CHAPTER

2 The first program: Little Crab

topics: writing code: movement, turning, reacting to the screen edges

concepts: source code, method call, parameter, sequence, if-statement

In the previous chapter, we discussed how to use existing Greenfoot scenarios: We created objects, invoked methods, and played a game.

Now we want to start to make our own game.

2.1 The Little Crab scenario

The scenario we use for this chapter is called *little-crab*. You will find this scenario in the *book-scenarios* folder.

The scenario you see should look similar to Figure 2.1.

Exercise 2.1 Start Greenfoot and open the *little-crab* scenario. Place a crab into the world and run the program (click the *Run* button). What do you observe? (Remember: If the class icons on the right appear striped, you have to compile the project first.)

On the right you see the classes in this scenario (Figure 2.2). We notice that there are the usual Greenfoot `Actor` and `World` classes, and subclasses called `CrabWorld` and `Crab`.

The hierarchy (denoted by the arrows) indicates an *is-a* relationship (also called *inheritance*): A crab *is an* actor, and the `CrabWorld` *is a* world.

Initially, we will work only with the `Crab` class. We will talk a little more about the `CrabWorld` and `Actor` classes later on.

If you have just done the exercise above, then you know the answer to the question "What do you observe?" It is: "nothing."

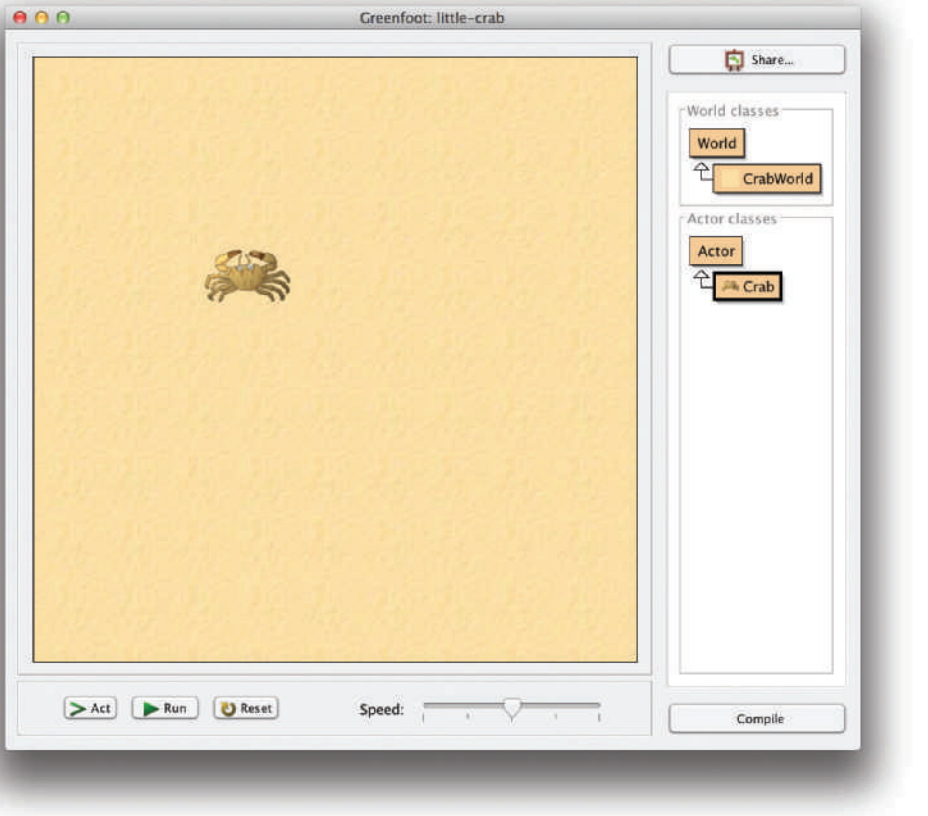

Figure 2.1
The Little Crab scenario

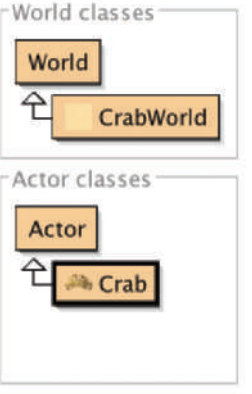

Figure 2.2
The Little Crab classes

The crab does not do anything when Greenfoot runs. This is because there is no source code in the definition of the `Crab` class that specifies what the crab should do.

