CHAPTER

3

Improving the Crab—more sophisticated programming

topics: random behavior, keyboard control, sound

concepts: dot notation, random numbers, defining methods, comments

In the previous chapter, we looked at the basics of starting to program our first game. There were many new things that we had to look at. Now, we will add more interesting behavior. Adding code will get a little easier from now on, since we have seen many of the fundamental concepts.

The first thing we shall look at is adding some random behavior.

3.1 Adding random behavior

In our current implementation, the crab can walk across the screen, and it can turn at the edge of our world. But when it walks, it always walks exactly straight. That is what we want to change now. Crabs don't always go in an exact straight line, so let us add a little random behavior: The crab should go roughly straight, but every now and then it should turn a little off course.

We can achieve this in Greenfoot by using random numbers. The Greenfoot environment itself has a method to give us a random number. This method, called `getRandomNumber`, expects a parameter that specifies the limit of the number. It will then return a random number between 0 (zero) and the limit. For example,

Concept:

When a method we wish to call is not in our own class or inherited, we need to specify the class or object that has the method before the method name, followed by a dot. This is called **dot notation**.

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

will give us a random number between 0 and 20. The limit—20—is excluded, so the number is actually in the range 0–19.

The notation used here is called *dot notation*. When we called methods that were defined in our own class or inherited, it was enough to write the method name and parameter list. When the method is defined in another class, we need to specify the class or object that has the method, followed by a period (dot), followed by the method name and parameter. Since the `getRandomNumber` method is not in the `Crab` or `Animal` class, but in a class called `Greenfoot`, we have to write "`Greenfoot`." in front of the method call.

Concept:

Methods that belong to classes (as opposed to objects) are marked with the keyword **static** in their signature. They are also called **class methods**.

Note: Static methods

Methods may belong to objects or classes. When methods belong to a class, we write

```
class-name.method-name (parameters);
```

to call the method. When a method belongs to an object, we write

```
object.method-name (parameters);
```

to call it.

Both kinds of methods are defined in a class. The method signature tells us whether a given method belongs to objects of that class, or to the class itself.

Methods that belong to the class itself are marked with the keyword **static** at the beginning of the method signature. For example, the signature of Greenfoot's `getRandomNumber` method is

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

This tells us that we must write the name of the class itself (Greenfoot) before the dot in the method call.

We will encounter calls to methods that belong to other objects in a later chapter.

Let us say we want to program our crab so that there is a 10 percent chance at every step that the crab turns a little bit off course. We can do the main part of this with an if-statement:

```
if ( something-is-true )
{
    turn(5);
}
```

Now we have to find an expression to put in place of *something-is-true* that returns true in exactly 10 percent of the cases.

We can do this using a random number (using the `Greenfoot.getRandomNumber` method) and a less-than operator. The less-than operator compares two numbers and returns true if the first is less than the second. "Less than" is written using the symbol "<". For example:

```
2 < 33
```

is true, while

```
162 < 42
```

is false.

Exercise 3.1 Before reading on, try to write down, on paper, an expression using the `getRandomNumber` method and the less-than operator that, when executed, is true exactly 10 percent of the time.

Exercise 3.2 Write down another expression that is true 7 percent of the time.

Note

Java has a number of operators to compare two values. They are:

<	less than	>=	greater than or equal
>	greater than	==	equal
<=	less than or equal	!=	not equal

If we want to express the chance in percent, it is easiest to deal with random numbers out of 100. An expression that is true 10 percent of the time, for example, could be

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

Since the call to `Greenfoot.getRandomNumber(100)` gives us a new random number between 0 and 99 every time we call it, and since these numbers are evenly distributed, they will be below 10 in 10 percent of all cases.

We can now use this to make our crab turn a little in 10 percent of its steps (Code 3.1).

