BotsInc: Learn programming with robots

Stéphane Ducasse

Version du 08/05/2009

Ce livre est disponible en libre téléchargement depuis http://. L'édition originale de ce livre est disponible depuis http:// sous le titre *Squeak par l'exemple*, ISBN 978-3-9523341-3-3. La première édition a été publié en Avril 2008 par *Square Bracket Associates*, Suisse (SquareBracketAssociates.org). Vous pouvez vous procurer une copie à l'adresse: SqueakByExample.org/fr.

Contents

Working on chapter 11. Composing messages p125 of pdf

Preface

Knowledge is only one part of understanding.Genuine understanding comes from hands-on experience. —S. Papert

Goals and Audience

The goal of this book is to explain elementary programming concepts (such as loops, abstraction, composition, and conditionals) to novices of all ages. I believe that learning by experimenting and solving problems is central to human knowledge acquisition. Therefore, I have presented programming concepts through simple problems such as drawing golden rectangles or simulat- ing animal behavior.

My ultimate goal is to teach you object-oriented programming, because this particular paradigm provides an excellent metaphor for teaching programming. However, object- oriented programming requires some more elementary notions of programming and abstrac- tion. Therefore, I wrote this book to present these basic programming concepts in an elementary programming environment with the special perspective that this book is the first in a series of two books. Nevertheless, this book is completely self-contained and does not require you to read the next one. The second book introduces another small programming environment. It focuses on intermediate-level topics such as finding a path through a maze and drawing frac- tals. It also acts as a companion book for readers who want to know more and who want to adapt the environment of this book to their own needs. Finally, it introduces object-oriented programming.

The ideal reader I have in mind is an individual who wants to have fun programming. This person may be a teenager or an adult, a schoolteacher, or somebody teaching program- ming to children in some other organization. Such an individual does not have to be fluent in programming in any language.

The material of this book was originally developed for my wife, who is a physics and mathematics teacher in a French school where the students are between eleven and fifteen years old. In late 1998, my wife was asked to teach computing science, and she was dismayed by the lack of appropriate material. She started out teaching HTML, Word, and other topics, and she remained dissatisfied, since these approaches failed to promote a scientific attitude toward computing science. Her goal was to teach computer science as a process of attacking problems and finding solutions.

As a computer scientist, I was aware of work on the programming language Logo, and I particularly liked the idea of experimentation as a basis for learning. I was also aware that the programming language Smalltalk had been influenced by the ideas of Seymour Papert and those behind Logo, and that it had originated from research on teaching programming to children. Moreover, Smalltalk has a simple syntax that mimics natural language. At about that time, the Squeak environment had arrived at a mature state, and books started to become available in late 1999. But these were for experienced programmers, so I started and wrote the present book.

The environments that I use in this book and its companion book are fully functional. They have gone through several iterations of improvements based on the feedback that I have received from teachers. A guiding rule in our work has been to modify the Squeak environment as little as possible, for our goal is for readers to be able to extend the ideas presented in this book and develop new ones of their own.

Object-Oriented Structure and Vocabulary

The chapters of this book are relatively small, so that each chapter can be turned into a one- or two-hour lab session. I do not advocate presenting the material directly to children for self- instruction, but each chapter in fact has all the material for such an approach. Although object-oriented programming is not developed in this book, I use its vocabu- lary. That is, we create objects from classes and send them messages. Object behavior is defined by methods. I made this choice because the metaphor offered by object-oriented pro- gramming is natural, and children have an intuitive understanding of the idea of objects and their behavior.

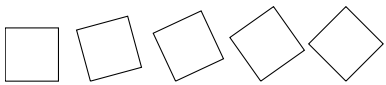
Those who are used to Logo may wonder why our robots do not have "pen up" and "pen down" methods, but instead "go" and "jump," where under the former, a robot moves leaving a trace, while the latter moves a robot forward without leaving a trace. I believe that the go and jump paradigm is better suited to the ideas of object-oriented programming and encapsula- tion of data than the traditional pen down and pen up design.

An excellent analysis of these two approaches was made by Didier Besset, who collaborated with me on this project in its early stages.

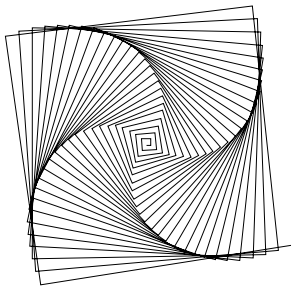
Organization

The book is divided into five parts, as described below.

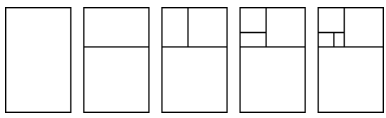
Getting Started. Part 1 shows how to get started with the Squeak environment. It explains the installation process and how to launch Squeak, and then presents robots and their behavior. A first simple program that draws some lines is presented.



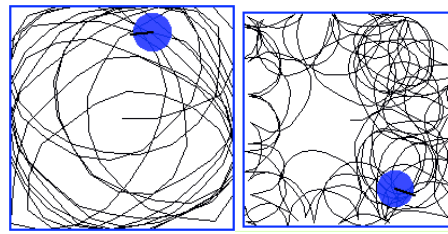
Elementary Programming Concepts.Part 2 introduces first programming concepts, such as loops and variables. It shows how messages sent to a robot are resolved.



Bringing Abstraction into Play. Part 3 introduces the necessity of abstraction, that is, methods or procedures that can be reused by different programs. The most difficult con- cept introduced is the idea of composing new methods from existing ones to solve more complex problems. Several nontrivial experiments are proposed, such as drawing golden rectangles. Techniques and tools for debugging programs are also introduced.



Conditionals. Part 4 introduces the notion of conditionals, conditional loops, and Boolean expressions, all of which are central to programming. This part also introduces the notion of references in a two-dimensional space and some other types of robot behavior. Finally, ways of using robots to simulate the behavior of simple animals are presented.



Other Squeak Worlds. Part 5 presents two entertaining programming environments that are available in Squeak: the eToy graphical scripting system and the 3D authoring envi- ronment Alice.

Why Squeak and Smalltalk?

You may be wondering why among the large number of programming languages available today I have chosen Smalltalk. Smalltalk and Squeak have been chosen for the following reasons:

- Smalltalk is a powerful language. You can build extremely complex systems within a language that is simple and uniform.
- Smalltalk was designed as a teaching language. It was influenced by Logo and LISP, and Smalltalk in turn heavily influenced languages such as Java and C#. However, those lan- guages are much too complex for a first exposure to programming. They have lost the beauty of Smalltalk's simplicity.
- Smalltalk is dynamically typed, and this makes transparent a number of concerns related to types and type coercion that are tedious to explain and of little interest to the novice.
- With Smalltalk you need to learn only key, essential concepts, concepts that are to be found in all programming languages. Thus with Smalltalk I can focus on explaining the important concepts without having to deal with difficult or unattractive aspects of more complex languages.

- Squeak is a powerful multimedia environment, so after reading my books you will be able to build your own programs in a truly rich context.
- Squeak is available without charge and runs on all of today's principal computing plat- forms. And it should be easily portable to the platforms of the future.
- Squeak is popular. For example, in Spain, it is used in schools, where it runs on over 80,000 computers.

Part I

Getting Started

Chapter 1

Installation and Creating a Robot

Set your stopwatch! Five minutes from now, the robot playground, called the environment, that you will be using in this book will be up and running and ready for you to have fun in. In this chapter you will learn how to install the environment, become acquainted with its different parts, and begin interacting with the robots that live in this environment. You will learn how to program these robots to accomplish challenging tasks by sending them messages. So let us get started installing the environment and preparing for all the challenges ahead in the rest of the book. If your environment is already installed, then turn off your stopwatch, skip the first section, and plunge directly into the following sections, which give an overview of the environment. After you have acquired some facility with robots in Chapters 2 through 4, I will go into more detail on using the environment in Chapter 5.

1.1 Installing the Environment

The environment used in this book has been developed to run on top of Squeak. Squeak is a rich and powerful Open Source multimedia environment written entirely in Smalltalk and freely available for most computer operating systems at http://www.squeak.org. Note, however, that you will not be using the default Squeak distribution. Rather, you will be using a distribution that I have prepared for use with this book. It can be downloaded from the publisher of this book at http://www.apress.com and , in the Downloads section.

