundamentos de programación.

Resumen de conceptos

- Una **simulación** es un programa de ordenador que simula algunos fenómenos del mundo real. Si las simulaciones son suficientemente precisas, observándolas podemos aprender cosas interesantes sobre el mundo real.
- Usar **métodos cortos** con un propósito específico conduce a mejor **calidad de código**.
- Las simulaciones de sistemas a menudo muestran **conducta emergente**. Esta conducta no se programa con actores individuales, sino que es el resultado de la suma de todas las conductas.

CAPÍTULO

10 Más ideas de escenarios

temas:	ideas para más escenarios
conceptos:	(no se introducen nuevos conceptos)

Éste es el último capítulo de este libro. Es diferente de los otros capítulos porque no intenta enseñarte ningún concepto o técnica de programación nuevos. En su lugar, presenta brevemente unos escenarios adicionales para darte ideas de algunas cosas que te podría gustar investigar y ponerte a trabajar en ellas.

Todos los escenarios presentados aquí también están disponibles como proyectos de Greenfoot con código fuente en los escenarios del libro. Sin embargo, la mayoría de ellos no ofrecen implementaciones completas de la idea que representan.

Algunos escenarios son casi completos, y puede gustarte estudiarlos para aprender nuevas técnicas y ver cómo se han conseguido algunos efectos. Otros son comienzos, implementaciones parciales de una idea que pueden servir como un punto inicial para tu proyecto. Incluso otros son ilustraciones de un único concepto o idea que podría proporcionar inspiración para algo que podrías incorporar en uno de tus propios escenarios.

En resumen, mira estos escenarios como una colección de ideas para proyectos futuros, y estúdialos para tener una pequeña visión de qué más es posible que consiga un programador competente.

10.1 Canicas

El escenario *marbles —canicas—* (Figura 10.1) implementa un juego en que tú lanzas una bola dorada sobre un tablero con el objetivo de sacar del tablero todas las bolas plateadas en un número limitado de movimientos. El juego está razonablemente completo.

Algunas cosas que merece la pena observar sobre este escenario. La primera cosa que se puede destacar es que tiene muy buena presentación visual. Esto no tiene mucho que ver con Java o programación con Greenfoot y se debe principalmente al uso de gráficos bonitos. Usar gráficos y sonidos bien diseñados puede marcar una gran diferencia en el atractivo de un juego.

Marbles usa una bonita imagen de fondo (el tablero y la voluta del mensaje de texto) y actores con sombras semitransparentes (las canicas y los obstáculos).

El otro aspecto interesante es examinar la detección de colisiones. Las canicas no usan ningún método de detección de colisiones proporcionado por Greenfoot, ya que todos trabajan con imágenes de actores rectangulares. Las canicas, en cambio, son redondas, y necesitamos precisar las colisiones en este caso.

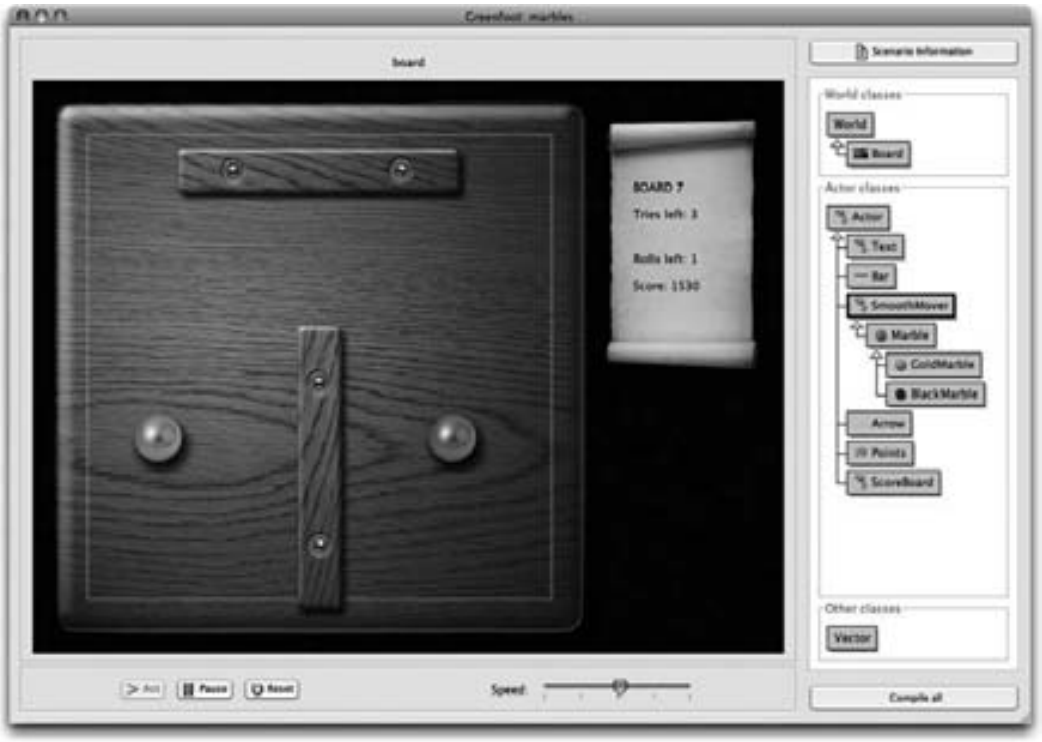


Figura 10.1
El juego de canicas

Afortunadamente, cuando los actores son redondos, no es muy difícil. Dos canicas chocan si su distancia (medida desde sus centros) es menor que su diámetro. Sabemos el diámetro, y la distancia se calcula fácilmente (usando el teorema de Pitágoras).

La siguiente cosa interesante es el modo en que se calcula la nueva dirección de movimiento de una canica que choca. Aquí hemos usado un poco de trigonometría, pero si conoces los principios, no es muy difícil.

Las colisiones con obstáculos fijos son más fáciles, ya que siempre son rectángulos orientados horizontal o verticalmente. Por tanto, una canica chocando con uno de estos obstáculos simplemente invierte su dirección en uno de los ejes (x o y).

Podrías reutilizar la lógica de colisión de canicas para todo tipo de juegos que incluyan colisión con objetos redondos.

10.2 Ascensores

El escenario *lifts —ascensores—* (Figura 10.2) muestra una simulación simple de un ascensor. Muestra varios pisos de un edificio y tres ascensores subiendo y bajando. La gente aparece en los pisos y llama al ascensor y entra en los ascensores cuando llega.

Esta implementación es muy rudimentaria y está sin acabar. La mayoría de lo que vemos en la pantalla es engañoso: no se simula de forma adecuada y lo que pasa está escrito sólo para mostrar el efecto.

Por ejemplo, la gente no entra de forma adecuada en los ascensores (se eliminan cuando un ascensor llega a un piso). El número de personas mostradas en un ascensor es simplemente un número aleatorio. Los ascensores tampoco reaccionan a los botones de llamada —simplemente suben y bajan de forma aleatoria—. No se ha implementado un algoritmo de control de los ascensores.