Code 3.1 Random course changes—first try

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

/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Animal
{
    public void act()
    {
        if ( atWorldEdge() )
        {
            turn(17);
        }

        if ( Greenfoot.getRandomNumber(100) < 10 )
        {
            turn(5);
        }

        move();
    }
}
```

Exercise 3.3 Try out the random course changes shown above in your own version. Experiment with different probabilities for turning.

This is a pretty good start, but it is not quite nice yet. First of all, if the crab turns, it always turns the same amount (5 degrees), and secondly, it always turns right, never left. What we would really like

to see is that the crab turns a small, but random amount to either its left or its right. (We will discuss this now. If you feel confident enough, try to implement this on your own first before reading on.)

The simple trick to the first problem—always turning the same amount, in our case 5 degrees—is to replace the fixed number 5 in our code with another random number, like this:

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

In this example, the crab still turns in 10 percent of its steps. And when it turns, it will turn a random amount, between 0 and 44 degrees.

Exercise 3.4 Try out the code shown above. What do you observe? Does the crab turn different amounts when it turns?

Exercise 3.5 We still have the problem that the crab turns only right. That's not normal behavior for a crab, so let's fix this. Modify your code so that the crab turns either right or left by up to 45 degrees each time it turns.

Exercise 3.6 Try running your scenario with multiple crabs in the world. Do they all turn at the same time, or independently? Why?

The project *little-crab-2* (included in the book scenarios) shows an implementation of what we have done so far, including the last exercises.

3.2 Adding worms

Let us make our world a little more interesting by adding another kind of animal.