Squeak runs exactly the same on all platforms. However, to make your life a little easier, I have prepared several platform-dependent compressed files. The principle is exactly the same on a Mac, PC, or any other platform. The only differences are in the tools for file decompression and the way that you will invoke Squeak. Once you have obtained a file named ReadyToUse.zip, you decompress it and then drag the file named Ready.image (Mac) or Ready (PC) onto the Squeak application, and that does it! The file Ready[.image] contains the complete environment used in this book. Note that you may get files with slightly different names, but that should have no effect on how everything works.

Installation on a Macintosh

For installation on a Macintosh, you should have a ZIP archive file named readyToUse.zip. Normally, double clicking on the file's icon should invoke the proper decompression software, such as StuffIt Expander. Once this archive has been decompressed, you should end up with four files, as shown in Figure 1.1. You should identify two files: the file named Ready .image and the *Squeak application* file (the one without a file extension in Figure 1.1; it is named Squeak).

Installation under Windows

For installation under Windows, you should have a ZIP archive file named readyToUse.zip. Once this archive has been decompressed using WinZip, you should end up with four files, as shown in Figure 1.2. You should identify two files: the file named Ready and the *Squeak application* file (the one without a file extension in Figure 1.2; it is named Squeak).

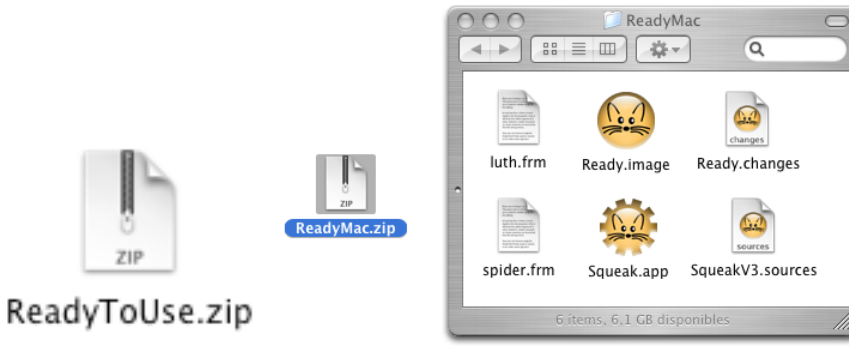


Figure 1.1: Ready-to-use files for the Macintosh. Left:the ZIP archive. Right:the decompressed files.

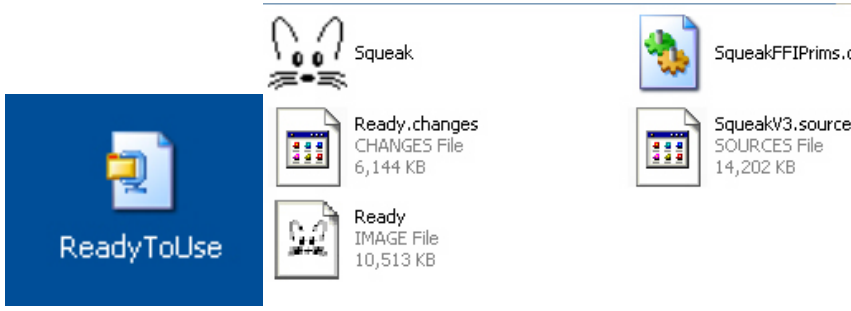


Figure 1.2: Ready-to-use files for Windows. Left:the ZIP archive. Right:the decompressed files.

1.2 Opening the Environment

To open the environment, drag the file Ready[.image]onto the Squeak application,that is, onto the file named Squeak, as shown in Figure 1.3. You should obtain the environment shown in Figure 1.4. If you do not get this environment, then read the section "Installation Trouble- shooting" near the end of this chapter.

Tips for Installation.

The environment can be opened simply by double clicking on the image file. However, there are several disadvantages to this: You may have to identify the Squeak application,and sometimes another application may

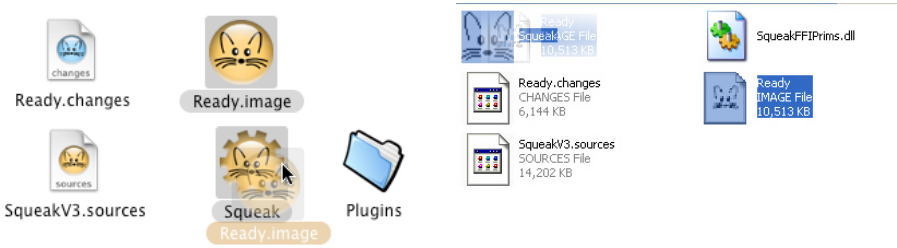


Figure 1.3: Dragging and dropping the *image* file onto the *Squeak application file* opens the environment on a Mac (left) or on PC (right).

interfere and try to use the image file. Moreover, you can find yourself in trouble if you have multiple installations of different versions of Squeak. So I suggest that you always open the environment by dragging and dropping the image file onto the Squeak application file or an alias of it.

Note that if you do not have enough space for the installation on your hard drive, you can use an alias to the SqueakV3.sources file, which can be shared among several installations.

Important! To start the environment, drag and drop the file Ready (with the .image extension for Mac) onto the squeak application.

1.3 First Interactions with a Robot

Once you have opened the environment by dragging the file named Ready[.image] onto the Squeak application as explained previously, the environment that you obtain should look something like the one presented in Figure 1.4.

The environment is composed of a robot factory and two flaps. A flap is a drawer contain- ing programming tools. You will not need these for a while, and so I will put off describing them until a later chapter. You should see a small blue robot in the middle of the screen. This is not a robot made of wires and metal, but a software robot, imagined as seen from above, pointing toward the right edge of the screen. A robot is a round blue circle; it has two wheels and a small red head that points in its current direction. As you work through this book, you will be sending orders to robots. These orders are called *messages*, and we say that the robots *execute* these messages.

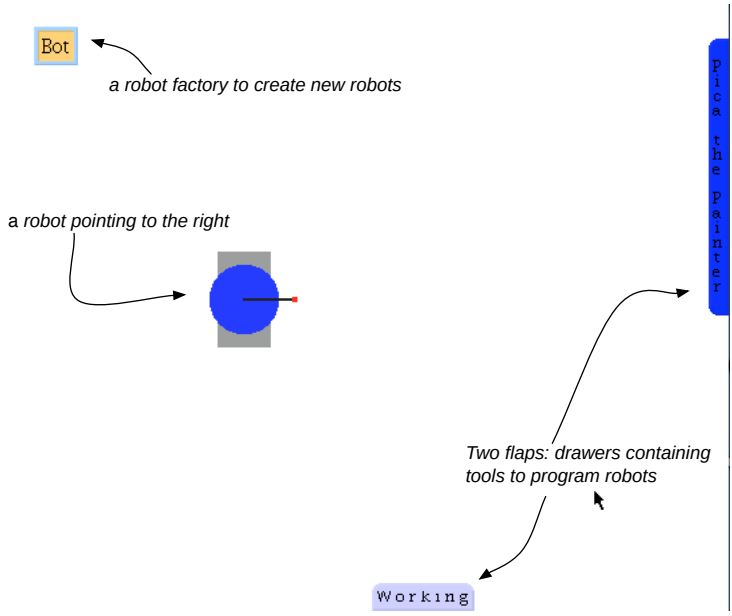


Figure 1.4: The environment is ready to use.

Place the mouse over the robot and wait a second. A balloon pops up with some information about the robot, such as its current location and its direction, as shown in Figure 1.5. Since computer monitors are of varying sizes and resolutions, your robot's position may have other values.

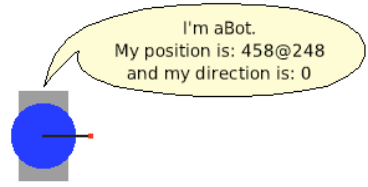


Figure 1.5: Place the mouse over a robot to pop up a balloon with information about the robot.

Sending Messages to a Robot

You can interact directly with a robot by left clicking on the robot with the mouse (or just clicking with a one-button mouse). A messaging balloon

pops up, as shown in the left picture in Figure 1.6. In this balloon you can type messages to be sent to the robot. After you type your messages, you send them to the robot by pressing the return key, and the robot then executes them.

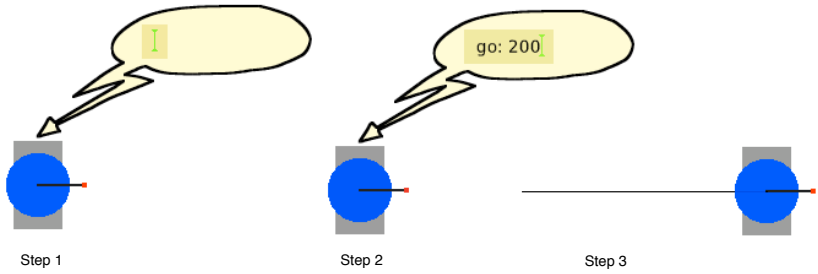


Figure 1.6: Step 1: Left-clicking on a robot causes a messaging balloon to appear. Step 2: You can type a message to the robot to move 200 pixels forward and then press the return key. Step 3: The robot executes the message; it has moved,leaving a trace on the screen behind it.