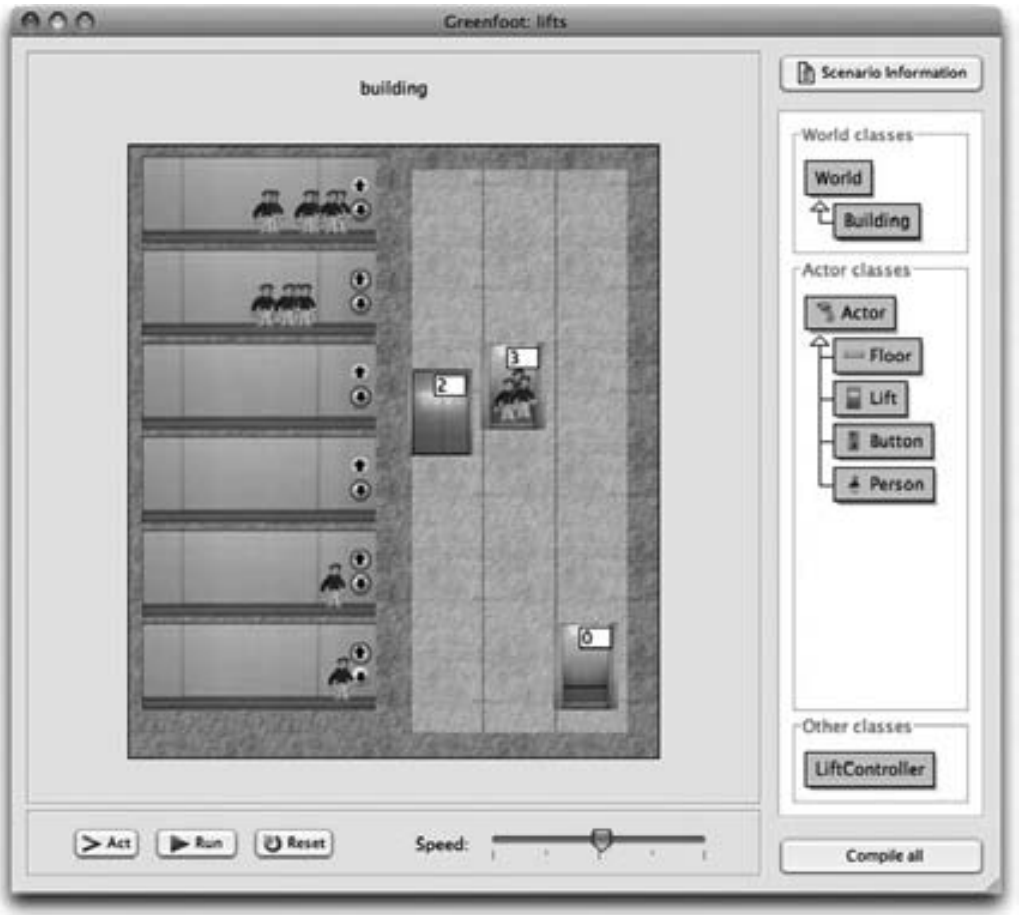


Figura 10.2
Una implementación (parcial) de un ascensor

Por tanto, es sólo una demo rápida que presenta la idea y los gráficos. Para terminar el proyecto, el movimiento de las personas tendría que ser modelado de forma adecuada (entrada y salida en los ascensores). Y entonces podríamos experimentar y probar diferentes algoritmos de control del ascensor.

10.3 Boids

El ejemplo *boids* (Figura 10.3) muestra una simulación de conducta gregaria de pájaros.

El término «boids» viene de un programa desarrollado en 1986 por Craig Reynolds que implementó por primera vez esta conducta gregaria. En ella, cada pájaro vuela según tres reglas:

- Separación: alejarse de otros pájaros si están muy cerca.
- Alineación: volar hacia la cabeza media de los otros pájaros vecinos.
- Cohesión: moverse hacia la posición media de los otros pájaros vecinos.

Con este algoritmo, los pájaros desarrollan un movimiento que es bastante bonito de ver. Se incluye también en este escenario que eviten obstáculos: intentan no volar contra los árboles.

Una versión de este algoritmo fue usado, por ejemplo, en la película «Batman vuelve» de Tim Burton en 1992 para bandadas de murciélagos y manadas de pingüinos, y en la película «El Señor de los Anillos» para crear los movimientos de los ejércitos de Orcos.

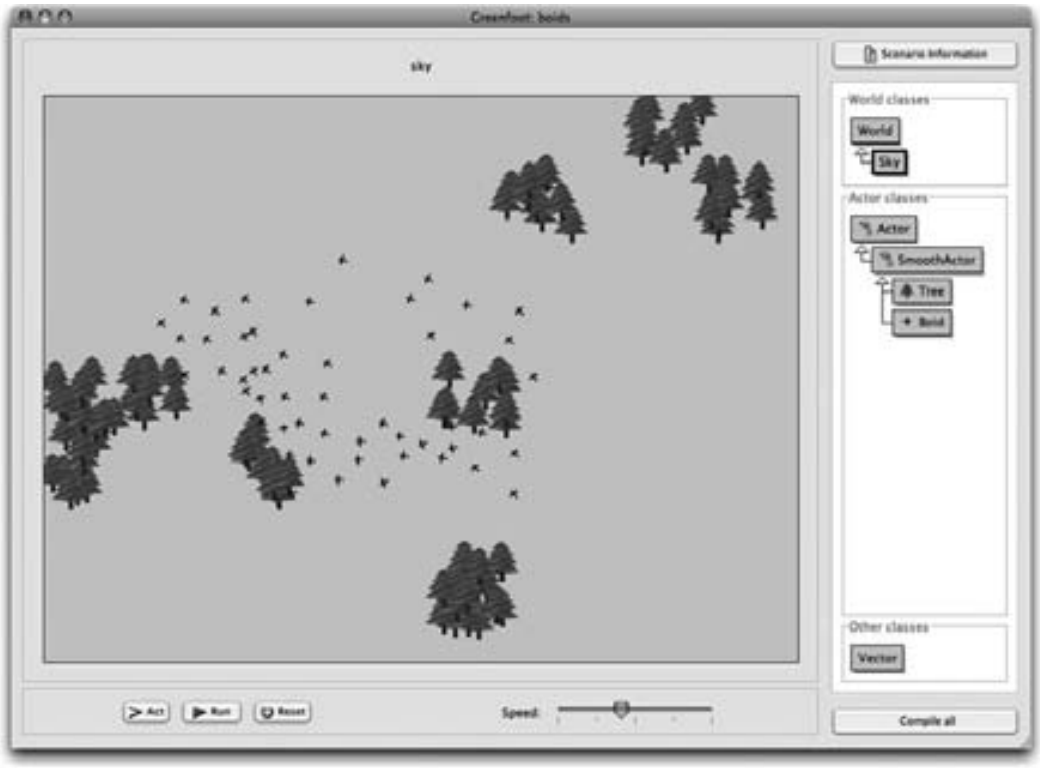


Figura 10.3
Boids: una simulación de conducta gregaria

La versión de este libro ha sido escrita por Poul Henriksen.

Puedes encontrar mucho más sobre esto buscando «boids» en la web. Y aunque este escenario actualmente no hace nada más que mostrar el movimiento, uno siente que hay un juego dentro en algún sitio…

10.4 Círculos

Círculos (Figura 10.4) es un proyecto que no parece tener un propósito especial, pero es interesante para jugar con él y ver su aspecto visual.

Usa una simulación física, tal como la gravedad y rebotar en los bordes, y alguna aleatoriedad para crear bonitas imágenes que se mueven. ¿Es física? ¿Es arte? ¿O puede que un poco de ambos?

Lo que quiera que sea, estoy seguro de que hay muchas otras formas de producir interesantes o bellas imágenes, patrones o animaciones de colores. (Una idea podría ser combinar los círculos con el algoritmo de detección de colisiones del juego de las canicas.)

Círculos ha sido escrito por Joe Lenton.

10.5 Explosión

El escenario *explosión* (Figura 10.5) demuestra cómo podemos implementar un efecto más espectacular de una explosión. El objeto que explota es, en este caso, una simple roca que ya encontramos en otros escenarios antes. (Jugó, por ejemplo, el papel de asteroide en el escenario *asteroides*.) Pero podríamos explotar en realidad cualquier cosa que quisiéramos.