Crabs like to eat worms. (Well, that is not true for all kinds of crab in the real world, but there are some that do. Let's just say our crab is one of those that like to eat worms.) So let us now add a class for worms.

We can add new actor classes to a Greenfoot scenario by selecting *New subclass* from one of the existing actor classes (Figure 3.1). In this case, our new class `Worm` is a specific kind of animal, so it should be a subclass of class `Animal`. (Remember, being a subclass is an *is-a* relationship: A worm *is an* animal.)

When we are creating a new subclass, we are prompted to enter a name for the class and to select an image (Figure 3.2).

In our case, we name the class "Worm". By convention, class names in Java should always start with a capital letter. They should also describe what kind of object they represent, so "Worm" is the obvious name for our purpose.

Then, we should assign an image to the class. There are some images associated with the scenario, and a whole library of generic images to choose from. In this case, we have prepared a worm image and made it available in the scenario images, so we can just select the image named *worm.png*.

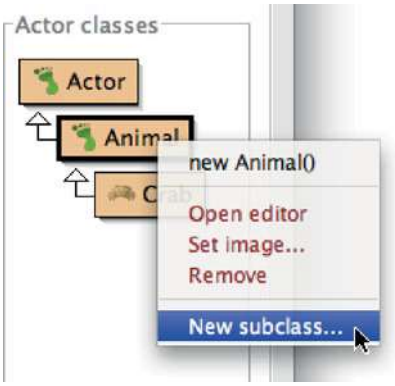

Figure 3.1
Creating new subclasses

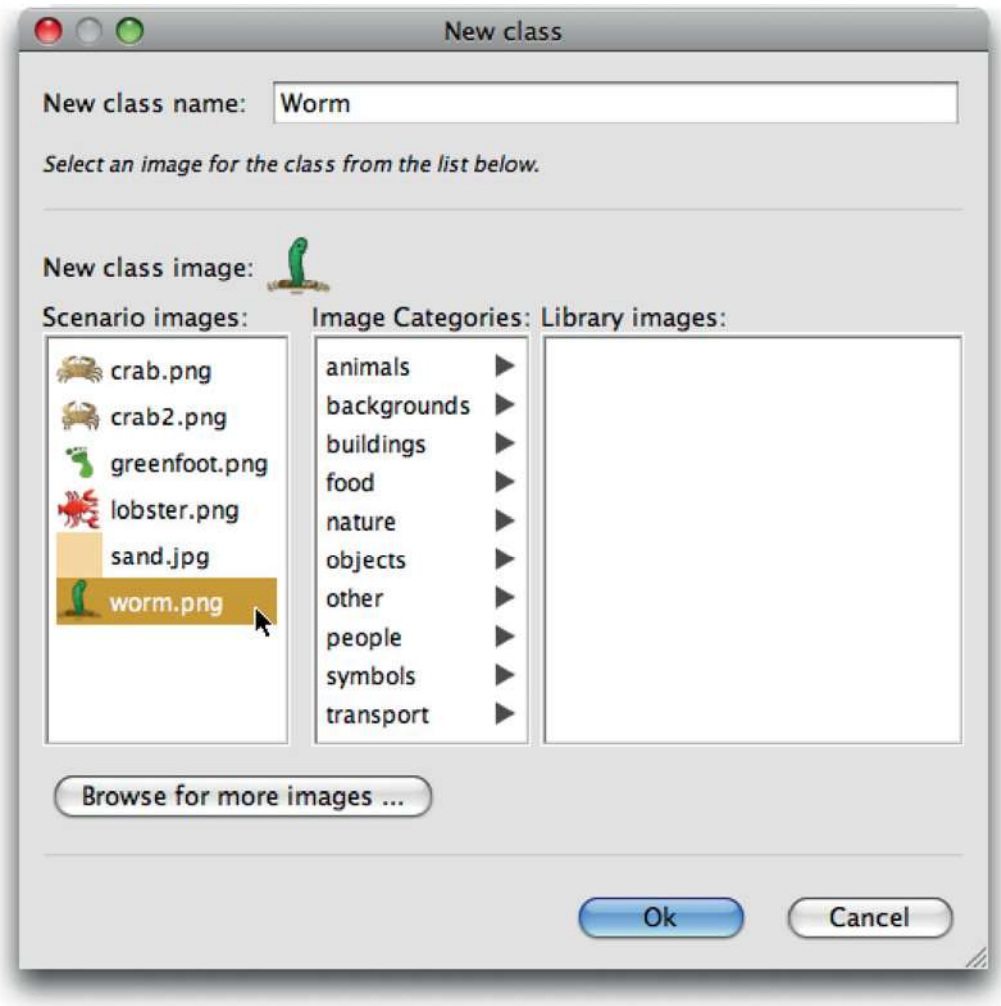

Figure 3.2
Creating a new class

Once done, we can click *Ok*. The class is now added to our scenario, and we can compile and then add worms to our world.

Exercise 3.7 Add some worms to your world. Also add some crabs. Run the scenario. What do you observe? What do the worms do? What happens when a crab meets a worm?

We now know how to add new classes to our scenario. The next task is to make these classes interact: When a crab finds a worm, it should eat it.

3.3 Eating worms

We now want to add new behavior to the crab: When the crab runs into a worm, it eats it. Again, we first check what methods we have already inherited from the `Animal` class. When we open the editor for class `Animal` again, and switch to the *Documentation* view, we can see the following two methods:

```
boolean canSee (java.lang.Class clss)
```
Return true if we can see an object of class 'clss' right where we are.

```
void eat (java.lang.Class clss)
```
Try to eat an object of class 'clss'.

Using these methods, we can implement this behavior. The first method checks whether the crab can see a worm. (It can see it only when it runs right into it—our animals are very short-sighted.) This method returns a `boolean`—*true* or *false*, so we can use it in an if-statement.

The second method eats a worm. Both methods expect a parameter of type `java.lang.Class`. This means that we are expected to specify one of our classes from our scenario. Here is some sample code:

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

In this case, we specify `Worm.class` as the parameter to both method calls (the `canSee` method and the `eat` method). This declares which kind of object we are looking for, and which kind of object we want to eat. Our complete `act` method at this stage is shown in Code 3.2.

Try this out. Place a number of worms into the world (remember: shift-clicking into the world is a shortcut for quickly placing several actors), place a few crabs, run the scenario, and see what happens.

Code 3.2
First version of eating a worm

```
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);
    }
}
```

Advanced note: Packages

(The notes labeled "Advanced note" are inserted for deeper information for those readers really interested in the details. They are not crucial to understand at this stage, and could safely be skipped.)

In the definition of the `canSee` and `eat` methods, we have seen a parameter type with the name `java.lang.Class`. What is going on here?

Many types are defined by classes. Many of those classes are in the standard Java class library. You can see the documentation of the Java class library by choosing *Java Library Documentation* from Greenfoot's *Help* menu.

The Java class library contains thousands of classes. To make these a little easier to work with, they have been grouped into *packages* (logically related groups of classes). When a class name includes dots, such as `java.lang.Class`, only the last part is the name of the class itself, and the former parts form the name of the package. So here we are looking at the class named "Class" from the package "java.lang".

Try to find that class in the Java library documentation.

3.4 Creating new methods

In the previous few sections, we have added new behavior to the crab—turning at the edge of the world, occasional random turns, and eating worms. If we continue to do this in the way we have done so far, the `act` method will become longer and longer, and eventually really hard to understand. We can improve this by chopping it up into smaller pieces.

Concept:

A **method definition** defines a new action for objects of this class. The action is not immediately executed, but the method can be called with a method call later to execute it.

We can create our own additional methods in the `Crab` class for our own purposes. For example, instead of just writing some code that looks for a worm and eats it into the `act` method, we can add a new method for this purpose. To do this, we first have to decide on a name for this method. Let us say we call it `lookForWorm`. We can then create a new method by adding the following code:

```
/**
 * Check whether we have stumbled upon a worm.
 * If we have, eat it. If not, do nothing.
 */
public void lookForWorm()
{
    if ( canSee(Worm.class) )
    {
        eat(Worm.class);
    }
}
```

Concept:

Comments are written into the source code as explanations for human readers. They are ignored by the computer.

The first four lines are a *comment*. A comment is ignored by the computer—it is written for human readers. We use a comment to explain to other human readers what the purpose of this method is.

When we define this method, the code does not immediately get executed. In fact, by just defining this method, it does not get executed at all. We are just defining a new possible action ("looking for a worm") that can be carried out later. It will only be carried out when this method is called. We can add a call to this method inside the act method:

```
lookForWorm();
```

Note that the call has the parentheses for the (empty) parameter list. The complete source code after this restructuring is shown in Code 3.3.

Code 3.3
Splitting code into separate methods

```
public void act()
{
    if ( atWorldEdge() )
    {
        turn(17);
    }

    if ( Greenfoot.getRandomNumber(100) < 10 )
    {
        turn(5);
    }

    move();
    lookForWorm();
}
```

Code 3.3 continued
Splitting code into separate methods