For example, if you type the message `go: 200` followed by the return key, you have told the robot to move forward 200 pixels in its current direction. If you type the message `turnLeft: 20 + 70`, you are instructing the robot to turn to its left (counterclockwise) 20 + 70 = 90 degrees, as shown in Figure 1.7. This second message is more complex than the previous one, because the value representing the number of degrees that the robot is to turn is itself a message (as I will soon explain), namely, `20 + 70`. We will call such messages *compound messages*.

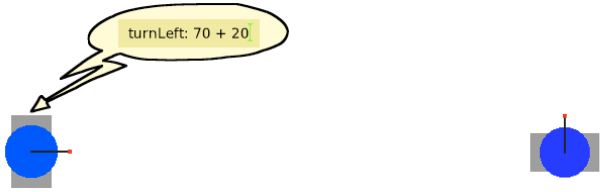


Figure 1.7: Left: sending a compound message. Right: The message has caused the robot to turn to its left by 90 degrees.

When the message `color: Color green` is sent to a robot, it changes its color, as shown in Figure 1.8. (You will have to imagine the green color in the grayscale picture.)

Figure 1.8: Left: Changing the color of a robot to another color. Right: its effect.

You may not understand the format of the messages that I have just presented. Some of them may appear a bit complex. In fact, color: Color green is another compound message. I will explain later how you can develop your own messages. For now, simply type the messages presented to you so that you can become familiar with the robot's environment. If you want to repeat a previous message, you do not have to retype it. Simply use the up and down arrows to navigate over the previous messages that you have sent to the robot. In subsequent chapters, you will learn step by step all the messages that a robot understands, and what is more, you will learn how to define new behaviors for your robots.

Important! To interact with a robot, click on it, type a message, and press the return key.

1.4 Creating a New Robot

The environment already contains a robot, but now I am going to show you how to create new robots. If you are not satisfied with having only one robot, you can create a new one by sending the appropriate message to a robot *factory*. A robot factory is graphically represented as an orange box surrounded by a light blue box, in the middle of which the word Bot is written, as shown in Figure 1.9. In Squeak jargon, and in general in the jargon of object-oriented programming, a robot factory is called a *class*. Classes (factories that produce objects, such as robots) have a name starting with an uppercase letter. Hence this is the class Bot and not bot.

Just as you did for robots, you can interact with a robot factory by sending it messages. The message to create a new robot is the message new, as shown in Figure 1.10. Note that newly created robots, like your original robot, point to the right of the screen. Each of the two robots has an

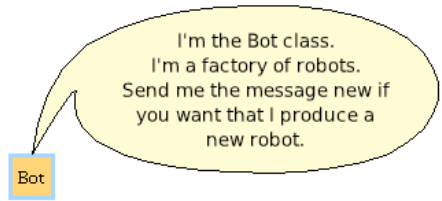


Figure 1.9: In Squeak jargon, a robot factory is called a class.Classes produce objects. The Bot class produces new robots.

independent existence, and you can send messages to each of them in turn.

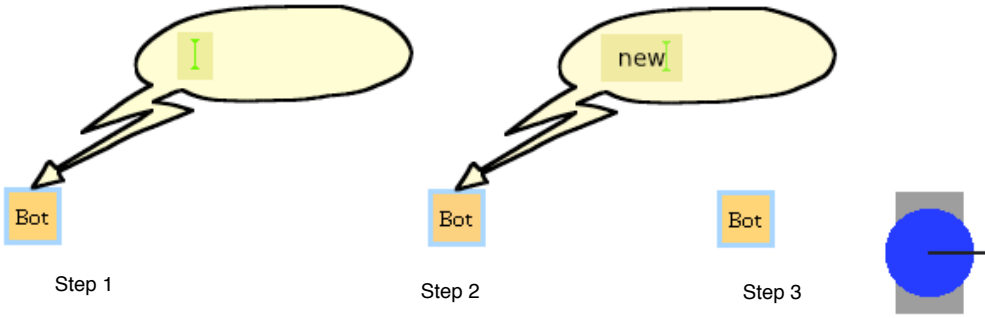


Figure 1.10: Step 1: Start typing a message. Step 2: The message new has been sent to the robot factory. Step 3: In response, the factory has created a robot and delivered it to you.

Important! To create a new robot, send the message *new* to the robot factory, which is the class Bot. When a robot is created, it is always pointing to the east, that is, to the right of the screen.

1.5 Quitting and Saving

The background of the Squeak window application is called the World. The World has a menu offering a number of different options. To display the

World menu, just (left) click on the back- ground. You should get a menu similar to the one shown in Figure 1.11. The last group of options consists of all the actions that you can take to quit out of the environment or save your work.

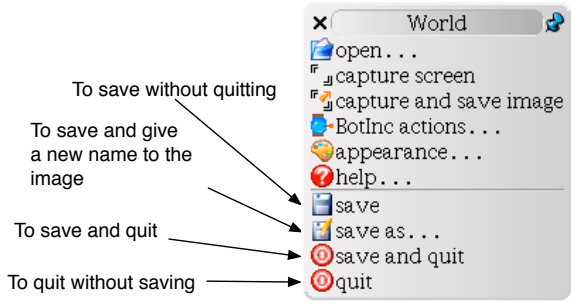


Figure 1.11: The World menu includes actions for quitting and saving.

Selecting the item `quit` simply quits the environment without saving your work. The result is that the next time you launch the environment, it will be in exactly the same state as the last time you saved it. Selecting the item `save` saves the complete environment. The next time you start the environment, it will be in exactly the same state as the last time you saved it. Finally, if you select the item `save as ...`, the environment asks you to create a new name, and it will then create two new files with that name: one with the extension .image and one with the extension .changes. That is how I created the files Ready[.image] and Ready.changes. To open the environment that you saved with a new name, drag and drop the file with the new name that has the extension .image onto the squeak application file icon as you did to start the environment by dragging and dropping the file Ready[.image].

Installation Troubleshooting

Sometimes things don't proceed just as they should, so in this section I will present some information that should be of help if you encounter problems during installation. First, I will explain the role of the principal files that you obtained when you decompressed the archive. To run the environment provided with this book or with any Squeak distribution, four files are necessary. Knowing about them can help in solving any problems you may encounter.

Image and changes. The file Ready[.image], called simply the image file, and the file Ready.changes, called simply the changes file, contain information about your current Squeak system. These two files are synchronized by Squeak automatically and should be writable (that is, not read-only). Each time you save your environment, these two files are synchronized. You should not edit them with a file editor or change the name of the file manually. If you want to use different names, just use the save as... menu item of the World menu. Squeak will then create a new pair of files for you.

Source. The file named SqueakV3.sources, called the *sources* file, contains the source code of a part of the Squeak environment. You will not need it in working through this book, so do not try to edit it manually. However, this file should always be in the same directory in which the image file is located.

Application. The application files Squeak for Mac and Squeak.exe for PC are the Squeak application. Each of these files is the application that runs when you are programming in Squeak. It should be executable. This file is referred to as the Squeak *application*. In computer-science jargon, this application is called a *virtual machine*, or VM for short.

Keep in mind that the image and changes files should be writable. Some operating systems change the properties of files to "read only" when they are copied from an external source. If that happens, Squeak warns you with a message, like that shown in Figure 1.12. If you get such a message, simply quit Squeak without saving, change the property of the file to permit write access, and restart.

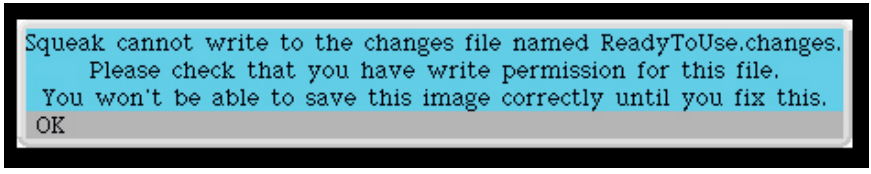


Figure 1.12: This message appears if the image (Ready.[image]) or changes (Ready.changes) file is not writable.