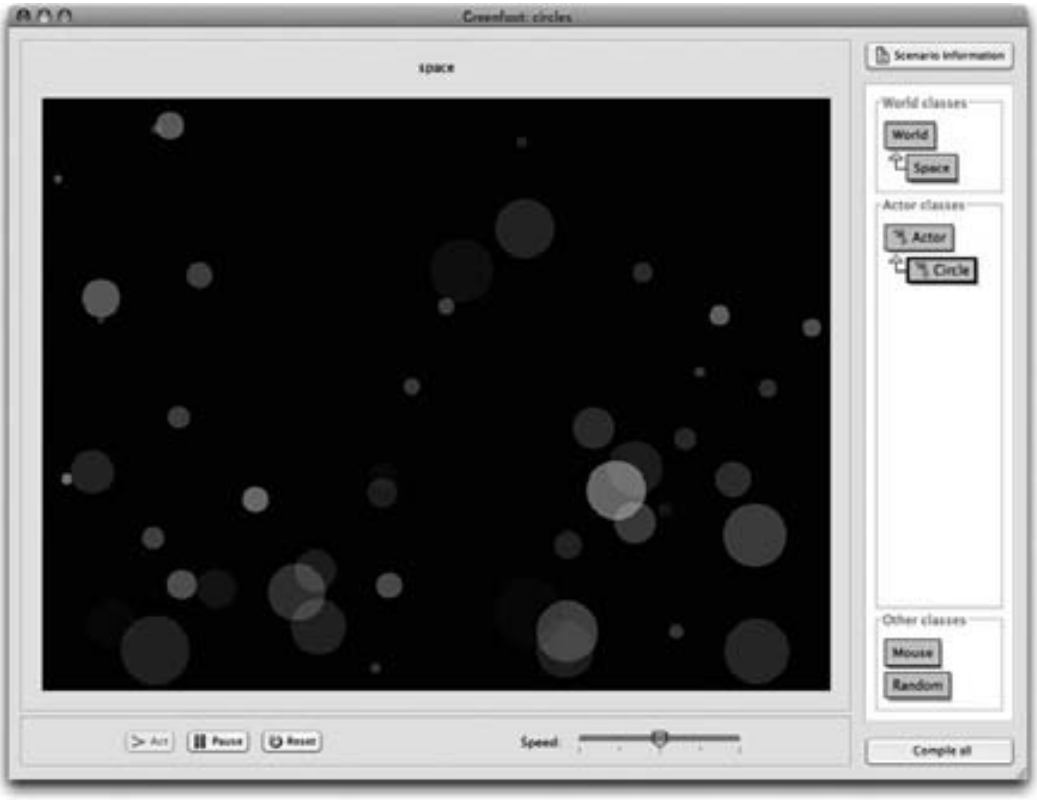


Figura 10.4
«Círculos» es una mezcla de física y arte

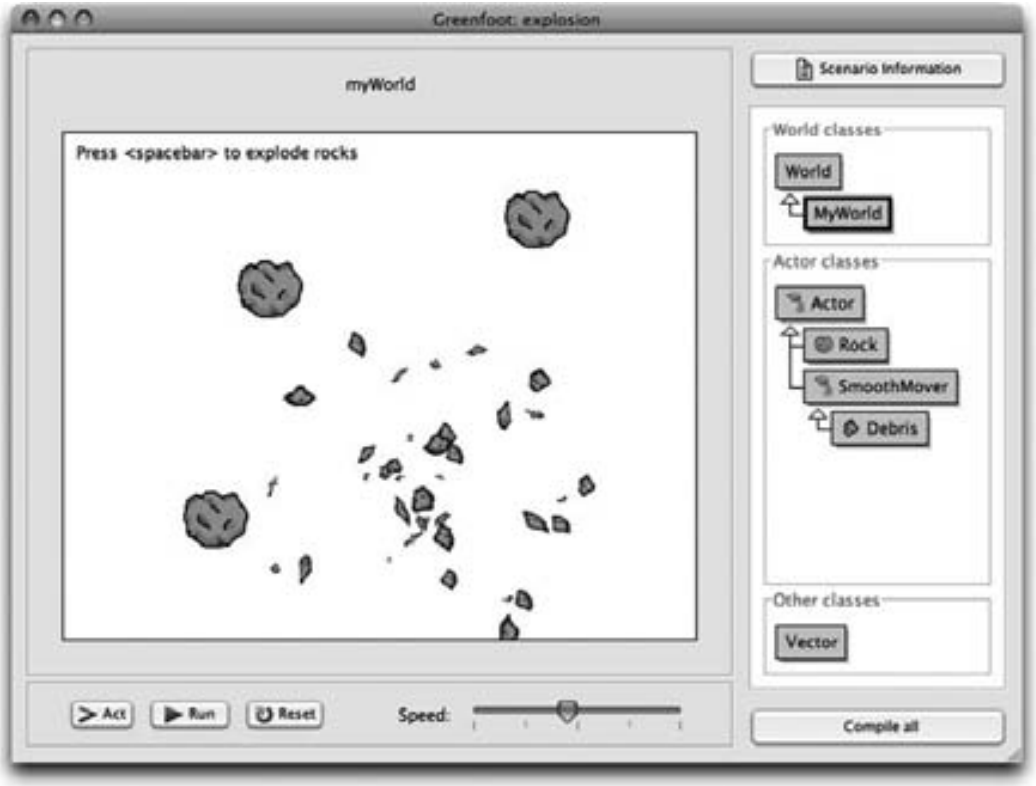


Figura 10.5
Un efecto de explosión

Para conseguir este efecto, tenemos una clase `Debris` (*Fragmento*) que representa una parte de la roca. Cuando la roca explota, la eliminamos del mundo y colocamos 40 piezas de fragmentos en su lugar.

Cada pieza de fragmento se estira y rota para hacer que parezca algo único, e inicialmente tienen un vector de movimiento con una dirección aleatoria. En cada paso, añadimos un poco de movimiento hacia abajo para simular la gravedad, y el resultado es la explosión que puedes ver cuando ejecutas el escenario.

Un videotutorial explicando este escenario con más detalle está disponible en el sitio web de Greenfoot en

`http://www.greenfoot.org/doc/videos.html.`

10.6 Breakout

«Breakout» (Figura 10.6) es un juego clásico de ordenador en que el jugador controla una paleta en la parte inferior de la pantalla para rebotar una pelota hacia arriba y eliminar algunos bloques. Si no conoces el juego, podrás obtener más información rápidamente.

Figura 10.6
Comienzo de un juego «breakout»

El escenario *breakout* es una implementación parcial de este juego. Usa la pelota con el efecto de humo que vimos en el Capítulo 8 y añade una paleta para que el jugador controle la pelota. No tiene, sin embargo, bloques a los que apuntar, por lo que en su forma actual no es muy interesante.

Se han creado muchas variaciones de *breakout* con el tiempo. Muchas usan patrones diferentes para disponer los bloques en diferentes niveles. La mayoría también tiene algún tipo de «elevador de potencia» —golosinas escondidas en algunos bloques que descienden cuando se elimina el bloque—. Al coger estas golosinas normalmente ocurre algo interesante en el juego (bolas extra, incremento de velocidad, paleta más grande o más pequeña, etc.).

Completar este juego de una forma interesante puede ser un buen proyecto. Podrá también modificarse para que haya dos paletas, una en cada lado, convirtiéndose en esencia en el juego clásico «Pong».

10.7 Salto de plataforma

Un estilo de juegos muy habitual es el juego de «plataforma». Los jugadores normalmente controlan un personaje del juego que tiene que moverse de un área de la pantalla a otra, mientras evita varios obstáculos. Uno de estos obstáculos puede ser una brecha en el suelo por el que anda el personaje, con algún medio de saltar sobre él.

El escenario *pengu* (Figura 10.7) implementa una pequeña parte de estos juegos. Hay dos piezas en el suelo en cada lado de la pantalla, y el pingüino puede cruzarlo saltando sobre una nube en movimiento.

Este escenario se incluye aquí para demostrar cómo un actor se puede mover por encima de otro (el pingüino está encima del suelo), y cómo se podría implementar que salten o se caigan.

Figura 10.7
Comienzo de un juego simple de salto de plataformas

Un videotutorial que trata de esto en más detalle está disponible en el sitio web de Greenfoot en `http://www.greenfoot.org/doc/videos.html`, con el nombre «Running, jumping and falling».

10.8 Onda

El último escenario presentado aquí se llama *wave —onda—* (Figura 10.8). Es una simulación simple de la propagación de una onda en una cadena. Juega con él un rato, y verás qué hace.

