

Introduction

Welcome to Greenfoot! In this book, we will discuss how to program graphical computer programs, such as simulations and games, using the Java Programming Language and the Greenfoot environment.

There are several goals in doing this: one is to learn programming; another is to have fun along the way. While the examples we discuss in this book are specific to the Greenfoot environment, the concepts are general: working through this book will teach you general programming principles in a modern, object-oriented programming language. However, it will also show you how to make your own computer game, a biology simulation, or an on-screen piano.

This book is very practically oriented. Chapters and exercises are structured around real, hands-on development tasks. First, there is a problem that we need to solve, then we look at language constructs and strategies that help us solve the problem. This is quite different from many introductory programming textbooks that are often structured around programming language constructs.

As a result, this book starts with less theory and more practical activity than most programming books. This is also the reason we use Greenfoot: It is the Greenfoot environment that makes this possible. Greenfoot allows us to play. And that does not only mean playing computer games; it means playing with programming: we can create objects, move them around on screen, call their methods, and observe what they do, all interactively and easily. This leads to a more hands-on approach to programming than what would be possible without such an environment.

A more practical approach does not mean that the book does not cover the necessary theory and principles as well. It's just that the order is changed. Instead of introducing a concept theoretically first and then doing some exercises with it, we often jump right in and use a construct, initially explaining only as much as necessary to solve the task at hand, then come back to the theoretical background later. We typically follow a spiral approach: we introduce some aspects of a concept when we first encounter it, then revisit it later in another context, and gradually deepen our understanding.

The emphasis throughout is to make the work we do interesting, relevant, and enjoyable. There is no reason why computer programming has to be dry, formal, or boring. Having fun along the way is okay. We think we can manage to make the experience interesting and pedagogically sound at the same time.

This book can be used both as a self-study book or as a textbook in a programming course. Exercises are worked into the text throughout the book—if you do them all, you will come out of this as a fairly competent programmer.

The projects discussed in this book are easy enough that they can be managed by high school students, but they are also open and extendable enough that even seasoned programmers can find interesting and challenging aspects to do. While Greenfoot is an educational environment, Java is not a toy language. Since Java is our language of choice for this book, the projects discussed here (and others you may want to create in Greenfoot) can be made as complex and challenging as you like.

While it is possible to create simple games quickly and easily in Greenfoot, it is equally possible to build highly sophisticated simulations of complex systems, possibly using artificial intelligence algorithms, agent technology, database connectivity, network communication, or anything else you can think of. Java is a very rich language that opens the whole world of programming, and Greenfoot imposes no restrictions as to which aspects of the language you can use.

In other words: Greenfoot scales well. It allows easy entry for young beginners, but experienced programmers can also implement interesting, sophisticated scenarios.

Programming is a creative discipline, and Greenfoot is a tool that helps you build what you invent.

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."