Another possible problem you may encounter is related to the sources file SqueakV3.sources. This file or an alias pointing to this file should be present in the directory in which the image file is located. If the file itself is not present, you may get the message shown in Figure 1.13. To cure this problem, create an alias to the sources file (SqueakV3.sources) in the directory containing the image file or simply copy the sources file into the directory

that contains the image file. You should not have this problem if you are using the distribution for this book.

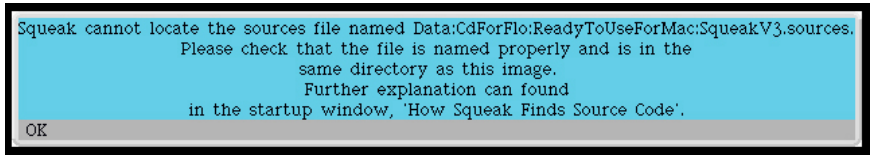


Figure 1.13: Possible messages indicating that the sources file (SqueakV3.sources) is missing from the directory containing the image file.

1.6 Summary

To start the environment, drag and drop the file Ready[.image] or another file that you have saved with the .image extension into the squeak application.

- To send a message to a robot, left click on it, type the message, and press the return key.
- To create a new robot, send the message new to the class Bot, which is your robot factory.
- When a robot is created, it is always pointing to the east, that is, to the right of the screen.
- To obtain the menu for saving the environment, click on the background.

Chapter 2

A First Script and Its Implications

While sending messages using direct interaction with a robot is a fun and powerful way of programming robots, it is rather limited as a technique for writing complex programs. To expand your programming horizons, I am going to teach you about the notion of a *script*, which is a sequence of *expressions*, together with all the fundamental concepts and vocabulary that you will need for the remainder of this book. It also serves as a map to subsequent chapters, which will introduce in depth the concepts briefly presented in this chapter.

First, I will show you how you to send multiple messages to the same robot by separating a sequence of messages with semicolons. Then you will learn how to write a script using a dedicated tool called a *workspace*. I will describe the different elements that compose a script and show some of the errors that one can make when writing a program.

2.1 Using a Cascade to Send Multiple Messages

Suppose you want to get your robot on the screen to draw a rectangle of height 200 pixels and width 100 pixels. To do so, you might click on your robot and then start to type the first message, go: 100, press the return key, then click on the robot and type the second expression, turnLeft: 90, and press the return key, then click on the robot and type the expression go: 200, and so on. You will quickly notice that this is truly a tedious way of interacting with your robot. It would be much more convenient if you could first type in all the instructions and then push a button to have the sequence of instructions executed.

In fact, you can send multiple messages to a robot by separating the messages with a semicolon character ;. To send a robot the messages go: 100, turnLeft: 90, and go: 200, simply separate them with a semicolon as follows: go: 100 ; turnLeft: 90 ; go: 200 (see Figure 2.1). This way of sending multiple messages to the same robot is called a *cascade of messages* in Squeak jargon.

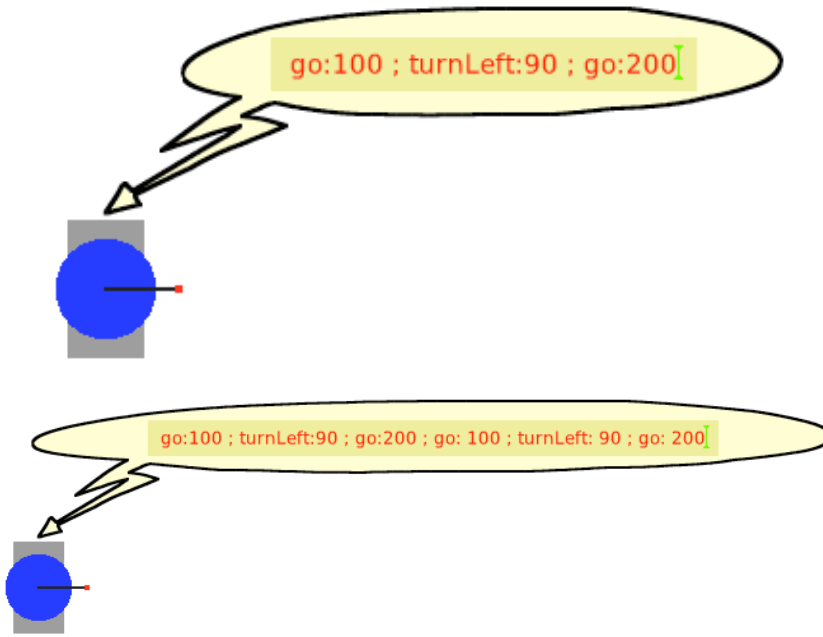


Figure 2.1: You can send several messages to a robot at once using the semicolon character (;).

However, the technique of writing a cascade of messages (that is, sending a robot multiple messages separated by semicolons) does not work

well for complex programs. Indeed, even for drawing a simple rectangle, the string of messages quickly grows too long, as shown by the second message in Figure 2.1). And there are other concerns as well. For example, programmers take into account issues such as whether they can store a sequence of messages and replay them later and whether they can reuse their messages and not have to type them in all the time. For all these reasons, we need other ways to program robots. The first way that you will learn is to write down a sequence of messages, called a *script*, in a text editor and ask the environment to execute your script.

2.2 A First Script

The BotInc environment provides a small text editor, called the Bot Workspace, which is dedi- cated to script execution (that is, executing the expressions that constitute a script). Click on the bottom flap, called Working. By default, it contains a Bot Workspace editor, as shown in Figure 2.2.

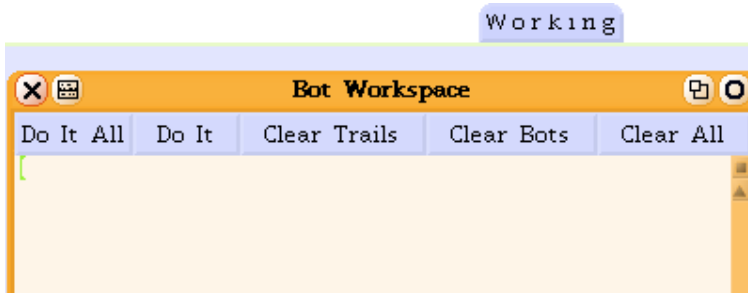


Figure 2.2: A Bot workspace is a small text editor dedicated to the execution of robot scripts.

I will start off by writing a script that draws a rectangle, and then I will explain it in detail (Script 2.1).

Script 2.1: *The robot pica is created and is made to move and turn.*

```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
```

```
pica turnLeft: 90
```

Figure 2.3 shows the script in a Bot workspace and the result of its execution obtained by pressing the Do It All button. Try to get the same result: type the script and press the Do It All button. I have named the robot pica as short for Picasso, since our robots are drawing pictures, just like those of the great Spanish artist.

The Do It All button of the Bot workspace executes *all* the messages that the workspace contains. Therefore, before typing a script, make sure that no other text is already present in the Bot workspace. Moreover, computers and programming languages cannot deal with even the most obvious mistakes, so be careful to type the text exactly as it is presented in Script 2.1.

For example, you must type the uppercase "B" of Bot on the second line, and you must end each line with a period. (There is no need to put a period at the end of the last line, because periods separate messages in Squeak. There is also no need for a period after the first line, because it doesn't contain a message.) But more on that a bit later in the chapter. The script and its result are shown in Figure 2.3.

2.3 Squeak and Smalltalk

Script 2.1 is admittedly simple, but nonetheless, it constitutes a genuine computer program. A *program* is a list of *expressions* that a computer can execute. To define programs we need programming languages, that is, languages that allow programmers to write instructions that a computer can "understand" and execute.

Programming Languages

A well-designed programming language serves to support programmers in expressing solutions to their problems. By support, I mean that the language should, among other things, facilitate expression of the task to be performed, provide efficient execution of the program code and reliability of the resulting application, give the programmer the ability to prove that programs are correct, encourage the production of readable code, and make it easy for programmers to make changes in their applications. There is no "best" or ideal programming language that satisfies all of these desirable properties, and different programming languages are best suited for different kinds of tasks.