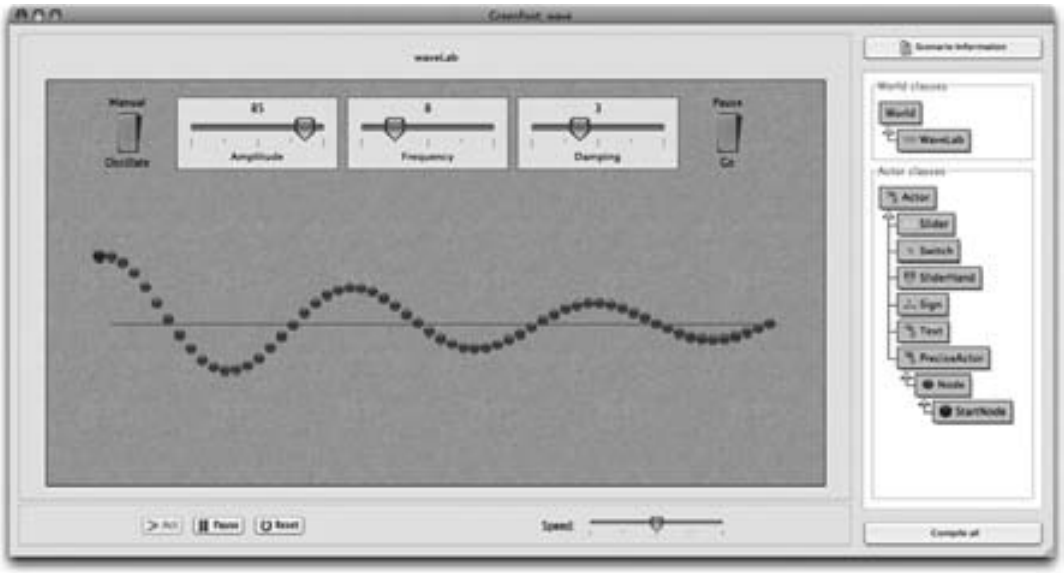


Figura 10.8
Simulación de la propagación de una onda en una cadena de cuentas

Uno de los aspectos fascinantes de este ejemplo es cómo una implementación bastante simple —en cada ronda de actuación, cada cuenta simplemente se mueve hacia el medio de sus dos vecinas— consigue una simulación bastante sofisticada de varios aspectos de propagación de ondas.

Este ejemplo se incluye aquí para ilustrar que, con un poco de razonamiento y preparación, se podrían simular varias conductas de otras disciplinas. En este caso, es un simple efecto físico. De igual forma, se podrían simular reacciones químicas, interacciones biológicas, interacciones de partículas subatómicas, y muchas más. Con una planificación cuidadosa, podemos aprender de otras áreas, a la vez que aprendemos a programar más.

Este escenario también implementa controles de barra deslizante y conmutador, que podrían ser útiles en otros proyectos.

10.9 Resumen

En este capítulo que concluye el libro, hemos intentado mostrar que hay muchas direcciones que puedes seguir más allá de los pocos ejemplos que hemos visto con más detalle a través del libro.

A medida que tengas más experiencia, tendrás más confianza y tendrás mayor capacidad para convertir tus ideas en realidad desarrollando tus programas. Como programador, te enfrentas a un mundo de infinitas ideas creativas, tanto dentro de Greenfoot, como sin él, usando otros entornos de desarrollo.

Cuando programes en otros entornos, al margen de Greenfoot, tendrás que aprender nuevas habilidades y técnicas, pero todo lo que has aprendido usando Greenfoot te será útil y aplicable.

Si has seguido este libro desde el principio hasta este punto, has aprendido muchos conceptos sobre programación en Java y de hecho sobre programación en un lenguaje orientado a objetos en general. Para aprender a programar, el comienzo siempre es la parte más dura, y ya la has dejado atrás.

Si deseas obtener ayuda e ideas para programar con Greenfoot, puedes hacer uso del sitio web de Greenfoot [1]. Usa la Galería de Greenfoot para publicar tus escenarios, mira el trabajo de otras personas y te dará más ideas. Mira en los videotutoriales para aprender trucos y atajos. Y únete al grupo de discusión para comunicarte con otros programadores de Greenfoot, obtener y dar ayuda, y comentar nuevas ideas.

Esperamos que hayas disfrutado programando tanto como nosotros lo hacemos. Si lo has hecho, un nuevo mundo se encuentra ante ti. ¡Programa, diviértete, y sé creativo!

[1] www.greenfoot.org

APÉNDICE

A

Instalando Greenfoot

Este apéndice te dirá dónde encontrar el software de Greenfoot y los escenarios usados en el libro, y cómo instalarlos.

Para trabajar con los proyectos ejemplo de este libro, necesitarás instalar tres cosas: un sistema Java, el software de Greenfoot, y los escenarios del libro.

A.1 Instalando Java

Descarga Java de `http://java.sun.com/javase/downloads`. Deberías instalar la última versión de Java SE Development Kit (JDK).

En Mac OS X, no es necesario instalar Java —se incluye en la instalación estándar de Mac OS.

A.2 Instalando Greenfoot

Descarga Greenfoot de `http://www.greenfoot.org`, y sigue las instrucciones de instalación.

A.3 Instalando los escenarios del libro

Descarga los escenarios del libro de `http://www.greenfoot.org/book`. Obtendrás un fichero llamado *book-scenarios.zip*. Es un fichero comprimido *zip* que debe descomprimirse. En sistemas Windows, se puede conseguir normalmente pinchando con el botón derecho del ratón y seleccionando *Extraer Todo* del menú. En sistemas Mac OS y Linux, puedes pinchar dos veces en el fichero para extraerlo.

Después de extraer el fichero, obtendrás una carpeta llamada *book-scenarios* guardada en tu sistema de ficheros. Recuerda que debes guardarla —necesitarás abrir los proyectos desde esta carpeta mientras avanzas en el libro.

APÉNDICE

B

API de Greenfoot

La API de Greenfoot consta de cinco clases:

Actor	Los métodos de Actor están disponibles a todas las subclases de Actor.
World	Los métodos de World están disponibles en el mundo.
Greenfoot	Usado para comunicarse con el entorno de Greenfoot.
MouseInfo	Proporciona información sobre los últimos eventos del ratón.
GreenfootImage	Para presentación y manipulación de imágenes.

Class World

`World(int worldWidth, int worldHeight, int cellSize)`	Construye un nuevo mundo.
`void act()`	Método Act del mundo. Se le llama una vez en cada ronda de actuación.
`void addObject(Actor object, int x, int y)`	Añade un Actor al mundo.
`GreenfootImage getBackground()`	Devuelve el fondo del mundo.
`int getCellSize()`	Devuelve el tamaño de una celda (en píxeles).
`Color getColorAt(int x, int y)`	Devuelve el color del centro de la celda.
`int getHeight()`	Devuelve el ancho del mundo (en número de celdas).
`List getObjects(Class cls)`	Devuelve todos los objetos del mundo.
`List getObjectsAt(int x, int y, Class cls)`	Devuelve todos los objetos en una celda dada.
`int getWidth()`	Devuelve el ancho del mundo (en número de celdas).
`int numberOfObjects()`	Devuelve el número de actores que hay en el mundo.
`void removeObject(Actor object)`	Elimina un objeto del mundo.
`void removeObjects(Collection objects)`	Elimina una lista de objetos del mundo.
`void repaint()`	Repinta el mundo.
`void setActOrder(Class... classes)`	Fija el orden de actuación de objetos en el mundo.
`void setBackground(GreenfootImage image)`	Fija una imagen de fondo en el mundo.
`void setBackground(String filename)`	Fija una imagen de fondo del mundo a partir de un fichero.
`void setPaintOrder(Class... classes)`	Fija el orden en que se pintan los objetos en el mundo.
`void started()`	Llamado por el sistema Greenfoot cuando comienza la ejecución.
`void stopped()`	Llamado por el sistema Greenfoot cuando para la ejecución.