In this chapter, we shall work on changing this. The first thing we will do is to make the crab move.

2.2 Making the crab move

Let us have a look at the source code of class `Crab`. Open the editor to display the `Crab` source. (You can do this by selecting the *Open editor* function from the class's popup menu, or you can just double-click the class.)

The source code you see is shown in Code 2.1.

Code 2.1
The original version of the "Crab" class

```
import greenfoot.*;

/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Actor
{
    public void act()
    {
        // Add your action code here
    }
}
```

This is a standard Java class definition. That is, this text defines what the crab can do.

You will notice the different colored backgrounds: The whole class definition is enclosed in a green box and, within it, every method definition is in a separate box with yellowish background. (There is also a separate statement at the top, before the class definition, on white background.)

We will look at this in more detail later. For now we will concentrate on getting the crab to move.

Within the class definition, we can see what is called the *act method* (the bit in the yellow box). It looks like this[1]:

```
public void act()
{
    // Add your action code here.
}
```

The first line is the *signature* of the method. The last three lines—the two curly brackets and anything between them—are called the *body* of the method. Here we can add some code that determines the actions of the crab. We can replace the grey text in the middle with a command. One such command is

```
move(5);
```

[1] In this book, when we show code inline in the text, we do not show the background colors. Don't worry about this: the colors do not alter the meaning of the code. They just help you read and write your code when you are in Greenfoot.

Note that it has to be written exactly as shown, including the parentheses and the semicolon. The act method should then look like this:

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

Exercise 2.2 Change the *act* method in your crab class to include the `move(5);` instruction, as shown above. Compile the scenario (by clicking the *Compile* button) and place a crab into the world. Try clicking the *Act* and *Run* buttons.

Exercise 2.3 Change the number 5 to a different number. Try larger and smaller numbers. What do you think the number means?

Exercise 2.4 Place multiple crabs into the world. Run the scenario. What do you observe?

Concept

A **method call** is an instruction that tells an object to perform an action. The action is defined by a method of the object.

You will see that the crab can now move across the screen. The `move(5)` instruction makes the crab move a little bit to the right.

When we click the *Act* button in the Greenfoot main window, the `act` method is executed once. That is, the instruction that we have written inside the `act` method (`move(5)`) executes. The number 5 in the instruction defines how far the crab moves in each step: In every act step, the crab moves five pixels to the right.

Clicking the *Run* button is just like clicking the *Act* button repeatedly, very quickly. The `act` method is executed over and over again, until we click *Pause*.

Exercise 2.5 Can you find a way to make the crab move backwards (to the left)?

Terminology

The instruction *move(5)* is called a **method call**. A **method** is an action that an object knows how to do (here, the object is the crab) and a **method call** is an instruction telling the crab to do it. The parentheses and number within them are part of the method call. Instructions like this are ended with a semicolon.

2.3 Turning

Let us see what other instruction we can use. The crab also understands a *turn* instruction. Here is what it looks like:

```
turn(3);
```

Concept

Additional information can be passed to some methods within the parentheses. The value passed is called a **parameter**.

The number 3 in the instruction specifies how many degrees the crab should turn. This is called a *parameter*. (The number 5 used for the `move` call above is also a parameter.)

We could also use other numbers, for example:

```
turn(23);
```

The degree value is specified out of 360 degrees, so any value between 0 and 359 can be used. (Turning 360 degrees would turn all the way around, so it is the same as turning 0 degrees, or not turning at all.)

If we want to turn instead of moving, we can replace the `move(5)` instruction with a `turn(3)` instruction. (The parameter values, 5 and 3 in this case, are picked somewhat arbitrarily; you can also use different values.) The `act` method then looks like this:

```
public void act()
{
    turn(3);
}
```

Concept

Multiple instructions are executed **in sequence**, one after the other, in the order in which they are written.

Exercise 2.6 Replace `move(5)` with `turn(3)` in your scenario. Try it out. Also, try values other than 3 and see what it looks like. Remember: every time after you change your source code, you must compile again.

Exercise 2.7 How can you make the crab turn left?

The next thing we can try is to both move and turn. The `act` method can hold more than one instruction—we can just write multiple instructions in a row.

Code 2.2 shows the complete `Crab` class, as it looks when we move and turn. In this case, at every act step, the crab will move and then turn (but this will happen so quickly after each other that it appears as if it happens at the same time).