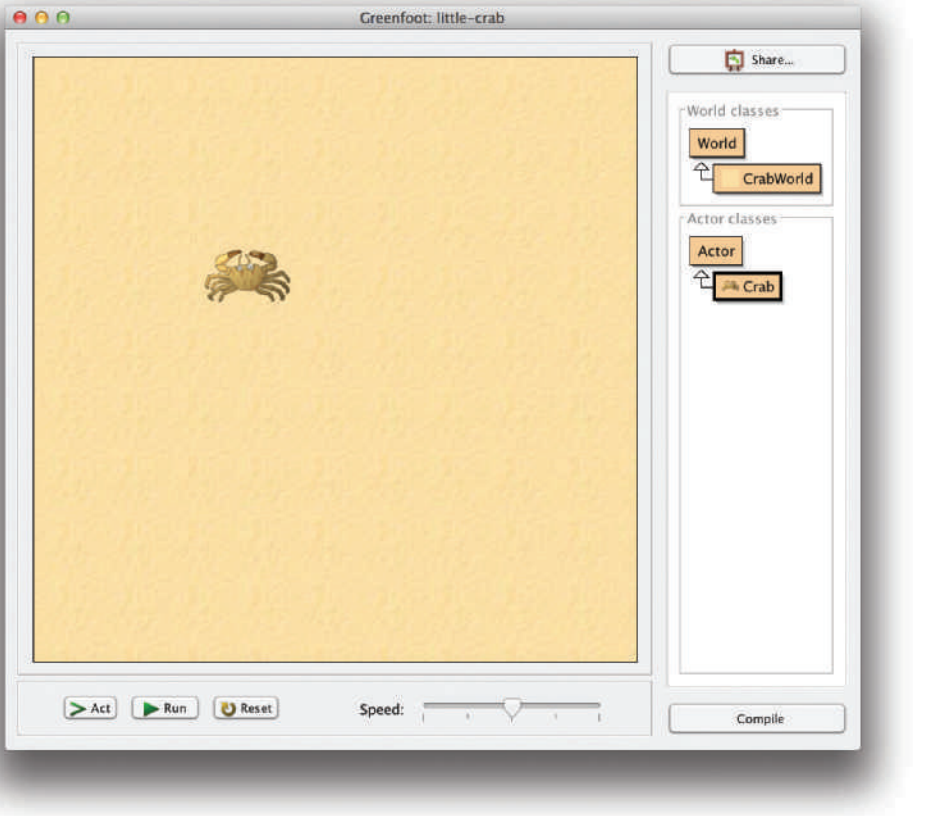


Figure 2.1
The Little Crab scenario

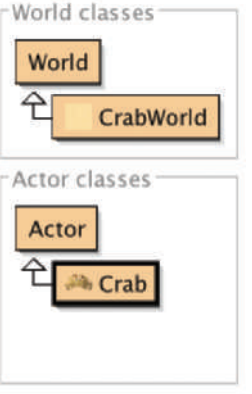


Figure 2.2
The Little Crab classes

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

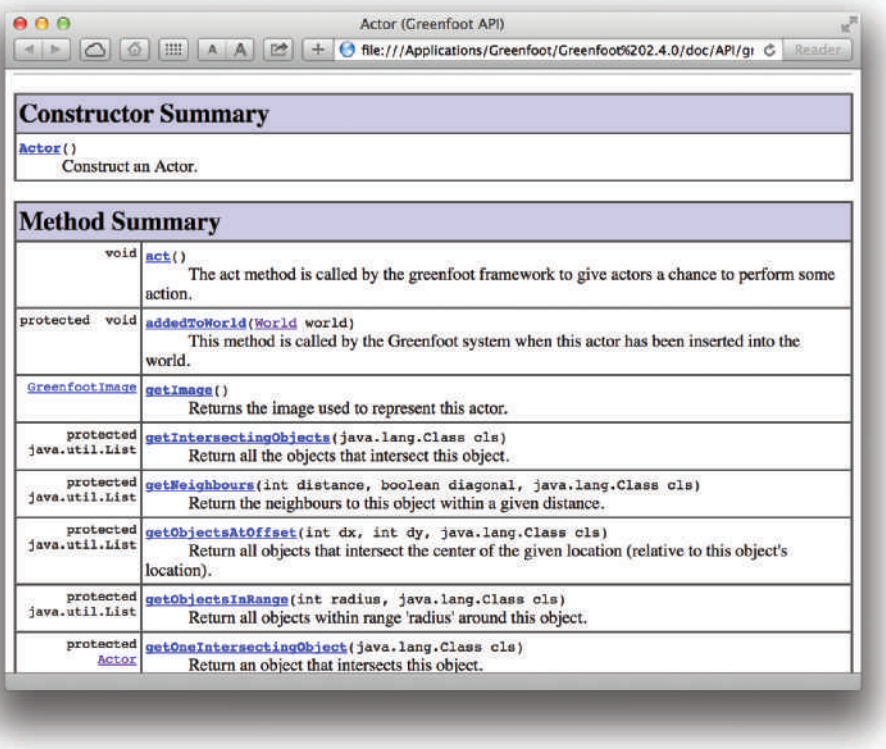


Figure 2.3
Documentation of the Actor class

`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.