```
/**
 * Check whether we have stumbled upon a worm.
 * If we have, eat it. If not, do nothing.
 */
public void lookForWorm()
{
    if ( canSee(Worm.class) )
    {
        eat(Worm.class);
    }
}
```

Note that this code change does not change the behavior of our crab at all. It just makes the code easier to read in the long run. As we add more code to the class, methods tend to become longer and longer. Longer methods are harder to understand. By separating our code into a number of shorter methods, we make the code easier to read.

Exercise 3.8 Create another new method named `randomTurn` (this method has no parameters and returns nothing). Select the code that does the random turning, and move it from the `act` method to the `randomTurn` method. Then call this new `randomTurn` method from your `act` method. Make sure to write a comment for this method.

Exercise 3.9 Create yet another method named `turnAtEdge` (it also has no parameters and returns nothing). Move the code that checks whether we are at the edge of the world (and does the turn if we are) into the `turnAtEdge` method. Call the `turnAtEdge` method from your `act` method. Your `act` method should now look like the version shown in Code 3.4.

Code 3.4
The new act method after creating methods for the subtasks

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

By convention, method names in Java always start with a lowercase letter. Method names cannot contain spaces (or many other punctuation characters). If the method name logically consists of multiple words, we use capitals in the middle of the method name to mark the start of each word.

3.5 Adding a Lobster

We are now at a stage where we have a crab that walks more or less randomly through our world, and eats worms if it happens to run into them.

To make it a little more interesting, let us add another creature: a lobster.

Lobsters, in our scenario, like to chase crabs.

Exercise 3.10 Add a new class to your scenario. The class should be a subclass of `Animal`, called `Lobster` (with a capital 'L'), and it should use the prepared image *lobster.png*.

Exercise 3.11 What do you expect lobsters to do when you place them into the world as they are? Compile your scenario and try it out.

We now want to program our new lobsters to eat crabs. This is quite easy to do, since the behavior is very similar to the behavior of crabs. The only difference is that lobsters look for crabs, while crabs look for worms.

Exercise 3.12 Copy the complete `act` method from the `Crab` class into the `Lobster` class. Also copy the complete `lookForWorm`, `turnAtEdge`, and `randomTurn` methods.

Exercise 3.13 Change the `Lobster` code so that it looks for crabs, rather than worms. You can do that by changing every occurrence of "Worm" in the source code to "Crab". For instance, where `Worm.class` is mentioned, change it to `Crab.class`. Also change the name `lookForWorm` to `lookForCrab`. Make sure to update your comments.

Exercise 3.14 Place a crab, three lobsters, and many worms into the world. Run the scenario. Does the crab manage to eat all worms before it is caught by a lobster?

You should now have a version of your scenario where both crabs and lobsters walk around randomly, looking for worms and crabs, respectively.

Now, let us turn this program into a game.

3.6 Keyboard control

To get game-like behavior, we need to get a player involved. The player (you!) should be able to control the crab with the keyboard, while the lobsters continue to run randomly by themselves, as they already do.

The Greenfoot environment has a method that lets us check whether a key on the keyboard has been pressed. It is called `isKeyDown`, and, like the `getRandomNumber` method that we encountered in section 3.1, it is a method in the `Greenfoot` class. The method signature is

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

We can see that the method is static (it is a class method) and the return type is `boolean`. This means that the method returns either *true* or *false*, and can be used as a condition in an if-statement.

We also see that the method expects a parameter of type `String`. A `String` is a piece of text (such as a word or a sentence), written in double quotes. The following are examples of `Strings`:

```
"This is a String"
"name"
"A"
```

In this case, the `String` expected is the name of the key that we want to test. Every key on the keyboard has a name. For those keys that produce visible characters, that character is their name, for example, the A-key is called "A". Other keys have names too. For instance, the left cursor key is called "left". Thus, if we want to test whether the left cursor key has been pressed, we can write

```
if (Greenfoot.isKeyDown("left"))
{
   ...// do something
}
```

Note that we need to write "`Greenfoot.`" in front of the call to `isKeyDown`, since this method is defined in the `Greenfoot` class.

Tip:

Greenfoot automatically saves classes and scenarios when their windows are closed. To keep a copy of interim stages of scenarios, use **Save A Copy As** from the Scenario menu.

If, for example, we want our crab to turn left by 4 degrees whenever the left cursor key is being pressed, we can write

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

The idea now is to remove the code from the crab that does the random turning and also the code that turns automatically at the world edge and replace them with the code that lets us control the crab's turn with our keyboard.

Exercise 3.15 Remove the random turning code from the crab.

Exercise 3.16 Remove the code from the crab that does the turn at the edge of the world.

Exercise 3.17 Add code into the crab's `act` method that makes the crab turn left whenever the left cursor key is pressed. Test.

Exercise 3.18 Add another—similar—bit of code to the crab's `act` method that makes the crab turn right whenever the right cursor key is pressed.

Exercise 3.19 If you have not done so in the first place, make sure that the code that checks the key-presses and does the turning is not written directly in the act method, but is instead in a separate method, maybe called `checkKeypress`. This method should be called from the `act` method.

Try solving the tasks by yourself first. If you get stuck, have a look on the next page. Code 3.5 shows the crab's complete act and `checkKeypress` methods after this change. The solution is also available in the book scenarios, as *little-crab-3*. This version includes all the changes we have discussed so far.

Code 3.5
The Crab's "act" method: Controlling the crab with the keyboard