Class Actor

`Actor()`	Construye un Actor.
`void act()`	El método act es llamado por el framework Greenfoot para dar a los objetos la oportunidad de hacer alguna acción.
`protected void addedToWorld(World world)`	Este método es llamado por el sistema Greenfoot cuando el objeto ha sido añadido al mundo.
`GreenfootImage getImage()`	Devuelve la imagen usada para representar este Actor.
`protected List getIntersectingObjects(Class cls)`	Devuelve todos los objetos que intersecan con este objeto.
`protected List getNeighbours(int distance, boolean diagonal, Class cls)`	Devuelve los vecinos de este objeto a una distancia dada.
`protected List getObjectsAtOffset(int dx, int dy, Class cls)`	Devuelve todos los objetos que intersecan en la ubicación dada (relativo a la ubicación del objeto).
`protected List getObjectsInRange(int r, Class cls)`	Devuelve todos los objetos en un rango «r» alrededor de este objeto.
`protected Actor getOneIntersectingObject (Class cls)`	Devuelve un objeto que interseca con este objeto.
`protected Actor getOneObjectAtOffset (int dx, int dy, Class cls)`	Devuelve un objeto que está ubicado en la celda especificada (relativa a la ubicación de este objeto).
`int getRotation()`	Devuelve la rotación actual del objeto.
`World getWorld()`	Devuelve el mundo en que el objeto vive.
`int getX()`	Devuelve la coordenada x de la ubicación actual del objeto.
`int getY()`	Devuelve la coordenada y de la ubicación actual del objeto.
`protected boolean intersects(Actor other)`	Comprueba si este objeto interseca con otro objeto dado.
`void setImage(GreenfootImage image)`	Fija la imagen para este objeto a la imagen dada.
`void setImage(String filename)`	Fija una imagen para este objeto a partir de un fichero.
`void setLocation(int x, int y)`	Asigna una nueva ubicación para este objeto.
`void setRotation(int rotation)`	Fija la rotación del objeto.

Class Greenfoot

`Greenfoot()`	Constructor.
`static void delay(int time)`	Difiere la ejecución un número de pasos de tiempo. El tamaño de un paso de tiempo se define con la barra deslizante de la velocidad.
`static String getKey()`	Obtiene la última tecla pulsada desde la última vez que se llamó a este método.
`static MouseInfo getMouseInfo()`	Devuelve un objeto MouseInfo con información del estado del ratón.
`static int getRandomNumber(int limit)`	Devuelve un número aleatorio entre 0 (inclusive) y el límite (exclusive).
`static boolean isKeyDown(String keyName)`	Comprueba si una tecla dada se está pulsando actualmente.

(*continúa*)

Class Greenfoot (*continuación*)	
`static boolean` **`mouseClicked`**`(Object obj)`	*True* si el ratón se ha pinchado en el objeto dado.
`static boolean` **`mouseDragEnded`**`(Object obj)`	*True* si se ha terminado de arrastrar el ratón.
`static boolean` **`mouseDragged`**`(Object obj)`	*True* si se ha arrastrado el ratón sobre el objeto dado.
`static boolean` **`mouseMoved`**`(Object obj)`	*True* si se ha movido el ratón sobre el objeto dado.
`static boolean` **`mousePressed`**`(Object obj)`	*True* si el ratón está pulsado sobre el objeto dado.
`static void` **`playSound`**`(String soundFile)`	Reproduce un sonido de un fichero.
`static void` **`setSpeed`**`(int speed)`	Fija la velocidad de ejecución de la simulación.
`static void` **`start`**`()`	Ejecuta (o reanuda) la simulación.
`static void` **`stop`**`()`	Para la simulación.

Class MouseInfo	
`Actor` **`getActor`**`()`	Devuelve el actor (si hay) con que está relacionada la conducta actual del ratón.
`int` **`getButton`**`()`	Número de botón pinchado o pulsado.
`int` **`getClickCount`**`()`	Número de clicks de este evento de ratón.
`int` **`getX`**`()`	Posición x actual del cursor del ratón.
`int` **`getY`**`()`	Posición y actual del cursor del ratón.
`String` **`toString`**`()`	Devuelve una representación textual de la información de este evento de ratón.

Class GreenfootImage	
`GreenfootImage``(GreenfootImage image)`	Crea un objeto GreenfootImage a partir de otro objeto GreenfootImage.
`GreenfootImage``(int width, int height)`	Crea una imagen vacía (transparente) con el tamaño especificado.
`GreenfootImage``(String filename)`	Crea una imagen a partir de un fichero.
`void` **`clear`**`()`	Borra la imagen.
`void` **`drawImage`**`(GreenfootImage image, int x, int y)`	Dibuja la imagen dada sobre esta imagen.
`void` **`drawLine`**`(int x1, int y1, int x2, int y2)`	Dibuja una línea, usando el color de dibujo actual, entre los puntos (x1, y1) y (x2, y2).
`void` **`drawOval`**`(int x, int y, int width, int height)`	Dibuja un óvalo dentro del rectángulo especificado con el color de dibujo actual.
`void` **`drawPolygon`**`(int[] xPoints, int[] yPoints, int nPoints)`	Dibuja un polígono cerrado definido por arrays de coordenadas x e y.
`void` **`drawRect`**`(int x, int y, int width, int height)`	Dibuja el contorno del rectángulo especificado.
`void` **`drawstring`**`(String string, int x, int y)`	Dibuja el texto dado en el string especificado, usando la fuente y color actual.

(*continúa*)

Class GreenfootImage (*continuación*)	
`void` **`fill`**`()`	Rellena la imagen por entero con el color actual de dibujo.
`void` **`fillOval`**`(int x, int y, int width, int height)`	Rellena un óvalo limitado por el rectángulo especificado con el color de dibujo actual.
`void` **`fillPolygon`**`(int[] xPoints, int[] yPoints, int nPoints)`	Rellena un polígono cerrado definido por arrays de las coordenadas x e y.
`void` **`fillRect`**`(int x, int y, int width, int height)`	Rellena el rectángulo especificado.
`BufferedImage` **`getAwtImage`**`()`	Devuelve el objeto BufferedImage que respalda este objeto GreenfootImage.
`Color` **`getColor`**`()`	Devuelve el dibujo actual.
`Color` **`getColorAt`**`(int x, int y)`	Devuelve el color en el píxel dado.
`Font` **`getFont`**`()`	Devuelve la fuente actual.
`int` **`getHeight`**`()`	Devuelve el ancho de la imagen.
`int` **`getTransparency`**`()`	Devuelve la transparencia de la imagen (rango 0-255).
`int` **`getWidth`**`()`	Devuelve el ancho de la imagen.
`void` **`mirrorHorizontally`**`()`	Refleja la imagen horizontalmente (según el eje x).
`void` **`mirrorVertically`**`()`	Refleja la imagen verticalmente (según el eje y).
`void` **`rotate`**`(int degrees)`	Rota esta imagen por el centro.
`void` **`scale`**`(int width, int height)`	Escala esta imagen a un nuevo tamaño.
`void` **`setColor`**`(Color color)`	Fija el color de dibujo actual.
`void` **`setColorAt`**`(int x, int y, Color color)`	Fija el color en el píxel dado al color dado.
`void` **`setFont`**`(Font f)`	Fija la fuente actual.
`void` **`setTransparency`**`(int t)`	Fija la transparencia de la imagen (rango 0-255).
`String` **`toString`**`()`	Devuelve una representación textual de la imagen.

APÉNDICE

C

Detección de colisiones

En este libro, se usan varios métodos de detección de colisiones en situaciones diferentes. A continuación, se incluye un resumen de los métodos de detección de colisiones de los actores de Greenfoot, y una breve explicación de su propósito y cuándo usarlos.

C.1 Resumen de métodos

Los métodos de detección de colisiones de Greenfoot se encuentran en la clase Actor. Hay los siguientes seis métodos relevantes:

`List getIntersectingObjects(Class cls)`
Devuelve todos los objetos que intersecan con este objeto.

`Actor getOneIntersectingObject(Class cls)`
Devuelve un objeto que interseca con este objeto.

`List getObjectsAtOffset(int dx, int dy, Class cls)`
Devuelve todos los objetos que intersecan en la ubicación dada (relativa la ubicación de este objeto).

`Actor getOneObjectAtOffset(int dx, int dy, Class cls)`
Devuelve un objeto que está ubicado en la celda especificada (relativa a la ubicación de este objeto).

`List getNeighbours(int distance, boolean diagonal, Class cls)`
Devuelve los vecinos de este objeto dentro de una distancia dada.

`List getObjectsInRange(int r, Class cls)`
Devuelve todos los objetos dentro de un rango «r» alrededor de este objeto.

C.2 Métodos de conveniencia

Dos métodos, `getIntersectingObjects` y `getObjectsAtOffset`, tienen asociados métodos de conveniencia que comienzan con `getOne`. . .