Code 2.2
Making the crab move and turn

```
import greenfoot.*;

/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Actor
{
    public void act()
    {
        move(5);
        turn(3);
    }
}
```

Exercise 2.8 Try it out: use a `move(N)` and `turn(M)` instruction in your crab's act method. Try different values for N and M.

Terminology

The number between the parentheses in the turn instruction—i.e., the 5 in *turn(5)*—is called a **parameter**. A parameter is an additional bit of information that we have to provide when we call some methods.

Some methods do not expect any parameters. We write those by writing the method name, the parentheses, and nothing in-between, for example *stop()*. Other methods, such as *turn* and *move*, want more information: *How much should I turn? How far should I move?* In this case, we have to provide that information in the form of a parameter value between the parentheses, for instance *turn(17)*.

Side note: Errors

Concept

When a class is compiled, the compiler checks to see whether there are any errors. If an error is found, an **error message** is displayed.

When we write source code, we have to be very careful: every single character counts. Getting one small thing wrong will result in our program not working. Usually, it will not compile.

This will happen to us regularly: when we write programs, we inevitably make mistakes, and then we have to correct them. Let us try that out now.

If, for example, we forget to write the semicolon after the `move(5)` instruction, we will be told about it when we try to compile.

Exercise 2.9 Open your editor to show the crab's source code, and remove the semicolon after `move(5)`. Then compile. Also experiment with other errors, such as misspelling `move` or making other random changes to the code. Make sure to change it all back after this exercise.

Exercise 2.10 Make various changes to cause different error messages. Find at least five different error messages. Write down each error message and what change you introduced to provoke this error.

Tip

When an error message appears at the bottom of the editor window, a question mark button appears to the right of it. Clicking this button displays some additional information about the error message.

As we can see with this exercise, if we get one small detail wrong, Greenfoot will open the editor, highlight a line, and display a message at the bottom of the editor window. This message attempts to explain the error. The messages, however, vary considerably in their accuracy and usefulness. Sometimes they tell us fairly accurately what the problem is, but sometimes they are cryptic and hard to understand. The line that is highlighted is often the line where the problem is, but sometimes it is the line after the problem. When you see, for example, a "; expected" message, it is possible that the semicolon is in fact missing on the line above the highlighted line.

We will learn to read these messages a little better over time. For now, if you get a message and you are unsure what it means, look very carefully at your code and check that you have typed everything correctly.

2.4 Dealing with screen edges

When we made the crabs move and turn in the previous sections, they got stuck when they reached the edge of the screen. (Greenfoot is designed so that actors cannot leave the world and fall off its edge.)

Now we shall improve this behavior so that the crab notices that it has reached the world edge and turns around. The question is: How can we do that?

Concept

A subclass **inherits** all the methods from its superclass. That means that it has and can use all methods that its superclass defines.

Above, we have used the `move` and `turn` methods, so there might also be a method that helps us with our new goal. (In fact, there is.) But how do we find out what methods we have got available?

The `move` and `turn` methods we have used so far, come from the `Actor` class. A crab is an actor (signified by the arrow that goes from `Crab` to `Actor` in the class diagram); therefore it can do whatever an actor can do. Our `Actor` class knows how to move and turn—that is why our crab can also do it. This is called *inheritance*: the `Crab` class inherits all the abilities (methods) from the `Actor` class.

The question now is: what else can our actors do?

To investigate this, we can open the `Actor` class. You will notice when you open (double-click) the `Actor` class, it does not open in a text editor like the `Crab` class, but shows some documentation in a Web browser instead (Figure 2.3). This is because the