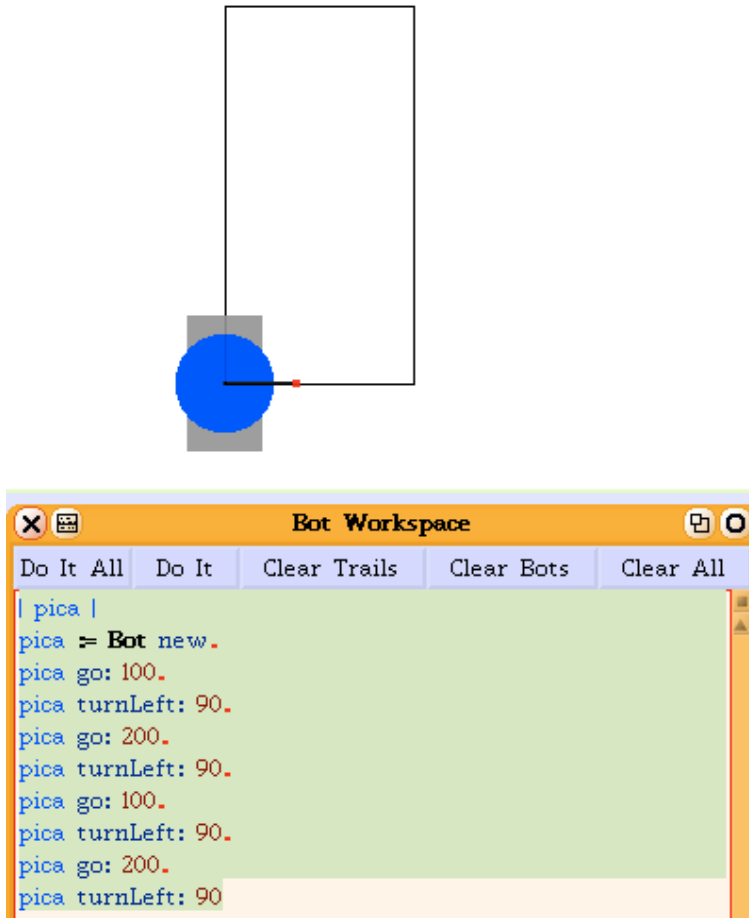


Figure 2.3: A script executed using the Do It All button of the Bot workspace and its result.

Smalltalk and Squeak

This book will teach you how to program in the Smalltalk *programming language* within the Squeak *programming environment*. A programming environment is a set of tools that programmers use to develop applications. Squeak contains a large number of useful tools: text editors, code browsers, a debugger, an object inspector, a compiler, widgets, and many others. And that's not all! In the Squeak environment you can program music, animate flash files, access the Internet, display 3D objects, and much more.

However, before you can start pro- gramming complex applications, you have to learn some basic principles, and that is the purpose of this book.

Squeak programmers develop their applications by writing programs using the programming language called Smalltalk. Smalltalk is an *object-oriented* programming language. Other object-oriented programming languages are Java and C++, but Smalltalk is the purest and simplest. As the term "object-oriented" suggests, such programming languages make use of objects. The objects that are created and used are, of course, not real objects, but logical structures, or "virtual" objects, within the computer. But they are called objects because it is useful to think of these structures as manufactured contraptions, such as a robot, for example, that are able to understand messages that are sent to them and to execute whatever instructions are contained in those messages. The point of the object analogy is that we can use a robot, or a radio, or a camera, without understanding its internal structure. We need only know how to use it by pushing its buttons or sending it messages via the remote control.

Where do manufactured objects come from? A factory, of course. The factories used to create objects are called classes in object-oriented programming languages. Defining classes is somewhat tricky, as is object-oriented programming in general, so in this introductory book I will not show you how to define classes. Instead, you will only define new types of behavior for your robot, and this will give you a good grounding in basic programming concepts.

I chose Smalltalk as the language for this book because it is simple, uniform, and pure. It is pure in that in Smalltalk, *everything* is an object that sends and receives messages to and from other objects. It is simple because in Smalltalk there are only a few basic rules, and it is uniform in that these rules are always applied consistently. In fact, Smalltalk was originally designed for teaching novices how to program. But that doesn't mean that Smalltalk can be used only for writing "baby" or "toy" applications. Indeed, large and complex applications have been written in Smalltalk, such as the applications controlling the machines that produce the AMD corporation's microprocessors that may be running in your computer.

Another application written entirely in Smalltalk is the Squeak environment itself. Now isn't that interesting! This means that once you develop a good understanding of Smalltalk, you can modify the Squeak environment in order to adapt the system to your own purposes or simply to learn more about the system. With Smalltalk, then, you have quite a bit of power in your hands.

I hope that this discussion about programming languages in general, and Smalltalk in particular, has motivated you to learn how to program. But please be aware that learning to program is like learning to play the

piano or to paint in oils. It is not simple, and so do not become discouraged if you have some difficulties. Just as a beginning piano player doesn't start off with Bach's Brandenburg concertos, and a novice painting student doesn't try to reproduce Michelangelo's Sistine Chapel ceiling, the beginning programmer starts off with simple tasks. I have designed this book so that topics are introduced in a logical order, so that what you learn in each chapter builds on your knowledge from previous chapters and prepares you for the material in the following chapters.

2.4 Programs, Expressions, and Messages

Now we are ready to take a closer look at your first script and explain just what is going on.

Typing and Executing Programs

When you wrote Script 2.1, you typed some text, constituting a sequence of expressions, and then you asked Squeak to execute it by pressing the Do it All button. Squeak executed the sequence of expressions; that is, it transformed the textual representation of your program into a form that is understandable by a computer, and then each expression was executed in sequence. In this first script, executing the sequence of expressions created a robot named pica, and then pica executed, one after another, the messages that were sent to it.

A program in Squeak consists of a sequence of *expressions* that are executed by the Squeak environment. In this book, such a sequence is called a *script*.

Important! A script is a sequence of expressions.

A program is a bit like a recipe for a chocolate cake. A good cake recipe describes all the steps to be carried out in correct sequence: cream the butter and sugar; melt the chocolate; add the chocolate to the butter and sugar mixture; sift in the flour; and so on right through placing the filled cake pans in a 350° oven, cooling the baked cake on a rack, and spreading on the frosting. Enjoy! Similarly, a computer program describes all the steps in sequence needed to produce a certain effect: declare a name for a robot; create a robot with that name; tell the robot to move 100 pixels; tell the robot to turn; and so on.

The Anatomy of a Script

The time has arrived to analyze your first script, which is copied here as Script 2.2.

Script 2.2: *A simple script yet a program*

```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 200.
pica turnLeft: 90
```

In a nutshell, Script 2.2 starts of by declaring that it will be using a *variable* named pica to refer to the robot it creates. Once the robot is created and associated with the variable pica, the script tells the robot to take a sequence of walks to different locations on the screen while turning 90 degrees to the left after each walk. Now let us analyze each line step by step. Don't worry if certain concepts such as the notion of a variable remain a bit fuzzy. Everything will be dealt with in due course, and if not in this chapter, then in a future chapter.

@@stef here@@

`| pica |` This first line declares a variable. It tells Squeak that we want to use the name pica to refer to an object. Think of it as saying to a friend, from now on I am going to use the word pica in my sentences to refer to the robot that I am about to order from the robot factory. You will learn more about variables in Chapter 8.

`pica := Bot new.` This line creates a new robot by sending the message newto the robot factory (class) named Botand associates the robot with the name pica, the variable that was declared in the previous step. The word Botrequires an uppercase letter B because it is a class, in this case the class that is a factory for producing robots.

`pica go: 100.` In this expression, the message go: 100 is sent to the robot we named pica. This line can be understood as follows: "pica, move 100 units across the computer monitor." It is implicit in this expression that a robot receiving a go: message knows in what direction to travel. In fact, a robot is always pointing in some direction, and when it receives a go: message, it knows to move in the direction in which it happens

to be point- ing. Note also that the message name go:terminates with a colon. This indicates that this message needs additional information, in this case a length. For example, go: 100 says that the robot should move 100 pixels. The message name is go:.

pica turnLeft: 90. This line tells pica to turn 90 degrees to its left (counter-clockwise). This line is again a message sent to the robot named pica. The message name turnLeft: ends with a colon, so additional information is required, this time an angle. The remaining lines of the script are similar.

Important! Important! Any message name that terminates with a colon indicates that the message needs addi- tional information,such as a length or an angle.For example,the message name turnLeft: requires a number representing the angle through which the robot is to turn counterclockwise.

About Pixels On a computer screen, the unit of distance is called a pixel. This word was invented in about 1970 and is short for "picture element." A pixel is the size of the smallest point that can be drawn on a computer screen. Depending on the type of computer monitor you are using, the actual size of a pixel can vary. You can see individual pixels by looking at the screen through a magnifying glass.

Expressions,Messages,and Methods

I have been using the terms expressionandmessage. And now it is time to define them. I will also define the important term method.