Estos métodos de conveniencia funcionan de forma muy similar a los que están basados, pero devuelven un único actor en vez de una lista de actores. En los casos en que podría haber otros actores (p. ej., varios actores que intersequen con el nuestro a la vez), la variante que devuelve una lista, devuelve todos los actores relevantes. La variante que devuelve un único valor, selecciona uno aleatoriamente entre todos los actores que intersecan y lo devuelve.

El propósito de estos métodos de conveniencia es simplemente simplificar el código: a menudo, sólo estamos interesados en un único actor que interseca. En estos casos, los métodos de conveniencia nos permiten gestionar el actor sin tener que usar la lista.

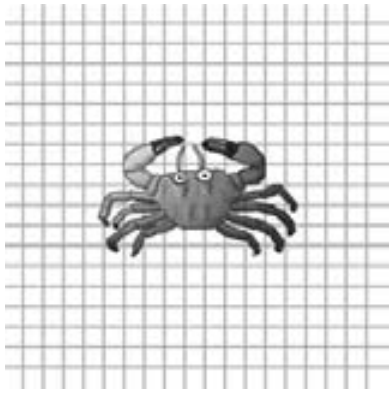
C.3 Resolución baja frente a resolución alta

Como hemos visto a lo largo de este libro, la resolución (tamaño de celda) de los mundos de Greenfoot puede variar. Esto es relevante para la detección de colisiones, ya que usaremos a menudo métodos diferentes dependiendo de la resolución.

Distinguimos dos casos: mundos de baja resolución, donde la imagen del actor está completamente contenida en una única celda (Figura C.1 a) y mundos de alta resolución, donde la imagen de un actor ocupa más de una celda (Figura C.1 b).

Figura C.1
Ejemplos de baja y alta resolución en los mundos de Greenfoot

a) un mundo de baja resolución



b) un mundo de alta resolución

C.4 Intersección de objetos

Métodos:

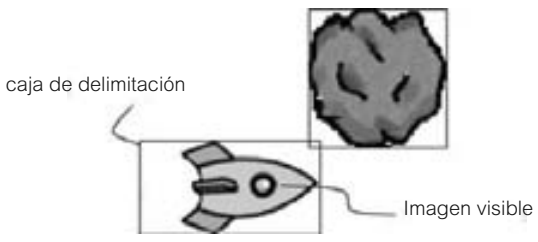
`List getIntersectingObjects(Class cls)`
Devuelve todos los objetos que intersecan con este objeto.

`Actor getOneIntersectingObject(Class cls)`
Devuelve un objeto que interseca con este objeto.

Estos métodos devuelven otros actores cuya imagen interseca con la imagen del actor llamante. Las imágenes intersecan cuando cualquier parte de una imagen toca cualquier parte de otra imagen. Estos métodos son principalmente útiles en escenarios de alta resolución.

La intersección se calcula usando cajas delimitadoras, de forma que solapes de las partes transparentes de las imágenes también se tratan como intersección (Figura C.2).

Figura C.2
Intersección de actores usando cajas de delimitación



Estos métodos se usan a menudo para comprobar si un actor se ha topado con otra clase de actor. La imprecisión resultante de usar cajas de delimitación (en vez de la parte visible de la imagen) a menudo se puede despreciar.

Los parámetros se pueden usar como un filtro. Si una clase se especifica como un parámetro a estos métodos, sólo los objetos de esa clase se consideran, y el resto de objetos se ignoran. Si se usa `null`, se devuelve cualquier objeto que interseque.

C.5 Objetos en un desplazamiento hasta una posición

Métodos:

`List getObjectsAtOffset(int dx, int dy, Class cls)`
Devuelve todos los objetos que intersecan con la ubicación dada (relativa a la ubicación de este objeto).

`Actor getOneObjectAtOffset(int dx, int  dy, Class cls)`
Devuelve un objeto que está ubicado en la celda especificada (relativa a la ubicación de este objeto).

Estos métodos se pueden usar para comprobar los objetos que están en un desplazamiento dado desde la posición actual de un actor. Son útiles para ambos tipos de escenarios, de alta y baja resolución.

Los parámetros *dx* y *dy* especifican el desplazamiento en número de celdas. La figura C.3 ilustra la ubicación en el desplazamiento (2,0) desde el wombátido (2 celdas de desplazamiento en la coordenada *x* y 0 celdas en la coordenada *y*).

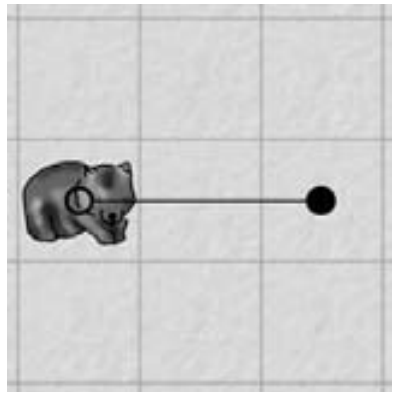


Figura C.3
Comprobando un desplazamiento dado desde una ubicación (ejemplo, aquí desplazamiento 2,0)

Se considera que otro actor está en este desplazamiento si cualquier parte de la imagen de ese actor interseca con el punto central de la celda especificada. El parámetro `cls` proporciona de nuevo la opción de filtrar los objetos que se consideran (mira arriba).

Estos métodos se usan a menudo para comprobar un área en frente de un actor (para comprobar si se puede mover hacia delante) o debajo de un actor (para comprobar si está de pie sobre algo).

C.6 Vecinos

Método:

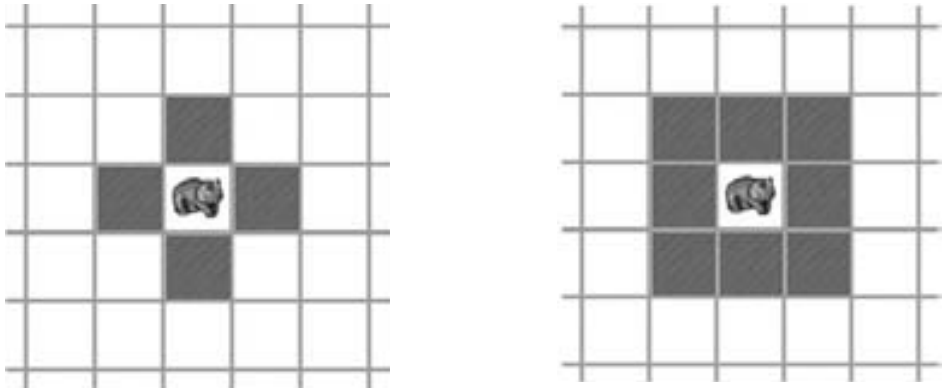
```
List getNeighbours(int distance, boolean diagonal, Class cls)
```
Devuelve los vecinos de este objeto dentro de una distancia dada.

Este método se usa para recuperar los objetos de las celdas que rodean al actor actual. Se usa principalmente en escenarios de baja resolución.

Observa que la ortografía del nombre del método es `getNeighbours` (con ortografía británica) —Greenfoot no es un sistema americano.

Los parámetros especifican la distancia desde el actor llamante que deberían considerarse y si deben incluirse o no las celdas de la diagonal. La Figura C.4 ilustra las celdas vecinas a distancia 1, incluyendo y sin incluir las diagonales.

Figura C.4
Ejemplo del método getNeighbours



a) vecinos con diagonal = false

b) vecinos con diagonal = true

Se define una distancia de *N* como todas las celdas a las que se puede llegar en *N* pasos desde la posición actual del actor. El parámetro diagonal determina si los pasos en diagonal se permiten en este algoritmo.

Como en los métodos previos, el parámetro `cls` proporciona la opción de considerar sólo objetos de una clase dada.

C.7 Objetos en un alcance

Método:

```
List getObjectsInRange(int r, Class cls)
```
Devuelve todos los objetos dentro de un rango «r» alrededor de este objeto.

Este método devuelve todos los objetos dentro de un rango dado del actor llamante. Un objeto está en este rango si una celda cuyo punto central está a una distancia *r* o menos del actor llamante (Figura C.5). El rango se mide en celdas.

Este método se usa principalmente en escenarios de alta resolución. Como en los métodos anteriores, se puede aplicar un filtro por clases.