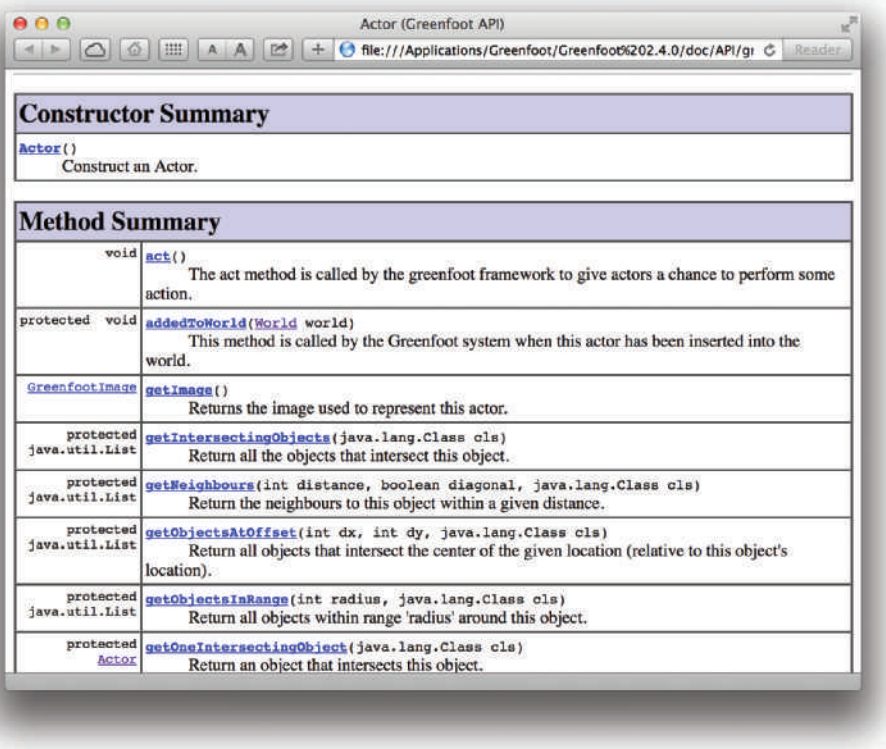

Figure 2.3
Documentation of the Actor class

`Actor` class is built-in in Greenfoot; it cannot be edited. But we can still use the `Actor`'s methods to call them. This documentation tells us what methods exist, what parameters they have, and what they do. (We can also look at the documentation of our other classes by switching the editor view from "Source Code" to "Documentation", using the pop-up control in the top right of each editor window. But for `Actor`, there is only the documentation view.)

Exercise 2.11 Open the documentation for the `Actor` class. Find the list of methods for this class (the "Method Summary"). How many methods does this class have?

Exercise 2.12 Look through the list of methods available. Can you find one that sounds like it might be useful to check whether we are at the edge of the world?

If we look at the method summary, we can see all the methods that the `Actor` class provides. Among them are three methods that are especially interesting to us at the moment. They are:

`boolean isAtEdge()`
Detect whether the actor has reached the edge of the world.

`void move(int distance)`
Move this actor the specified distance in the direction it is currently facing.

`void turn(int amount)`
Turn this actor by the specified amount (in degrees).

Here we see the signatures for three methods, as we first encountered them in Chapter 1. Each method signature starts with a return type, and is followed by the method name and the parameter list. Below it, we see a comment describing what the method does. We can see that the three method names are `isAtEdge`, `move`, and `turn`.

The `move` and `turn` methods are the ones we used in the previous sections. If we look at their parameter lists, we can see what we observed before: they each expect one parameter of type *int* (a whole number). For the `move` method, this specifies the distance to move; for the `turn` method, this is the amount to turn. (Read Section 1.5 again if you are unsure about parameter lists.)

We can also see that the `move` and `turn` methods have `void` as their return type. This means that neither method returns a value. We are commanding or instructing the object to move, or to turn. The crab will just obey the command and not respond with an answer to us.

The signature for `isAtEdge` is a little different. It is

```
boolean isAtEdge()
```

This method has no parameters (there is nothing between the parentheses), but it specifies a return value: `boolean`. We have briefly encountered the boolean type in Section 1.4—it is a type that can hold two possible values: *true* or *false*.

Concept

Calling a method with a **void return type** issues a command. Calling a method with a **non-void return type** asks a question.

Calling methods that have return values (where the return type is not *void*) is not like issuing a command, but asking a question. If we use the `isAtEdge()` method, the method will respond with either *true* (Yes!) or *false* (No!). Thus, we can use this method to check whether we are at the edge of the world.

Exercise 2.13 Create a crab. Right-click it, and find the `boolean isAtEdge()` method. (It is in the "inherited from Actor" submenu, since the crab inherited this method from the `Actor` class). Call this method. What does it return?

Exercise 2.14 Let the crab run to the edge of the screen (or move it there manually), and then call the `isAtEdge()` method again. What does it return now?

We can now combine this method with an *if-statement* to write the code shown in Code 2.3.