Expression Anexpressionis any meaningful element of a program. Here are some examples of expressions: • | pica |is an expression that declares a variable (more in Chapter 8). • pica := Bot newis an expression involving an operation, called assignment, that asso- ciates a value with a variable (see Chapter 8). Here, the newly created robot obtained by sending the message newto the class Botis associated with the variable pica. • pica go: 100is an expression that sends a message to an object. Such an expression is called a message send. The message go: 100is sent to the object named pica. • 100 + 200is also a message send. The message + 200is sent to the object 100. Message Amessageis a pair composed of a message name, also called a message selector, and possible messagearguments, which are the values that the object receiving the message needs for exe- cuting the message. These relationships are illustrated in Figure 2-4. The object receiving a

message is called a message receiver. A message together with the message receiver is called a message send. Here are some examples of messages: • In the expression pica beInvisible, the message beInvisibleis sent to a receiver, a robot. This message has no arguments. • In the expression pica go: 100, the message go: 100is sent to a receiver, a robot named pica. It is composed of the method selector go:and a single argument, the number 100. Here, 100represents the distance in pixels through which the robot should move. Note that the colon character is part of the message selector. • In the expression 33 between: 30 and: 50, the message between: 30 and: 50is composed of the method selector between:and:and two arguments, 30and50. This message asks the receiver, here the number 33, whether it is between two values, here the numbers 30 and50. • In the expression 4 timesRepeat: [pica go: 100], the message timesRepeat: [pica go: 100], which is sent to the number 4, is composed of the message selector timesRepeat:and the argument [pica go: 100]. This argument is called a block, which is a sequence of expressions (in this case a single expression) inside square brackets (more on this in Chapter 7). • In the expression 100 + 200, the message + 200is composed of the method selector + and an argument, the number 200. The receiver is the number 100.

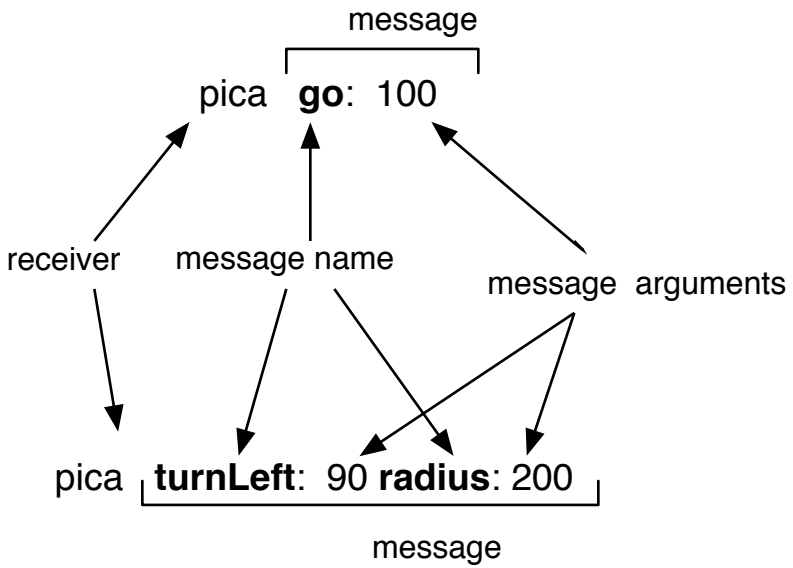


Figure 2.4: Two messages composed of a message name or method selector, and a set of arguments.

Message Separation

As mentioned earlier, each line of Script 2-1, except the first and last, is terminated by a period. The first line does not contain a message. Such a line is called a variable declarationin computer jargon. Thus, we can make the following observation: each message send must be separated from the following one by a period. Note that putting a period after the last message is possible but not mandatory. Smalltalk accepts both.

Important! Important! Message sends should be separated by a period.The last statement does not require a terminal period.Here are four message sends separated by three periods.

```
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100
```

Important! Important! A period character . is a message separator, so there is no need to place one after a message send if there is no following message send. Therefore,no period is necessary at the end of a script or of a block of messages.

Method

When a robot (or other object) receives a message, it executes a method, which is a kind of script that has a name. More formally, a method is a named sequence of expressions that a receiving object executes in response to the receipt of a message. A method is executed when an object receives a message of the same name as one of its methods. For example, a robot executes its methodgo:when it receives a messagewhose name is go:. Thus the expression pica go: 224causes the message receiver picato execute its method go:with argument 224, resulting in its moving 224pixels forward in its current direction. Later in the book, I will explain how you can define new methods for your robot, but for now, we do not need them to start programming.

Cascade

As I mentioned in the first section of this chapter, you can send multiple messages to a robot by separating them with semicolons. Such a sequence of messages is called a cascade. You can also use a cascade in a script to send multiple messages to a robot. Script 2-3 is equivalent to Script 2-2, except that now all the messages sent to the robot picaare separated by semicolons. Using cascades is handy when you want to avoid typing over and over the name of the receiver of the multiple messages. Cascades are useful because they shorten scripts. However, be careful! Shortcuts can lead to trouble if you don't watch your step, so be sure that you truly intend for all your messages to be sent to one and the same receiver.

```
| pica |
pica := Bot new.
pica
go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90 ;
go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90.
```

Important! Important! To send multiple messages to a robot,use a semicolon character ; to separate the messages,following the pattern aBot message1 ; message2.Here is an example:pica go: 100 ; turnLeft: 90 ;go: 200 ; turnLeft: 90

Creating New Robots

To obtain a new robot, you have to send an order to the robot factory to manufacture one for you. That is, you have to send the message newto the class Bot. There is nothing new here. It is exactly what you did in the previous chapter when you clicked on the blue and orange box namedBot, which represents the class of the same name, and typed newin the bubble. In Squeak, we always send messages to robots, other objects, or classes to interact with them. There is no difference in treatment, except that classes and objects understand different messages. It is the job of classes to create objects. An object does not know how to create other objects, so sending a robot the message newleads to an error. Classes, on the other hand, generally do not have colors and do not know how to move, and so sending the message coloror go: 135to a class does not make sense, and doing so leads to an error. Nonetheless, in both cases you are sending messages!

TheBotclass is not the only manufacturing company in the Squeak environment. There are other classes, and they understand different messages

and employ different methods for creating different kinds of objects. For example, the class Colormanufactures color objects. It returns a blue or green color object in response to the message blueorgreen. Whenever in this book a new object must be obtained from a specific class, I will tell you how it is done.

Important! Important! To obtain a new object from a class,you generally send the message newto the class.Thus Bot newcreates a new robot.Other classes may offer different messages for obtaining new objects.For example,Color bluetells the class Colorto create a new bluecolor object.

2.5 Errors in Programs

Computers are very good at making highly complex calculations at incredible speed, but they lack the intelligence to correct small mistakes. If I accidentally wrote, "now turn on your computer," you might chuckle over my misspelling, but you would have no trouble under- standing what I meant. But computers have no such intelligence, which means that each expression given to a computer must be given precisely, without the least error. The smallest seemingly insignificant mistake in a program, even something as trivial as using a lowercase letter instead of an uppercase one, will almost certainly be misunderstood by the computer. If you have errors in your scripts, two things can go wrong: either an error message will appear on the screen, and this is likely to occur when you are doing your first experiments, or the pro- gram will be executed, but the result will not be what you intended. So when things go wrong, do not despair and try to find the error in your program.

Squeak has a helpful error-prevention and error-correction facility. It colors the letters while you are typing. When a word becomes red, this means that you are writing something that Squeak does not understand. An example is shown in Figure 2-5. When a word is blue, for a variable or a message, or black, for a class, this indicates that everything is structurally correct.

If you attempt to execute an expression containing an error, Squeak tries to help you by notifying you when it encounters the error in your code. The error messages that Squeak uses are actually menus. The top part of the menu window contains a short description of the error; then, depending on the type of error, some suggested corrections may be listed as options. If you don't like any of the options, you can always cancel execution by

choosing "cancel" in the menu. Then you should locate the place in your script that Squeak did not understand, correct it, and try again to execute the script. I will now tell you about some of the most common errors.

Misspelling a Message Selector

Misspelling the name of a message leads to an error. In Figure 2-5, I misspelled the message selectorgo:, typing god:instead. The message god:does not exist in Squeak, and therefore Squeak turned the word red. Ignoring Squeak's friendly warning, I tried to execute the script. Squeak tried to guess what message selector I had in mind, and prompted me with a menu of possibilities. At this point, I could choose the correct message selector (go:), and the message god:will be replaced by go:. Or I can simply choose "cancel." If I take the latter option, I will have to change god:togo:manually.