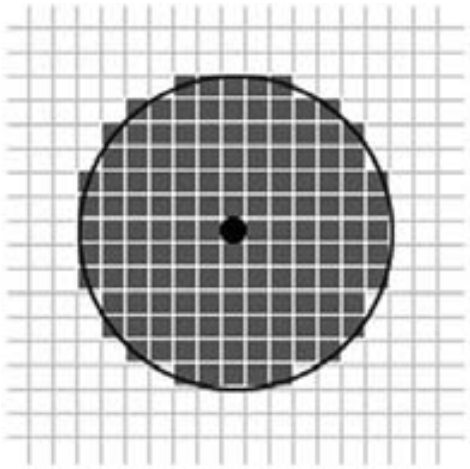


Figura C.5
Las celdas en un rango dado alrededor de una ubicación

Términos en inglés

Inglés	Español
convenience method	método de conveniencia

APÉNDICE

D

Algunos detalles de Java

D.1 Tipos de datos Java

Java tiene dos tipos de datos: tipos primitivos y tipos objeto. Los tipos primitivos se almacenan en las variables directamente, y tienen semántica de valor (los valores se copian cuando se asignan a otra variable). Los tipos objeto se almacenan almacenando referencias al objeto (no el objeto mismo). Cuando se asignan a otra variable, sólo se copia la referencia, y no el objeto.

D.1.1 Tipos primitivos

La tabla siguiente lista todos los tipos primitivos del lenguaje Java:

Nombre tipo	Descripción	Ejemplo literales		
Números enteros				
`byte`	Entero tamaño de byte (8 bit)	`24`	`-2`	
`short`	Entero corto (16 bit)		`137`	`-119`
`int`	Entero (32 bit)	`5409`	`-2003`	
`long`	Entero largo (64 bit)	`423266353L`	`55L`	
Números reales				
`float`	Punto flotante precisión simple	`43.889F`		
`double`	Punto flotante doble precisión	`45.632.4e5`		
Otros tipos				
`char`	Carácter simple (16 bit)	`'m'`	`'?'`	`'\u00F6'`
`boolean`	Valor booleano (true o false)	`true false`		

Notas:

- *Un número sin un punto decimal se interpreta generalmente como un* `int`, *pero se convierte automáticamente a tipos* `byte`, `short`, *o* `long` *cuando se asigna (si el valor encaja). Puedes declarar un literal como* `long` *poniendo una* `L` *después del número.* (`l` —`L` *minúscula —también funciona pero debería evitarse porque se puede confundir fácilmente con un uno.*)
- *Un número con un punto decimal es de tipo* `double`. *Puedes especificar un literal* `float` *poniendo una* `F` *o* `f` *tras el número.*
- *Un carácter se puede escribir como un carácter simple Unicode entre comillas simples o como un valor Unicode de cuatro dígitos, precedido de* `\u`.
- *Los dos literales booleanos son* `true` *y* `false`.

Como las variables de tipos primitivos no se refieren a objetos, no hay métodos asociados a los tipos primitivos. Sin embargo, cuando se usan en un contexto que requiere un tipo de objeto, se emplea la técnica «autoboxing» para convertir un tipo primitivo en su objeto correspondiente.

La siguiente tabla detalla los valores máximos y mínimos de los tipos números.

Tipo	Mínimo	Máximo
`byte`	-128	127 short -32768 32767
`int`	-2147483648	2147483647
`long`	-9223372036854775808	9223372036854775807
	Mínimo positivo	**Máximo positivo**
`float`	1.4e–45	3.4028235e38
`double`	4.9e–324	1.7976931348623157e308

D.1.2 Tipos objeto

Todos los tipos no listados en la sección de *tipos primitivos* son tipos objeto. Aquí se incluyen los tipos clase e interfaz de la librería estándar de Java (tales como `String`) y los tipos definidos por el usuario.

Una variable de un tipo objeto contiene una referencia (o «puntero») a un objeto. Las asignaciones y pasos de parámetros tienen semántica de referencia (esto es, se copia la referencia, no el objeto). Después de asignar una variable a otra, ambas variables se refieren al mismo objeto. Se dice que las dos variables son alias del mismo objeto.

Las clases son plantillas de los objetos, que definen los campos y métodos que cada instancia tiene. Los arrays se comportan como tipos objeto —tienen también semántica de referencia.

D.2 Operadores Java

D.2.1 Expresiones aritméticas

Java tiene un gran número de operadores disponibles tanto para expresiones aritméticas como lógicas. La tabla D.1 muestra todo lo que está clasificado como un operador, incluyendo cosas como la conversión de tipos (casting) y el paso de parámetros. La mayoría de los operadores son operadores binarios (tomando un operando izquierdo y derecho) u operadores unarios (tomando un único operador). Las principales operaciones aritméticas binarias son:

`+` *suma*
`–` *resta*
`*` *multiplicación*
`/` *división*
`%` *módulo o resto-tras-la-división*

Tabla D.1
Los operadores Java con mayor precedencia están arriba

[]	.	++	– –	(*parámetros*)							
++	– –	+	–	!	~						
new	(*cast*)										
*	/	%									
+	–										
<<	>>	>>>									
<	>	>=	<=	instanceof							
==	!=										
&											
^											
|											
&&											
||											
?:											
=	+=	–=	*=	/=	%=	>>=	<<=	>>>=	&=	|=	^=

Los resultados tanto de las operaciones de división como módulo dependen de si sus operandos son valores enteros o punto flotante. Entre dos valores enteros, la división da un resultado entero y descarta el resto, pero entre dos valores punto flotante, el resultado es un valor punto flotante:

```
5 / 3  da un resultado de 1
5.0 / 3  da un resultado de 1.6666666666666667
```

(Observa que sólo uno de los operandos tiene que ser de tipo punto flotante para producir un resultado punto flotante.)

Cuando más de un operador aparece en una expresión, entonces se aplican las *reglas de precedencia* para determinar el orden de aplicación. En la Tabla D.1 los operadores con mayor precedencia aparecen arriba, de forma que podemos ver que la multiplicación, la división, y el módulo tienen precedencia sobre la suma y la resta, por ejemplo. Esto significa que los dos ejemplos siguientes dan 100 como resultado:

```
51 * 3 – 53
154 – 2 * 27
```

Los operadores binarios con el mismo nivel de precedencia se evalúan de izquierda a derecha, y los operadores unarios con el mismo nivel de precedencia se evalúan de derecha a izquierda.

Cuando es necesario alterar el orden normal de evaluación, se usan paréntesis. Por tanto, los dos ejemplos siguientes dan el resultado 100:

```
(205 – 5) / 2
2  * (47 + 3)
```

Los principales operadores unarios son –, !, ++, – –, [], y new. Observarás que ++ y – – aparecen ambos en las filas superiores de la Tabla D.1. Los que están en la fila superior toman un único operando a su izquierda, mientras que los que están en la segunda fila toman un único operando a su derecha.

D.2.2 Expresiones booleanas

En las expresiones booleanas, los operadores se pueden usar para combinar operandos para producir un valor que sea cierto o falso. Estas expresiones se encuentran normalmente en la expresión que evalúa las sentencias *if-else* y los bucles.

Los operadores relacionales se usan normalmente para combinar un par de operadores aritméticos, aunque la comprobación de igualdad o desigualdad se hace también con referencias a objetos. Los operadores de comparación de Java son:

```
==  igual-a          !=  no-igual-a
<   menor-que        <=  menor-o-igual-a
>   mayor-que        >=  mayor-o-igual-a
```

Los operadores binarios lógicos combinan dos expresiones booleanas para producir otro valor booleano. Los operadores son:

```
&&  y (and)
||  o (o)
^   o exclusivo (exclusive-or)
```

Además,

```
!  no (not)
```

toma una expresión booleana y la cambia de `true` a `false`, y viceversa.

Tanto `&&` como `||` son un poco especiales en la forma que se usan. Si el operando de la izquierda de `&&` es falso entonces el valor del operando derecho es irrelevante y no será evaluado. De forma similar, si el operando izquierdo de `||` es cierto, entonces el operando derecho no se evalúa. Por esto, se conocen como operadores cortocircuito.