Code 2.3 Turning around at the edge of the world

```
import greenfoot.*;

/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Actor
{
    public void act()
    {
        if ( isAtEdge() )
        {
            turn(17);
        }
        move(5);
    }
}
```

Concept

An **if-statement** can be used to write instructions that are only executed when a certain condition is true.

The if-statement is part of the Java language that makes it possible to execute commands only if some condition is true. For example, here we want to turn only if we are at the edge of the world. The code we have written is:

```
if ( isAtEdge() )
{
    turn(17);
}
move(5);
```

The general form of an *if-statement* is this:

```
if ( condition )
{
    instruction;
    instruction;
    ...
}
```

Tip

In the Greenfoot editor, when you place the cursor behind an opening or closing bracket, Greenfoot will mark the matching closing or opening bracket. This can be used to check whether your brackets match up as they should.

In place of the *condition* can be any expression that is either true or false (such as our `isAtEdge()` method call), and the *instructions* will only be executed if the condition is true. There can be one or more instructions.

If the condition is false, the instructions are just skipped, and execution continues under the closing curly bracket of the if-statement.

Note that our `move(5)` method call is outside the if-statement, so it will be executed in any case. In other words: If we are at the edge of the world, we turn and then move; if we are not at the edge of the world, we just move.

Exercise 2.15 Try it out! Type in the code discussed above, and see if you can make your crabs turn at the edge of the screen. Pay close attention to the opening and closing brackets—it is easy to miss one or have too many.

Exercise 2.16 Experiment with different values for the parameter to the `turn` method. Find one that looks good.

Exercise 2.17 Place the `move(5)` statement into the if statement, rather than behind it. Test—what is the effect? Explain the behavior you observe. (Then fix it again by moving it back where it was.)

Side note: Scope coloring and indentation

When you look at source code in Greenfoot or in the code examples in this book (for instance, Code 2.3), you will notice the colored boxes used for the background. These are called *scopes*. A scope is the extent of a given Java construct. In Greenfoot, different kinds of construct have been given different colors: a class is green, for example, a method is yellow, and an if-statement is a purplish-grey. You see that these scopes can be *nested*: an if-statement is inside a method, a method is inside a class.

Paying attention to these colored scopes pays off over time; they can help you avoid some common errors. Scopes are usually defined in your code by a pair of curly brackets (usually with a header above the opening bracket that defines what kind of scope we are looking at). It can happen very easily to get the curly brackets out of balance—to have more opening than closing ones, or vice versa. If this happens, your program will not compile.

Scope coloring helps you detect such a problem. You will get used to what the scopes should look like quite quickly, and you will notice that it just looks wrong when a bracket is mismatched.

Hand-in-hand with scope colors goes indentation.

In all the code examples you have seen so far, you may have noticed some careful indentation being used. Every time a curly bracket opens, the following lines are indented one level more than the previous ones. When a curly bracket closes, the indentation

goes back one level, so that the closing curly bracket is directly below the matching opening bracket. This makes it easy to find the matching bracket.

We use four spaces for one level of indentation. The TAB key will insert spaces in your editor for one level of indentation. Greenfoot can also help you should your indentation get too messy: the editor has an `Auto-Layout` function in its `Edit` menu, which will try to fix your indentation for the whole class.

Taking care with indentation in your own code is very important. If you do not indent carefully, the scope coloring will look messy, become useless, and some errors (particularly misplaced or mismatched curly brackets) are very hard to spot. Proper indentation makes code much easier to read, and thus avoids potential errors.

Exercise 2.18 Open the source code for your `Crab` class. Remove various opening or closing curly brackets and observe the change in scope coloring. In each case, can you explain the change in color? Also experiment with changing the indentation of brackets and other code and observe how it affects the look. At the end, fix the brackets and indentation so that the code looks nice again.

Summary of programming techniques

In this book, we are discussing programming from a very example-driven perspective. We introduce general programming techniques as we need them to improve our scenarios. So from now on, we shall summarize the important programming techniques at the end of each chapter, to make it clear what you really need to take away from the discussion to be able to progress well.

In this chapter, we have seen how to call methods (such as `move(5)` or `isAtEdge()`), with and without parameters. This will form the basis for all further Java programming. We have also learnt to identify the body of the act method—this is where we start writing our instructions.

You have encountered some error messages. This will continue throughout all your programming endeavors. We all make mistakes, and we all encounter error messages. This is not a sign of a bad programmer—it is a normal part of programming.