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 will do is add 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.

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**.

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,

```
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 to 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 `Actor` 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 Actor
{
    public void act()
    {
        if ( isAtEdge() )
        {
            turn(17);
        }

        if ( Greenfoot.getRandomNumber(100) < 10 )
        {
            turn(4);
        }

        move(5);
    }
}
```

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 perfect yet. First of all, if the crab turns, it always turns the same amount (four 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 before reading on.)

The simple trick to the first problem—always turning the same amount, in our case 4 degrees—is to replace the fixed number 4 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 right only. 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?

Exercise 3.5 is not easy at first if you have never seen anything like this before and probably deserves a hint. There is a temptation at first to perhaps use an if-statement to deal with the two directions, but there is actually a much simpler solution.

Think about it this way: `Greenfoot.getRandomNumber` only gives us numbers with a lower bound of zero. We want numbers from −45 to 45. That means we want (roughly) 90 different numbers. We can get a range of 90 numbers using `Greenfoot.getRandomNumber(90)`, but this gives us 0–90. How can we get from 0–90 to −45–45?

The answer is: we can just get random numbers out of 90, and then subtract 45. Figure 3.1 attempts to illustrate this. (Note that this is actually out by 1: the range of the random number is actually 0–89, so after subtracting 45 we end up with −45–44. But since we don't care about accuracy with our random turns, that is okay.)

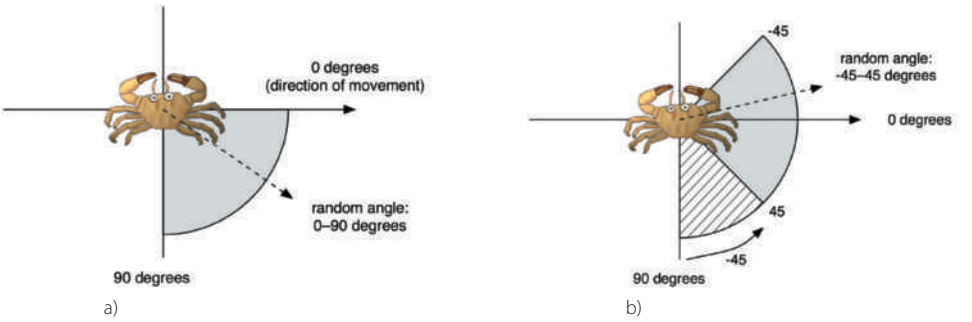


Figure 3.1
a) Random angle 0–90 degrees b) Random angle -45–45 degrees

Try solving this exercise yourself. The project *little-crab-2* (included with this book) 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 3.2). In this case, our new class `Worm` is a subclass of class `Actor`. (Remember, being a subclass is an *is-a* relationship: a worm *is an* actor.)

When we are creating a new subclass, we are prompted to enter a name for the class and to select an image (Figure 3.3).

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.