D.3 Estructuras de control en Java

Las estructuras de control afectan al orden en que se ejecutan las sentencias dentro del cuerpo de un método o un constructor. Hay dos tipos principales de categorías: *sentencias de selección* y *bucles.* Una sentencia de selección proporciona un punto de decisión para escoger seguir una ruta dentro del cuerpo de un método o de un constructor en vez de otra ruta. Una sentencia *if-else* supone una decisión entre dos bloques de sentencias, mientras que una sentencia *switch* permite seleccionar entre varias alternativas de una única opción.

Los bucles ofrecen la opción de repetir sentencias, bien un número definido o bien un número indefinido de veces. El primero se tipifica en los bucles *for-each* y *for*, mientras que el segundo caso se tipifica en los bucles *while* y *do*.

En la práctica, debe tenerse en cuenta que son bastante habituales las excepciones a las caracterizaciones anteriores. Por ejemplo, una sentencia *if-else* se puede usar para seleccionar entre varias alternativas si la parte *else* contiene sentencias anidadas *if-else*; y un bucle *for* se puede usar para hacer un bucle infinito.

D.3.1 Sentencias de selección

D.3.1.1 *if-else*

La sentencia *if-else* tiene dos formas principales, y ambas están controladas por la evaluación de una expresión booleana:

```
if (expresión)
{
    sentencias
}

if (expresión)
{
    sentencias
}
else
{
    sentencias
}
```

En la primera forma, el valor de la expresión booleana se usa para decidir si se ejecutan las sentencias o no. En la segunda forma, la expresión se usa para escoger entre dos bloques de sentencias alternativos, y sólo se ejecuta uno de ellos.

Ejemplos:

```
if (campo.size() == 0)
{
    System.out.println("El campo está vacío.");
}

if (numero <  0)
{
    reportError();
}
else
{
    procesaNumero(numero);
}
```

Es muy común enlazar las sentencias *if-else* juntas poniendo un segundo *if-else* en la parte *else* del primero. Esto se puede hacer varias veces. Es buena idea acabar siempre con una parte final *else*.

```
if (n < 0)
{
    gestionaNegativo();
}
else if (n == 0)
{
    handleZero();
}
else
{
    handlePositive();
}
```

D.3.1.2 *switch*

La sentencia *switch* evalúa un único valor y según éste ejecuta las sentencias de un número arbitrario de casos. Hay dos patrones posibles:

```
switch (expresión)
{
    case valor: sentencias;
        break;
    case value: sentencias;
        break;
    otros casos omitidos
    default: sentencias;
        break;
}

switch (expresión)
{
    case valor1:
    case valor2:
    case valor3:
        sentencias;
        break;
    case valor4:
    case valor5:
        sentencias;
        break;
        otros casos omitidos
        default:
        sentencias;
        break;
```

Notas:

- *Una sentencia* `switch` *puede tener cualquier número de etiquetas case.*
- *La instrucción* `break` *después de cada caso es necesaria, si no, la ejecución seguiría en las sentencias de la etiqueta siguiente. La segunda forma hace uso de esto. En este caso, todos los valores primeros se ejecutan en la primera sección de sentencias, mientras que los valores cuarto y quinto se ejecutarán en la segunda sección de sentencias.*
- *El caso* `default` *es opcional. Si no se da ningún caso* default, *puede ocurrir que ningún caso se ejecute.*
- *La instrucción* `break` *después de* `default` *(o en el último caso, si no hay* `default`*) no es necesaria, pero se considera de buen estilo.*

Ejemplo:

```
switch(dia)
{
    case 1: diaString = “Lunes”;
        break;
    case 2: diaString = “Martes”;
        break;
    case 3: diaString = “Miércoles”;
        break;
    case 4: diaString = “Jueves”;
        break;
```

```
    case 5: diaString = “Viernes”;
        break;
    case 6: diaString = “Sábado”;
        break;
    case 7: diaString = “Domingo”;
        break;
default: diaString = “día inválido”;
        break;
}

switch(mes)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        numeroDeDias = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        numeroDeDias = 30;
        break;
    case 2:
        if(isAñoBisiesto())
            numeroDeDias = 29;
        else
            numeroDeDias = 28;
        break;
}
```

D.3.2 Bucles

Java tiene tres bucles: *while*, *do-while*, y *for.* El bucle *for* tiene dos formas. Salvo en el bucle *for-each*, la repetición se controla en todos con una expresión booleana.

D.3.2.1 *while*

El bucle *while* ejecuta un bloque de sentencias mientras una expresión dada se evalúe como *true.* La expresión se comprueba antes de cada ejecución del cuerpo del bucle, de forma que el cuerpo puede ejecutarse cero veces (esto es, ninguna vez). Esta capacidad es un rasgo importante del bucle *while*.

```
while (expresión)
{
    sentencias;
}
```

Ejemplos:

```
System.out.print(“Por favor introduce un nombre de fichero: ”);
entrada = readInput();
```

```
while (entrada == null)
{
    System.out.print("Por favor, intenta de nuevo: ");
    entrada= readInput();
}

int indice = 0;
boolean encontrado = false;
while (!encontrado &&  indice < list.size())
{
    if(lista.get(indice).equals(input))
    {
         encontrado = true;
    }
    else
    {
    indice++;
    }
}
```

D.3.2.2 *do-while*

El bucle *do-while* ejecuta un bloque de sentencias mientras una expresión dada sea cierta (*true*). La expresión se comprueba tras cada ejecución del cuerpo del bucle, de forma que el cuerpo se ejecuta siempre al menos una vez. Ésta es una diferencia importante con el bucle *while*.

```
do
{
    sentencias;
} while (expresión);
```

Ejemplo:

```
do
{
    System.out.print("Por favor, introduce un nombre de fichero: ");
    entrada = readInput();
} while (entrada == null);
```

D.3.2.3 *for*

El bucle *for* tiene dos formas diferentes. La primera forma se conoce también como un bucle *for-each,* y se usa exclusivamente para iterar sobre los elementos de una colección. La variable del bucle toma el valor de los elementos sucesivos de la colección en cada iteración del bucle.

```
for (declaración-variable : colección)
{
    sentencias;
}
```

Ejemplo:

```
for (String nota : list)
{
    System.out.println(nota);
}
```

La segunda forma de bucle *for* se ejecuta mientras una *condición* se evalúa a *true.* Antes de que el bucle comience, se ejecuta una sentencia de *inicialización* exactamente una vez. La *condición* se evalúa antes de cada ejecución del cuerpo del bucle (así el bucle se puede ejecutar cero veces, es decir, no se ejecuta). Una sentencia de *incremento* se ejecuta tras cada ejecución del cuerpo del bucle.

```
for (inicialización; condición; incremento)
{
     sentencias;
}
```

Ejemplo:

```
for (int i = 0; i < texto.size(); i++)
{
     System.out.println(texto.get(i));
}
```

Ambos tipos de bucle *for* se usan comúnmente para ejecutar el cuerpo del bucle un número definido de veces —por ejemplo, una vez para cada elemento de una colección. Un bucle *for-each* no se puede usar si la colección se modifica mientras se está iterando por ella.

Índice analítico

D

E

F

G

H

I

L

M

N

O

P

Escrito por el creador de Greenfoot y desarrollador de BlueJ, Michael Kölling, ***Introducción a la Programación con Greenfoot*** usa Greenfoot, un **entorno educativo** de programación ganador de premios, para enseñar **programación** Java estándar.

Greenfoot utiliza simulaciones y juegos para enseñar los principios y conceptos de la programación orientada a objetos de una forma divertida y asequible. Greenfoot facilita la transición a la programación en Java, haciendo que incluso temas avanzados sean fáciles de enseñar.

Introducción a la Programación con Greenfoot cubre:

- Fundamentos de programación Java estándar.
- Enfoque objetos-primero.
- Aprendizaje práctico dirigido por proyectos.
- Oportunidades para que los estudiantes desarrollen rápidamente animaciones, juegos y simulaciones.

Para unirse a la comunidad en línea Greenfoot y obtener los recursos compartidos para profesores y estudiantes, visite **www.greenfoot.org/book/**.

Otro libro de interés

Programación orientada a objetos con Java
David J. Barnes
Michael Kölling
PEARSON PRENTICE HALL
ISBN 978-84-832-2350-5

Prentice Hall
es un sello editorial de

PEARSON

www.pearsoneducacion.com

ISBN 978-84-8322-766-4