We have encountered a first glimpse of inheritance: Classes inherit the methods from their superclasses. The documentation of a class gives us a summary of the methods available.

And, very importantly, we have seen how to make decisions: We have used an if-statement for conditional execution. This went hand in hand with the appearance of the type *boolean*, a value that can be *true* or *false*.

Concept summary

- A **method call** is an instruction that tells an object to perform an action. The action is defined by a method of the object.
- Additional information can be passed to some methods within the parentheses. The value passed is called a **parameter**.
- Multiple instructions are executed **in sequence**, one after the other, in the order in which they are written.
- When a class is compiled, the compiler checks to see whether there are any errors. If an error is found, an **error message** is displayed.
- A subclass **inherits** all the methods from its superclass. That means that it has, and can use, all methods that its superclass defines.
- Calling a method with a **void return type** issues a command. Calling a method with a **non-void return type** asks a question.
- An **if-statement** can be used to write instructions that are only executed when a certain condition is true.

Drill and practice

Some of the chapters include "Drill and practice" sections at the end. These sections introduce no new material but give you a chance to practice an important concept that has been introduced in this chapter in another context, and to deepen your understanding.

The two most important constructs we have encountered in this chapter are method calls and if-statements. Here we do some more exercises with these two constructs. (See Figure 2.4.)

Figure 2.4
Fat Cat

Method signatures

Exercise 2.19 Look at the following method signatures:

```
public void play();
public void addAmount(int amount);
public boolean hasWings();
public void compare(int x, int y, int z);
public boolean isGreater (int number);
```

For each of these signatures, answer the following questions (in writing):

a) What is the method name?
b) Does the method return a value? If yes, what is the type of the return value?
c) How many parameters does the method have?

Exercise 2.20 Write a method signature for a method named "go." The method has no parameters, and it does not return a value.

Exercise 2.21 Write a method signature for a method named "process." The method has a parameter of type "int" that is called "number", and it returns a value of type "int."

Exercise 2.22 Write a method signature for a method named "isOpen." This method has no parameters and returns a value of type "boolean."

Exercise 2.23 On paper, write a method call (note: this is a method call, not a signature) for the "play" method from Exercise 2.19. Write another method call for the "addAmount" method from Exercise 2.19. And finally, write a method call for the "compare" method from the same exercise.

Reading documentation

All the following exercises are intended to be implemented in the Greenfoot scenario "fatcat." Open the scenario in Greenfoot before continuing.

Exercise 2.24 Open the editor for class `Cat`. Change the view of the editor from "Source Code" to "Documentation" view using the control in the top right of the editor window. How many methods does the class `Cat` have?

Exercise 2.25 How many of the `Cat`'s methods return a value?

Exercise 2.26 How many parameters does the `sleep` method have?

Writing method calls (with and without parameters)

Exercise 2.27 Try calling some of your cat's methods interactively, by using the cat's popup menu. The interesting methods are all "inherited from Cat."

Exercise 2.28 Is the cat bored? How can you make it not bored?

Exercise 2.29 Open the editor for class `MyCat`. (This is where you will write the code for all the following exercises.)

Exercise 2.30 Make the cat eat when it acts. (That is, in the `act` method, write a call to the `eat` method.) Compile. Test by pressing the Act button in the execution controls.

Exercise 2.31 Make the cat dance. (Don't do this interactively—write code in the act method to do this. When done, click the Act button in the execution controls.)

Exercise 2.32 Make the cat sleep.

Exercise 2.33 Make the cat do a routine of your choice, consisting of a number of the available actions in sequence.

If-statements

Exercise 2.34 Change the `act` method of your cat so that, when you click Act, if the cat is tired, it sleeps a bit. If it is not tired, it doesn't do anything.

Exercise 2.35 Change the `act` method of your cat so that it dances if it is bored. (But only if it is bored.)

Exercise 2.36 Change the `act` method of your cat so that it eats if it is hungry.

Exercise 2.37 Change the `act` method of your cat to the following: If the cat is tired, it sleeps a bit, and then it shouts hooray. If it is not tired, it just shouts hooray. (For testing, make the cat tired by calling some methods interactively. How can you make the cat tired?)

Exercise 2.38 Write code in the `act` method to do the following: If your cat is alone, let it sleep. If it is not alone, make it shout "Hooray." Test by placing a second cat into the world before clicking Act.