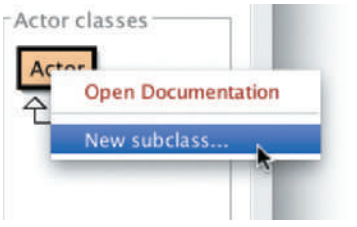


Figure 3.2
Creating new subclasses

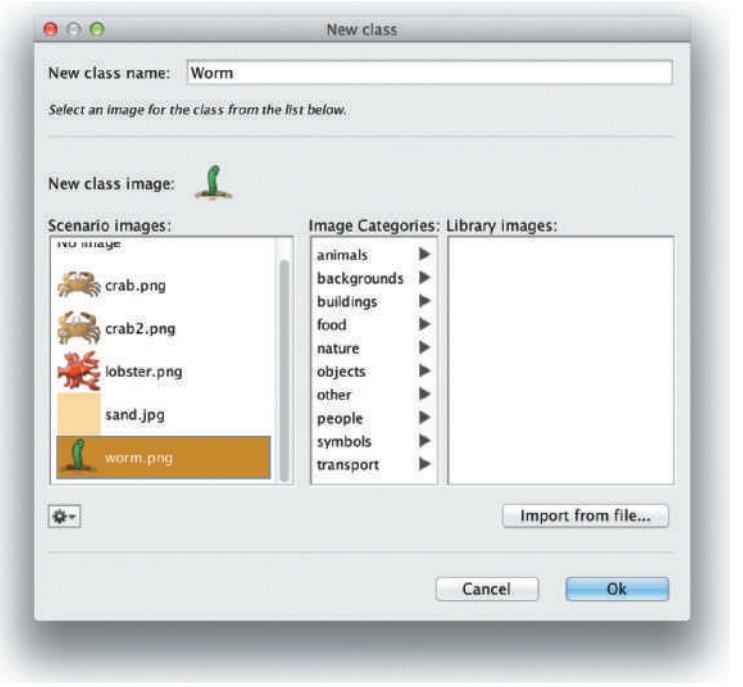


Figure 3.3
Creating a new class

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*.

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.

3.3 Eating worms

We now want to add new behavior to the crab: when a crab runs into a worm, it eats it. Again, we first check what methods we have already inherited from the `Actor` class. When we open the documentation for class `Actor` again, we can see the following two methods:

`boolean isTouching (java.lang.Class cls)`
Check whether this actor is touching any other objects of the given class.

`void removeTouching (java.lang.Class cls)`
Remove one object of the given class that this actor is currently touching (if any exist).

Using these methods, we can implement this behavior. The first method checks whether the crab touches a worm. This method returns a boolean—*true* or *false*—so we can use it in an if-statement.

The second method removes 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 ( isTouching(Worm.class) )
{
    removeTouching(Worm.class);
}
```

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

Code 3.2
First version of eating a worm

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

    if ( Greenfoot.getRandomNumber(100) < 10 )
    {
        turn(Greenfoot.getRandomNumber(90)-45);
    }

    move(5);

    if ( isTouching(Worm.class) )
    {
        removeTouching(Worm.class);
    }
}
```

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.

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 `isTouching` and `removeTouching` methods, we have seen a parameter type with the name `java.lang.Class`. What does this mean?

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 are 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 removes 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 ( isTouching(Worm.class) )
    {
        removeTouching(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 ( isAtEdge() )
    {
        turn(17);
    }

    if ( Greenfoot.getRandomNumber(100) < 10 )
    {
        turn(Greenfoot.getRandomNumber(90)-45);
    }

    move(5);
    lookForWorm();
}
```

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

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 sub-tasks

```
public void act()
{
    turnAtEdge();
    randomTurn();
    move(5);
    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 again, let us add another creature: a lobster (Figure 3.4).

Lobsters, in our scenario, like to chase crabs.



Figure 3.4
Adding an enemy: a lobster

Exercise 3.9 Add a new class to your scenario. The class should be a subclass of `Actor`, called `Lobster` (with a capital "L"), and it should use the prepared image *lobster.png*.

Exercise 3.10 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 3.11 Copy the complete act method from the `Crab` class into the `Lobster` class. Also copy the complete `lookForWorm`, `turnAtEdge`, and `randomTurn` methods.

Exercise 3.12 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 3.13 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, e.g., 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.

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 3.14 Remove the random turning code from the crab.

Exercise 3.15 Remove the code from the crab that does the turn at the edge of the world.

Tip

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

Exercise 3.16 Add code into the crab's act method that makes the crab turn left whenever the left cursor key is pressed. Test.

Exercise 3.17 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 3.18 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.

Exercise 3.19 Currently, if you have simply copied the crab code to the lobster, both walk with the same speed. You can make them walk at different speeds by changing the parameter to the `move(5)` method call in either of them. Try making lobsters faster or slower. Try out how the game feels if you do that. Choose a speed for the crab and lobster that suits you.

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.

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 3.5).