```
/**
 * Act - do whatever the crab wants to do.
 */
public void act()
{
    checkKeypress();
    move();
    lookForWorm();
}

/**
 * Check whether a control key on the keyboard has been pressed.
 * If it has, react accordingly.
 */
public void checkKeypress()
{
    if (Greenfoot.isKeyDown("left"))
    {
        turn(-4);
    }
    if (Greenfoot.isKeyDown("right"))
    {
        turn(4);
    }
}
```

You are now ready to have a first try at playing your game! Place a crab, some worms, and a few lobsters into the world, and see whether you can get all the worms before the lobsters catch you. (Obviously, the more lobsters you place, the harder it gets…)

3.7 Ending the game

One simple improvement we can make is to end execution of the game when the crab is caught by a lobster. Greenfoot has a method to do this—we just need to find out what it is called.

Concept:

The **API Documentation** lists all classes and methods available in Greenfoot. We often need to look up methods here.

To find out what the available methods in Greenfoot are, we can look at the documentation of the Greenfoot classes.

In Greenfoot, choose *Greenfoot Class Documentation* from the *Help* menu. This will show the documentation for all the Greenfoot classes in a Web browser (Figure 3.3).

This documentation is also called the *Greenfoot API* (for application programmers' interface). The API shows all available classes and, for each class, all the available methods. You can see that Greenfoot offers five classes: `Actor`, `Greenfoot`, `GreenfootImage`, `MouseInfo`, and `World`.

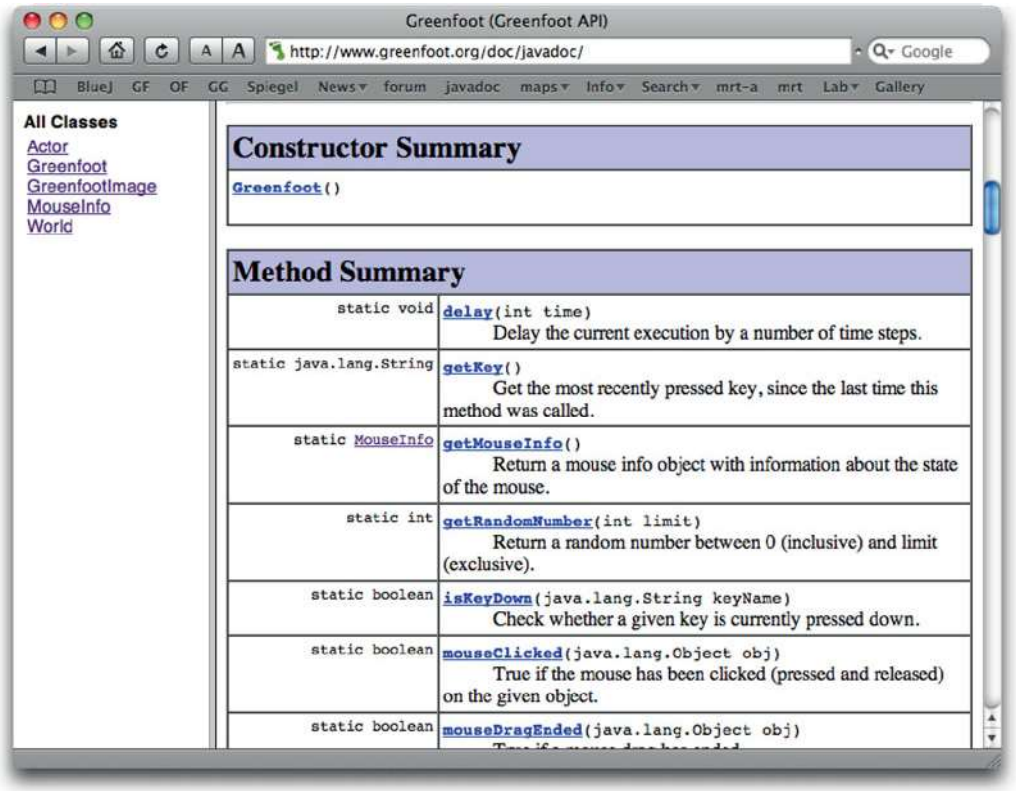

Figure 3.3
The Greenfoot API in a browser window

The method we are looking for is in the `Greenfoot` class.

Exercise 3.20 Open the Greenfoot API in your browser. Select the `Greenfoot` class. In its documentation, find the section titled "Method Summary". In this section, try to find a method that stops the execution of the running scenario. What is this method called?

Exercise 3.21 Does this method expect any parameters? What is its return type?

We can see the documentation of the Greenfoot classes by selecting them in the list on the left. For each class, the main panel in the browser displays a general comment, details of its constructors, and a list of its methods. (Constructors will be discussed in a later chapter.)

If we browse through the list of available methods in the class `Greenfoot`, we can find a method named `stop`. This is the method that we can use to stop execution when the crab gets caught.

We can make use of this method by writing

