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 have created objects, invoked methods, and played a game.

Now, we shall 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 projects for this book.

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 is the usual Greenfoot `Actor` class, a class called `Animal`, and the `Crab` class.

The hierarchy (denoted by the arrows) indicates an *is-a* relationship (also called *inheritance*): A crab *is an* animal, and an animal *is an* actor. (It follows then, that a crab also is an actor.)

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

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

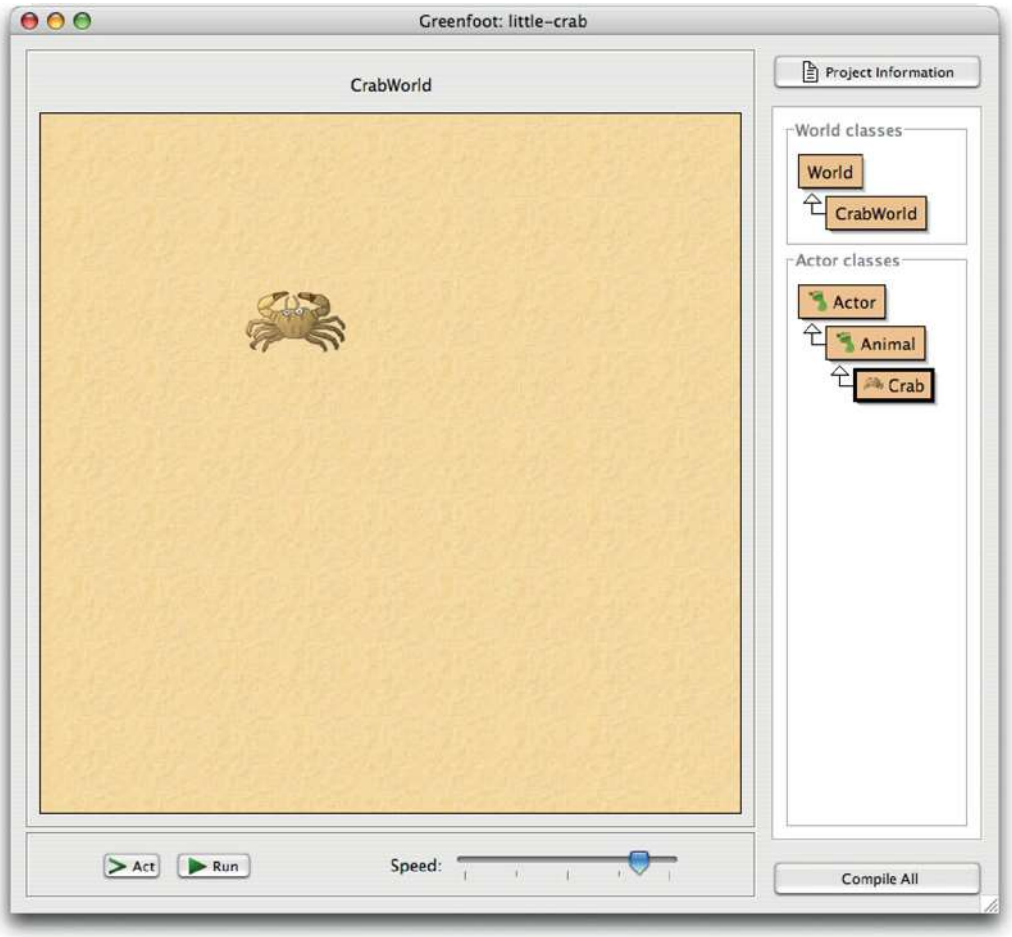

Figure 2.1
The *Little Crab* scenario

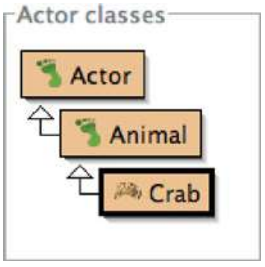

Figure 2.2
The *Little Crab* actor classes

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 shall 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.*; // (World, Actor, GreenfootImage, and Greenfoot)

/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Animal
{
    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. We will look at it in detail a little later. For now, we will concentrate on getting the crab to move.

Within this class definition, we can see what is called the *act method.* It looks like this:

```
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—is called the *body* of the method. Here, we can add some code that determines the actions of the crab. We can replace the gray text in the middle with a command. One such command is

```
move();
```

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();
}
```

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

Exercise 2.3 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()` 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()`) executes.

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

Terminology

The instruction `move()` 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 are part of the method call. Instructions like this end with a semicolon.

2.3 Turning

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

```
turn(5);
```

The number 5 in the instruction specifies how many degrees the crab should turn. This is called a *parameter*. We can 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.)

Concept:

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

If we want to turn instead of moving, we can replace the `move()` instruction with a `turn(5)` instruction. The `act` method then looks like this:

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

Exercise 2.4 Replace `move()` with `turn(5)` in your scenario. Try it out. Also, try values other than 5 and see what it looks like. Remember: Every time you change your source code, you must compile again.

Exercise 2.5 How can you make the crab turn left?

Concept:

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

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 these actions will happen so quickly after each other that it appears as if they happen at the same time).

Code 2.2
Making the crab move and turn

```
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()
    {
        move();
        turn(5);
    }
}
```

Exercise 2.6 Use a `move()` and `turn(N)` instruction in your crab's `act` method. Try different values for *N*.