This documentation is also called the *Greenfoot API* (Application Programmers' Interface). The API shows all available classes and for each class, all the available methods. You can see that Greenfoot offers seven classes: `Actor`, `Greenfoot`, `GreenfootImage`, `GreenfootSound`, `MouseInfo`, `UserInfo`, and `World`.

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(5);
    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);
    }
}
```

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.

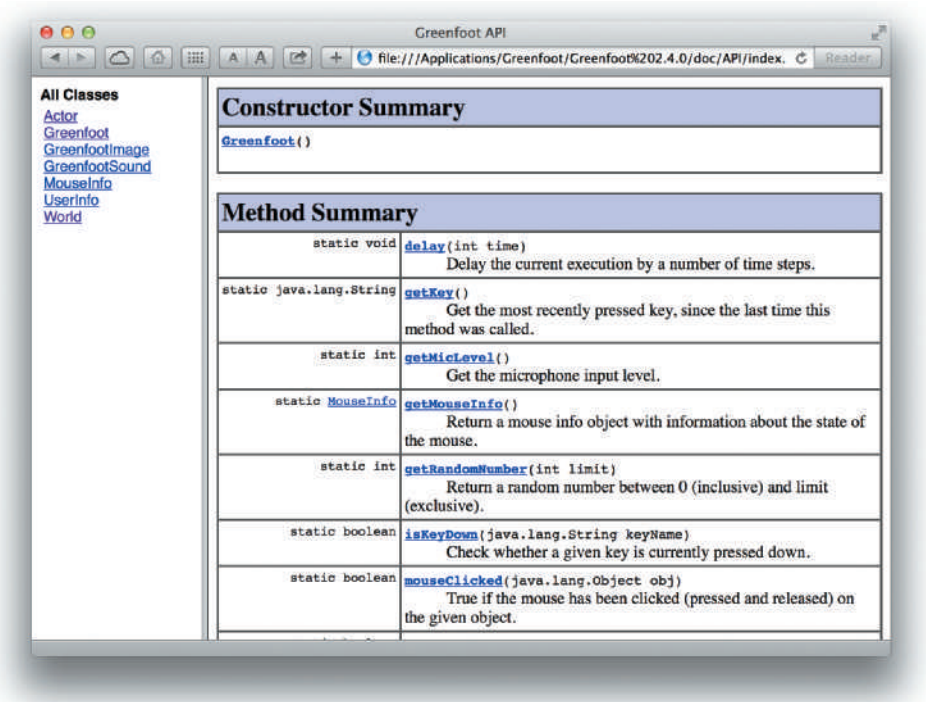


Figure 3.5
The Greenfoot API in a browser window

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 is executed when a lobster catches 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) to find out about playing sounds. There is a `GreenfootSound` class for full sound control, and also a convenience method in the `Greenfoot` class to produce a sound quickly and easily. Find the details of the method in the `Greenfoot` class. 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 (a 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 3.6).

3.9 Making your own sounds

There are various ways to add your own sounds to a Greenfoot scenario. You can, for example, find sound effects in various sound libraries on the Internet or produce your own using some sound recording and effects software.

Using sounds sometimes gets a bit tricky because sounds on a computer can be stored in many different file formats, and Greenfoot can play some, but not others. Greenfoot can generally play sound files in MP3, AIFF, AU, and WAV formats (although certain WAV files cannot be played—it gets complicated).

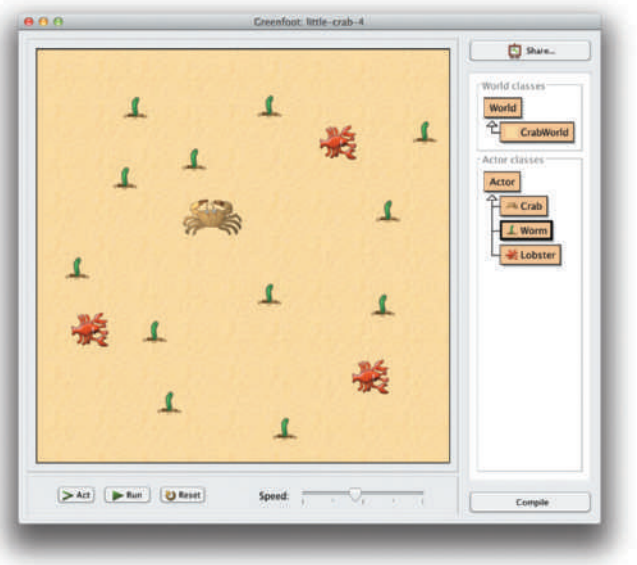


Figure 3.6
The crab game with worms and lobsters

We will discuss sound more completely in Chapter 10; here, we will use the easiest method to get our own sounds into our scenario: recording them ourselves directly in Greenfoot.

Both of the sound effects we used in the previous section were recorded by simply speaking into the microphone. Greenfoot has a sound recorder built-in for you to do the same.

To get started, use the *Show Sound Recorder* function from the *Controls* menu. You will see the Greenfoot sound recorder control (Figure 3.7).