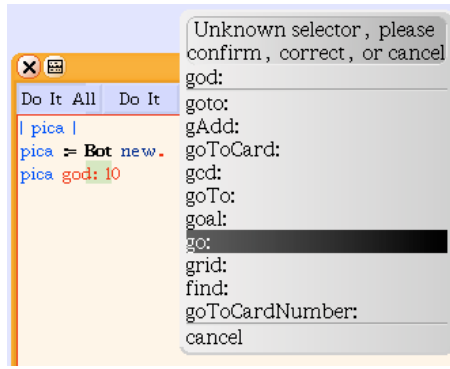


Figure 2.5: We mispell the message go: and type god: instead. The message god: does not exist therefore Squeak prompts us.

Misspelling a Variable Name

There are two ways to misspell the name of a variable: in the body of the script itself and when it is declared (between two vertical bars as in | pica |). Figure 2-6 shows the two cases: In the left-hand figure I declared a variable pica, but then I typed pica1 instead of pica in the script. Squeak noticed that I was trying to use an undeclared variable, so it turned the text red and prompted me, suggesting that I either declare the undeclared variable by declaring pica1 as a new variable, or replace pica1 by pica. Since pica is the variable name that I wanted, and pica1was just a typo, I chose the option pica, as shown in the figure. The right-hand figure shows that I

accidentally typed a space between the candainpicawhen I attempted to declare the variable pica. Squeak did not consider this an error. It simply "thought" that I was trying to declare two variables, picanda. Then in the script I typed pica, thinking that I had declared that variable. But Squeak saw that in fact, picawas an undeclared variable, so it turned the text red and gave me some options, including declaring a new variable with the namepicaor else replacing what I had typed with the declared variable pic.

Unused Variables

It may happen that you accidentally declare too many variables. For example, you might declare the variables picaanddaly, thinking that you will need two robots, but then you never usedalyin your script. This is not really an error, and your program will run correctly even if it has declared variables that are ever used. It is analogous to buying two suitcases, just in case, but using only one of them. You simply have some extra baggage around that you are not using. But just in case you really did mean to use dalyand forgot, Squeak checks for unused declared variables and if it finds any, suggests that you might want to remove them. For exam- ple, in Figure 2-7, the script declares the variables picaanddalybut uses only pica. Squeak notices this and asks you whether you would like to remove the unused variable daly.

Figure 2.6: All the variables and messages are correct. However the variable daly is defined but not used so Squeak indicates it to us. Unused variables are not a problem so you can proceed.

Uppercase or Lowercase?

Another common mistake is to forget a required uppercase letter. Names of classes begin with an uppercase letter, so don't forget this when you want to send a message to an object factory. Figure 2-8 shows that I unthinkingly

typed botinstead of Bot. Squeak tried to figure out what I meant, but it failed, and so none of the options that it offered for fixing the problem will do. In such a case you have to correct the error yourself. In the context of this book, the only classes you have to worry about are Bot, the robot factory, and Color, the color factory.

Forgetting a Period

Finally, one of the most common mistakes, one that even fluent programmers make, is to for- get a period between two message sends or a semicolon between two messages in a cascade. A period indicates that a new message send is about to begin, but without the period, Squeak thinks that the current message is being continued, and that the variable meant to be the message receiver of a new message is just another message selector. Since there is no message selector with the name of one of your variables, Squeak tells you that you have typed an unknown selector and offers you some possible corrections. For example, in Figure 2-9, a period is missing after the expression pica := Bot new, and Squeak tries to parse (that is, figure out the structure of) the message pica := Bot new pica go: 120, and according to the rules of message syntax (structure), about which you will learn in Chapter 11, picashould be a message selector. But such a message selector does not exist, so Squeak protests and proposes some possible replacements. Since you know that picais your declared variable and not a message selector, you realize that you forgot a period and so you select "cancel" and type the period manually.

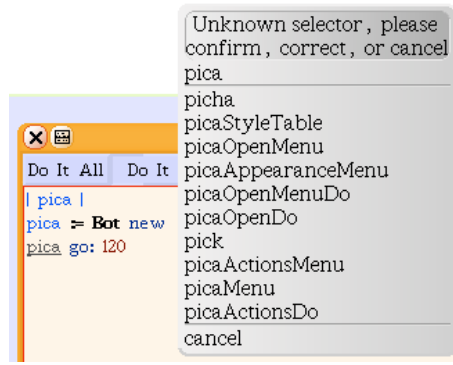


Figure 2.7: Two examples of error. We mistyped the names of the variable. First in the script pica1 has not been declared and second while defining the variable: car o defines two variables car and o but not a variable pica.

Words That Change Color

Squeak tries to identify mistakes while you are typing your scripts. If it detects something fishy, it changes the color of the text and provides some visual cues that suggest what might be wrong. Figure 2-10 shows some typical situations. Unfortunately, the black-and-white figure does not show its true colors. But use your imagination!

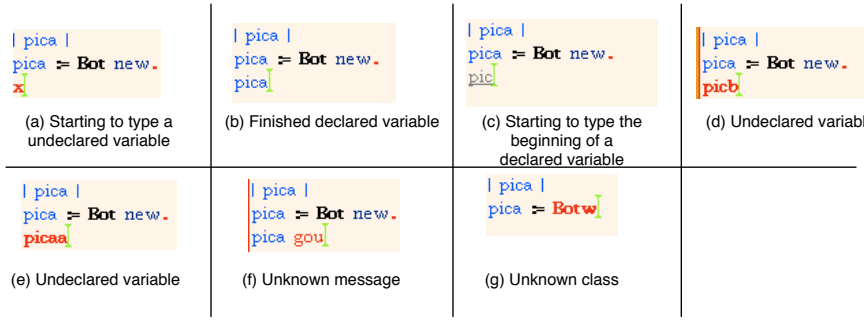


Figure 2.8: Squeak using colors to help finding mistakes.

Here is a key to the figure: (a) I started to type the first letter of an undeclared or unknown variable. Since no variable starting with the letter x has been declared, Squeak turns the x red, letting me know that something is amiss. (b) I finished typing a variable that has been declared. Squeak shows me that I have typed a declared variable correctly by turning the text blue. (c) I am in the process of typing the name of a variable. As long as what I have typed is the beginning of the name of a declared variable, Squeak underlines it to let me know that so far, everything is ok. (d) As soon as I type a character in a variable name that results in a sequence of letters that is not the beginning of the name of a declared variable, Squeak turns the word red. Note the difference with the previous case. In case (c), I could have typed the char- acteraand thereby completed the declared variable pica, as in (a). However, I typed the character b, and ended up with a sequence of letters (picb) that is not the begin- ning of the name of any declared variable. (e) After I typed the name of a declared variable (pica, as in case (b)), I accidentally added an extra character a, which leads to the sequence of letters (picaa), which is not the beginning of the name of a declared variable. (f) Squeak tries to do the same for message selectors as it does for variable names. Here I mistyped the message go:and typed instead gou. Squeak was looking for a message selector, and as soon as I typed the character u, it realized that there is no message selector that

begins gou, so it turned the text red.

(g) Squeak tries to do the same for classes as it does for variables and message selectors. Here I typed the character wafterBot, and Squeak, expecting a class name because of the uppercase B in Botw, indicates by turning the text red that there is no class in the system whose name begins Botw.

2.6 Summary

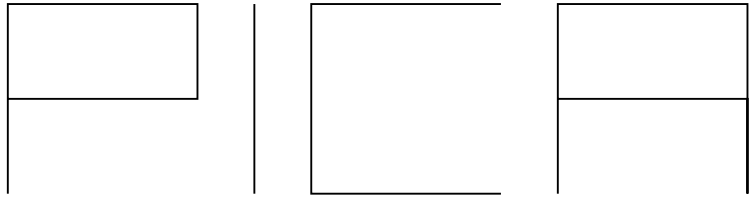
- To execute an expression. Press the Do It All buttonof the Workspace.
- A script is a sequence of expressions that performs a task.
- A message is composed of a message selector and possibly one or more arguments. Some message selectors do not take arguments, as in the message send pica beInvisible.
- Any message selector that ends with a colon requires additional information (one or more arguments), such as a length or an angle. For example, the message selector turnLeft:requires an argument whose value is a number representing the angle through which the robot should turn counterclockwise.
- To obtain a new object, you generally send the message newto a class. For example, Bot newcreates a new robot. Other classes may understand different messages for produc- ing new objects. For example, Color yellowasks the class Colorto create a new yellow color object.
- A class is a factory for producing objects. Class names always start with an uppercase letter. For example, Botis the factory for creating new robots, and Coloris the color factory. The message Bot new color: Color yellowasks the Bot class to create a new robot, and then the color factory is asked to create a yellow color object. Finally, the messagecolor:is sent to the new robot with the yellow color object as argument, resulting in the new robot having its color changed to yellow.
- Message sends should be separated by a period. A terminal period after the last mes- sage send is not required. Here is an example of four message sends separated by three periods:

```
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100
```

- To send multiple messages to the same object use a semicolon to separate the mes- sages, as in aBot message1; message2. For example, pica go: 100 ; turnLeft: 90 ; go: 200 ; turnLeft: 90send the sequence of four messages (1) go: 100, (2)turnLeft: 90, (3)go: 200, (4)turnLeft: 90 to the robot named pica.