Terminology

The number within the parentheses in the `turn` instruction—that is, 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, like `move`, expect no parameters. They are happy to just execute as soon as we write the `move()` instruction. Other methods, such as `turn`, want more information: *How much should I turn?* In this case, we have to provide that information in the form of a parameter value within 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()` instruction, we will be told about it when we try to compile.

Exercise 2.7 Open your editor to show the crab's source code, and remove the semicolon after `move()`. 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 2.8 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 `Animal` class. A crab is an animal (signified by the arrow that goes from `Crab` to `Animal` in the class diagram), therefore it can do whatever an animal can do. Our `Animal` 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 `Animal` class.

The question now is, what else can our animals do?

To investigate this, we can open the editor for the `Animal` class. The editor can display two different views: It can show the source code (as we have seen for the `Crab` class) or it can show the documentation. The view can be switched using a popup selection menu in the top right corner of the editor window. We now want to look at the `Animal` class in the *Documentation view* (Figure 2.3).

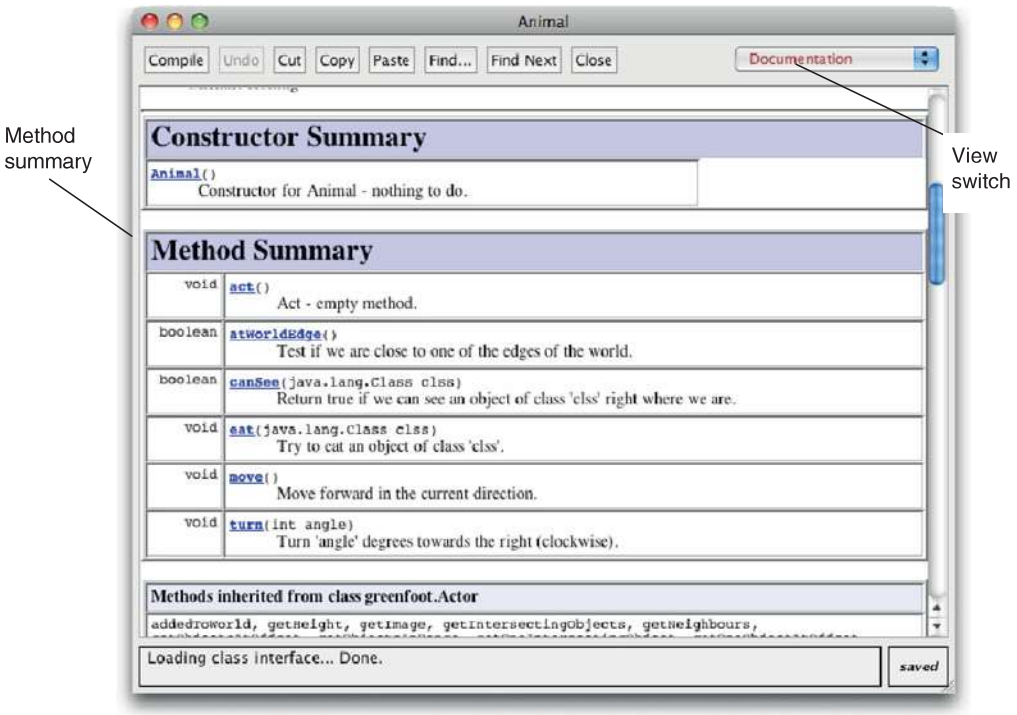

Figure 2.3
Documentation view (with method summary) of the `Animal` class

Exercise 2.9 Open the editor for the `Animal` class. Switch to Documentation view. Find the list of methods for this class (the "Method Summary"). How many methods does this class have?

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

`boolean atWorldEdge()`
Test if we are close to one of the edges of the world.

`void move()`
Move forward in the current direction.

`void turn(int angle)`
Turn "angle" degrees toward the right (clockwise).

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 `atWorldEdge`, `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: `move` has no parameters (the parentheses are empty), and `turn` expects one parameter of type `int` (a whole number) for the angle. (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 animal will just obey the command but not respond with an answer to us.

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

```
boolean atWorldEdge()
```

Concept:

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

This method has no parameters (there is nothing within 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*.

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 `atWorldEdge()` 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 2.10 Create a crab. Right-click it, and find the `boolean atWorldEdge()` method (it is in the *inherited from Animal* submenu, since the crab inherited this method from the `Animal` class). Call this method. What does it return?

Exercise 2.11 Let the crab run to the edge of the screen (or move it there manually), and then call the `atWorldEdge()` 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.*; // (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);
        }
        move();
    }
}
```

Concept:

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

The if-statement is a 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 near the edge of the world. The code we have written is:

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

The general form of an if-statement is this:

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

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

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.

Exercise 2.12 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 2.13 Experiment with different values for the parameter to the `turn` method. Find one that looks good.

Exercise 2.14 Place the `move()` statement inside the if-statement, rather than after it. Find out what is the effect and explain the behavior you observe. (Then, fix it again by moving it back where it was.)

Note: Indentation

In all the code examples you have seen so far (for instance, Code 2.3), 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 as well.

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

2.5 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. 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 in order to progress well.

In this chapter, we have seen how to call methods (such as `move()`), 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 view 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 executed only when a certain condition is true.