Using this sound recorder, you can now record your own sounds by pressing the *Record* button and speaking into the microphone.[1] Press *Stop Recording* when you are finished. You can use the *Play* button to check what your recording sounds like.

There is only one edit operation available: *Trim to selection*. The purpose is to cut off unwanted sections of the recording at the beginning and then end. Often you will have a bit of noise or silence at the beginning of the recording, and this will not sound good in your program: silence at the beginning will make your sound appear delayed.

To remove the unwanted parts, select the part of the sound you want to keep with your mouse (Figure 3.8) and then press *Trim to selection*.

Finally, choose a name for your sound, type it into the filename field, and click *Save*. The sound will be saved in WAV format and will automatically receive a ".wav" filename suffix. So if you, for example, call your sound file "bark," it will be saved as "bark.wav." Greenfoot will automatically save it to the right location (the "sounds" folder in your scenario folder).

[1] Obviously, this only works if your computer has a microphone. Most recent laptops have microphones built-in. For some desktop computers, you will have to connect an external microphone. If you do not have a microphone, just skip this section.

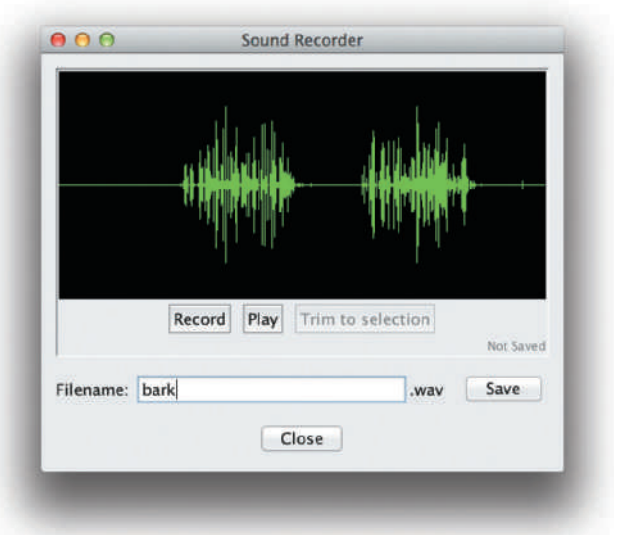


Figure 3.7
The Greenfoot sound recorder

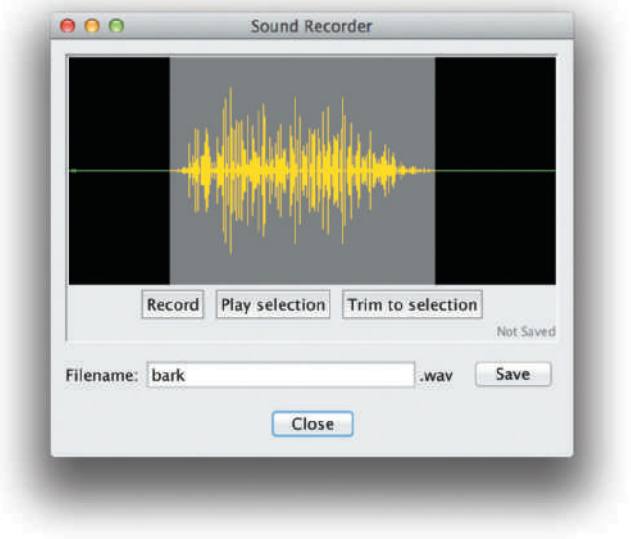


Figure 3.8
Selecting the good part of the sound for trimming

You can then use it in your code using the method we have seen before:

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

It's time to try for yourself.

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 them, then use them in your code.



Figure 3.9
Using code completion (CTRL+Space)

3.10 Code completion

A useful productivity tip is to use *code completion* to enter your method calls (Figure 3.9). You can use code completion whenever you are about to type the name of a method you want to call. It is activated by typing CTRL+Space.

Tip

Use **code completion** to make entry of method calls easier.

For example, if you type