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

into our source code.

Exercise 3.22 Add code to your own scenario that stops the game when a lobster catches the crab. You will need to decide where this code needs to be added. Find the place in your code that gets executed when a lobster eats a crab, and add this line of code there.

We will use this class documentation frequently in the future to look up details of methods we need to use. We will know some methods by heart after a while, but there are always methods we need to look up.

3.8 Adding sound

Another improvement to our game is the addition of sounds. Again, a method in the `Greenfoot` class helps us with this.

Exercise 3.23 Open the *Greenfoot Class Documentation* (from the *Help* menu), and look at the documentation of class `Greenfoot`. Find the details of the method that can be used to play a sound. What is its name? What parameters does it expect?

By looking through the documentation, we can see that the `Greenfoot` class has a method called `playSound`. It expects the name of a sound file (as String) as a parameter, and returns nothing.

Note

You may like to look at the structure of a Greenfoot scenario in your file system. If you look into the folder containing the book scenarios, you can find a folder for each Greenfoot scenario. For the crab example, there are several different versions (*little-crab*, *little-clab-2*, *little-crab-3*, etc.). Inside each scenario folder are several files for each scenario class, and several other support files. There are also two media folders: *images* holds the scenario images and *sounds* stores the sound files.

You can see the available sounds by looking into this folder, and you can make more sounds available by storing them here.

In our crab scenario, two sound files are already included. They are called *slurp.wav* and *au.wav*.

We can now easily play one of the sounds by using the following method call:

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

Try it out!

Exercise 3.24 Add playing of sounds to your scenario: When a crab eats a worm, play the "slurp.wav" sound. When a lobster eats the crab, play the "au.wav" sound. To do this, you have to find the place in your code where this should happen.

The *little-crab-4* version of this scenario shows the solution to this. It is a version of the project that includes all the functionality we have discussed so far: worms, lobsters, keyboard control, and sound (Figure 3.4).

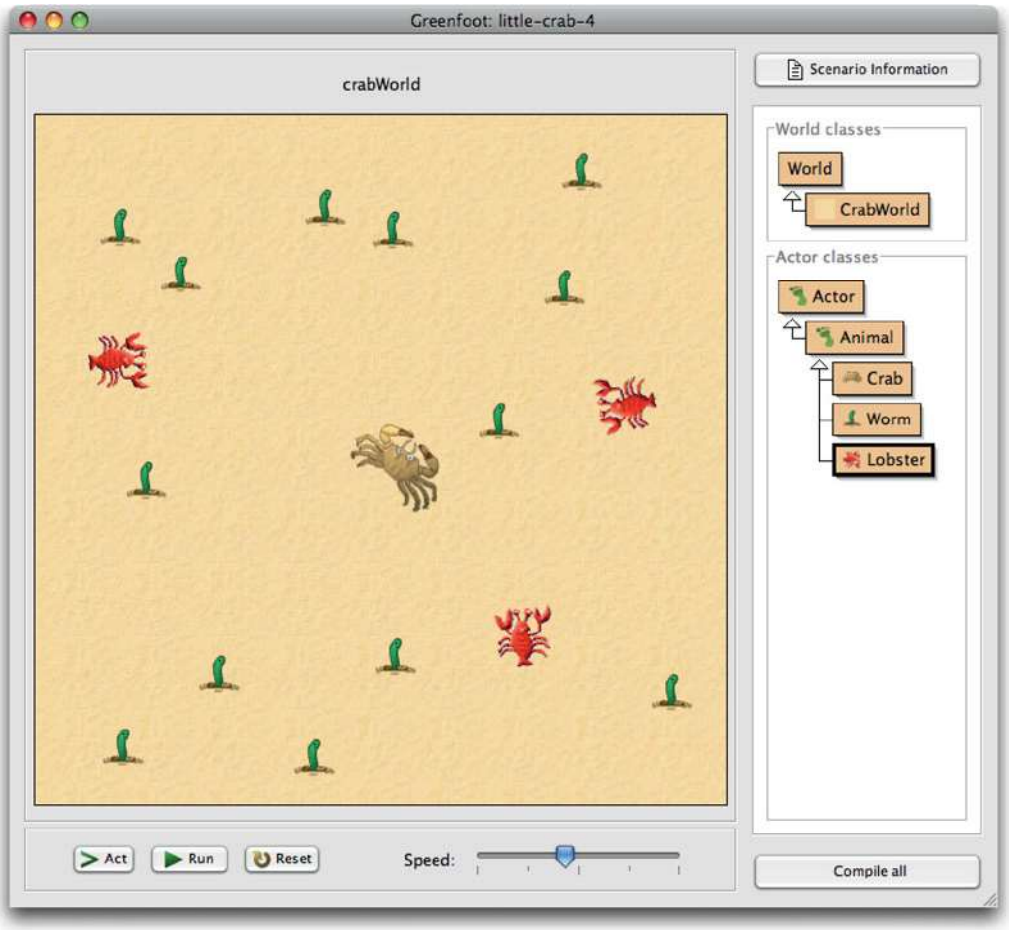

Figure 3.4
The crab game with worms and lobsters

About sound recording