```
Greenfoot.
```

and the cursor is behind the dot, and then you type CTRL+Space, a dialog will pop up listing all the methods you can call here (in this case: all the methods of the `Greenfoot` class). If you then start to type the beginning of a method name (for example, in Figure 3.9 we have typed the letter "s"), the list of methods reduces to those starting with what you have typed.

You can select methods from this list using your cursor keys or with your mouse, and insert them into your code by using the *Return* key.

Using code completion is useful to investigate which methods exist, to find a method that you know exists but cannot remember the exact name of, to look up parameters and documentation, or just to save yourself some typing and speed up entry of your code.

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*), methods 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.

Finally, a very important concept we have encountered is the ability to define our own methods. We have seen how to define methods for distinct subtasks, and how to call them from other methods.

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.

Drill and practice

The concepts that we want to reinforce here are if-statements (again), reading API documentation, calling a method from another class, and creating our own methods.

To do this, we will use another scenario: *stickman*. Find it in the book scenarios and open it.

Reading API documentation

Exercise 3.26 The `Greenfoot` class has a method to get the noise level from the computer's microphone. Find this method in the API documentation. How many parameters does it have?

Exercise 3.27 What is the return type of this method? What does this tell us?

Exercise 3.28 What are the possible values returned from this method?

Exercise 3.29 Is this method *static* or not? What does that tell us?

Exercise 3.30 How do we call this method? Write down a correct method call to this method.

Calling class methods/if-statements

Exercise 3.31 In your *stickman* scenario, make the stickman move to the right, so that when you run your scenario, he walks over to the right side of the screen.

Exercise 3.32 Using an if-statement and the microphone input method you found above, make the stickman move right only when you make some noise. Experiment with different values for the noise level. This will depend on your microphone and your environment. A good starting point is to make him move when the microphone level is greater than 3. Test.

Exercise 3.33 Make the stickman move left when you make some noise, but move continuously right if there is no noise. Test. Try to keep him near the center of the screen by shouting.

Defining methods

Exercise 3.34 Move the code that moves left when you make noise into its own method. Call this method `moveLeftIfNoise`. Test. Make sure it works as before.

Exercise 3.35 Add another method to your class called `gameOver`. This method should be called if the stickman reaches the edge of the screen. When called, the method plays a game-over sound and stops the scenario.

Exercise 3.36 Move the check for the game-over condition itself into a separate method called `checkGameOver`. The `act` method should call `checkGameOver`, which in turn calls `gameOver` if appropriate.

Exercise 3.37 Make the stickman float up when there is noise. The height he floats should be proportionate to the noise level. He comes down if there is no noise. Note: the proportionate height requirement is quite difficult—it requires you to look up some things we have not discussed yet. If you find it too hard, leave this out.

Exercise 3.38 Introduce another actor (maybe an animal) that starts at the left screen edge and moves sideways. If it reaches the right edge of the screen, it gets reset to the left side. On its path across the screen, it should touch the stickman if he is resting. You can then make the stickman jump over the animal by making some noise.

Exercise 3.39 Stop the scenario (with game-over sound) when the stickman touches the animal.