You can also make your own sounds. Both the sounds included are recorded by simply speaking into the computer's microphone. Use one of the many free sound recording programs[1], record your sound, and save (or export) it as a sound file, in either WAV, AIFF, or AU format. Making your own sounds is further discussed in Chapter 8.

Exercise 3.25 If you have a microphone on your computer, make your own sounds to use when the worms or the crab get eaten. Record the sounds with any sound recording program, store them in the scenario's *sounds* folder, and use them in your code.

3.9 Summary of programming techniques

In this chapter we have seen more examples of using an if-statement—this time for turning at random times and reacting to key presses. We have also seen how to call methods from another class, namely the `getRandomNumber`, `isKeyDown`, and `playSound` methods from the `Greenfoot` class. We did this by using dot notation, with the class name in front of the dot.

Altogether, we have now seen examples of calling methods from three different places. We can call methods that are defined in the current class itself (called *local methods*), method that were defined in a superclass (*inherited methods*), and static methods from other classes. The last of these uses dot notation. (There is one additional version of a method call: calling methods on other objects—we will encounter that a little later.)

Another important aspect that we explored was how to read the API documentation of an existing class to find out what methods it has and how to call them.

Concept summary

- When a method we wish to call is not in our own class or inherited, we need to specify the class or object that has the method before the method name, followed by a dot. This is called **dot notation**.
- Methods that belong to classes (as opposed to objects) are marked with the keyword **static** in their signature. They are also called **class methods**.
- A **method definition** defines a new action for objects of this class. The action is not immediately executed, but the method can be called with a method call later to execute it.
- **Comments** are written into the source code as explanations for human readers. They are ignored by the computer.
- The **API Documentation** lists all classes and methods available in Greenfoot. We often need to look up methods here.

[1] Using an Internet search, you should be able to find several free programs that can record and save sounds. One good program is *Audacity* (http://audacity.sourceforge.net), but there are many others.