Chapter 3

Of Robots and Men



In this chapter I describe the creation of robots and the different types of movements that robots know about and are capable of performing. I offer some simple experiments for you to perform, so that you can practice what you have learned in the previous chapters. I also will show you how robots can change direction along the fixed, or *absolute*, points of the compass.

3.1 Creating Robots

In the previous chapter you created *a* robot, not *the* robot. That is, robots are not unique, and you can create as many robots as you want. Script 3.1 creates two robots: pica and daly.

Script 3.1: *Two robots are born.*

```
| pica daly |
pica := Bot new.
daly := Bot new.
pica color: Color yellow.
daly jump: 100.
```

The second line creates a robot named pica as in Script 2.1. The third line creates a new robot that we refer to using the variable daly. (Just as pica's name is in homage to Pablo Picasso, that of daly is to honor Salvador Dali.) Both robots are created at the same location on the screen. In line four, we tell pica to change its color to yellow so that we can distinguish the two robots.

Smalltalk is an object-oriented programming language, as I have mentioned. This means not only that we can create objects and interact with them, but that objects can create other objects and communicate with them. Moreover, in Smalltalk, there are special objects, called classes, that are used to create objects. Sending the message `new` to a class creates an object described by its class. Sending the message `new` to the `Bot` class creates a robot.

To understand what classes are, imagine a class as a sort of factory. A factory for creating tin boxes might turn out large numbers of generic boxes, all of the same size, color, and shape. After they have been manufactured, some boxes might be filled with biscuits, while others might be crushed. When one box is crushed, other boxes are not affected. The same holds for objects created inside Squeak. In our case, daly did not change color, but pica did, while pica did not move, but daly did. You can think of a class as a factory able to produce unlimited sup- plies of objects of the same type. Once produced, each object exists independently of the others and can be modified as one wishes.

In Smalltalk, class names always begin with an uppercase letter. That's why the name of the robot class is `Bot` with an uppercase "B." Notice that in the command Color yellow, the word `Color` is written with an uppercase "C." That is because `Color` is a class, and what it man- ufactures is color objects. By specifying the color name, you get an object of the color you want. (The expression `Color yellow` is actually a short form for creating a

yellow color object. First, a color object is created by sending the message new to the class Color, and then some extra messages define the color to be yellow.)

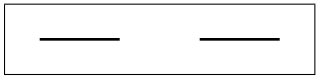
Important! A class is a factory that manufactures objects. Sending the message new to a class creates an object of that class. Class names always start with an uppercase letter. Here Bot is the name of the factory for creating new robots, and Color is the factory for colors.

Thus the command Bot new color: Color blue sends a message to the Bot class to create a new robot and then sends a message to the new robot to color itself with the color blue.

3.2 Drawing Line Segments

Asking a robot to draw a line is rather simple, as you already saw in the previous chapter. The message go: 100 tells a robot to move ahead 100 pixels, and the robot leaves a trace during its move. However, when you draw, even if you are an expert Chinese or Japanese calligrapher, you need to lift the brush from time to time. For this purpose, a robot knows how to jump; that is, a robot can move without leaving a trace. A robot understands the message jump:, whose argument is the same as that for go:; namely, it is a distance, given in pixels. Script 3.2 draws two segments. To keep the picture uncluttered, I have kept the robots out of the illustration using the message beInvisible.

Script 3.2: *Pica is created and then draws two lines.*



```
| pica |
pica := Bot new.
pica go: 30.
pica jump: 30.
pica go: 30.
```

Experiment 3.3: *Creating and Moving a Robot*

Experiment by changing the values in the previous script.

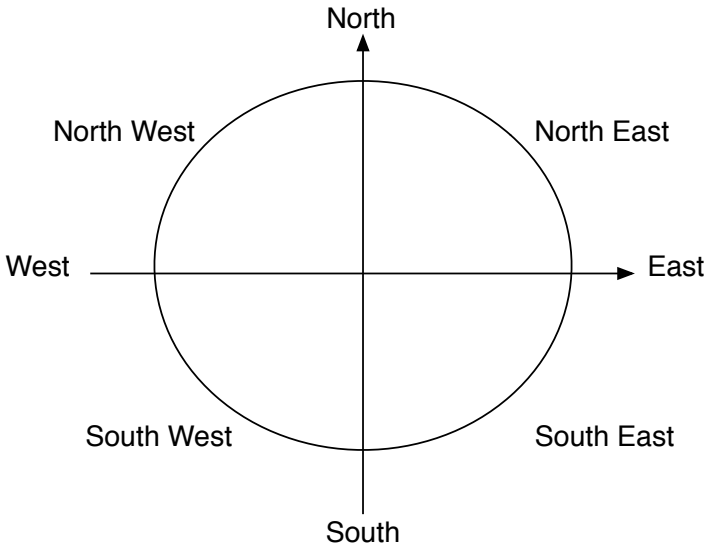


Figure 3.1: The default absolute directions of the compass in which a robot can point.

Experiment 3.4: *SOS*

Write a script that draws the message *"SOS"* in Morse code. (In Morse code,an *"S"* is represented by three short lines, and an *"O"* is represented by three long lines, as shown in figure below.

3.3 Changing Directions

A robot can orient itself along the eight principal directions of the compass, as shown in Figure 3.1. The directions are like those on a standard map: east is to the right, west to the left, north up, and south down. These directions are absolute, which means that regardless of the direction in which a robot is currently pointing, if you tell it to point east, the robot will point to the screen's right, not to the robot's right. To point a robot in a given absolute direction, just send it a message with the name of the direction. Thus, to tell pica to face south, you simply type pica south.

Robots understand the following compass direction messages: east, north, northEast, northWest, south, southEast, southWest, and west. In the next chapter, I will show you how to make a robot turn relative to its current position through an arbitrary angle. Script 3.5 illustrates the four cardinal directions with four different robots; here Picasso and Dali are joined by Paul Klee and Alfred Sisley. Except for pica, who remains in the default direction east in which it was created, each robot is oriented in a different direction before being told to move.

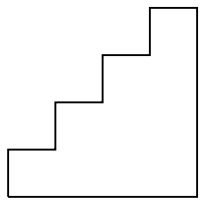
Script 3.5: *A gaggle of robots go walking.*

```
| pica daly klee sisl |
pica := Bot new.
pica color: Color green.
pica go: 100.
daly := Bot new.
daly north.
daly color: Color yellow.
daly go: 100.
klee := Bot new.
klee west.
klee color: Color red.
klee go: 100.
sisl := Bot new.
sisl south.
sisl go: 100.
```

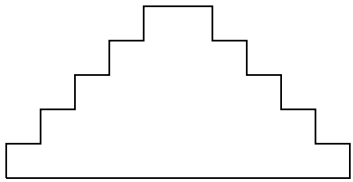
You can use these orientation methods to make more complex drawings.

Experiment 3.1 (*A square)*
As a first exercise,draw a square with sides of length 50 pixels. Then draw another square of side length 250 pixels.

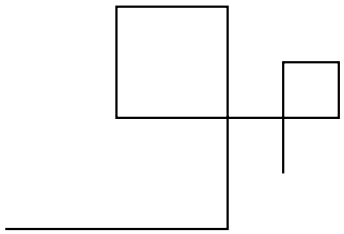
Experiment 3.2
A Staircase You are not limited in your robot drawings to squares. You can create a wide range of geometrical figures. For example, here is a drawing of a small staircase. Write a script to reproduce this drawing.



Experiment 3.3
The Step Pyramid of Saqqara Now you are ready to spread your architectural wings and draw a schematic side view of the step pyramid of Saqqara,built around 2900 B.C.E. by the architect Imhotep. Write a script to draw a side view of this pyramid, as shown in the figure. The pyramid has four terraces,and its top is twice as large as each terrace.



Experiment 3.4
Abstract Art Write a script to draw the picture shown in the figure below.

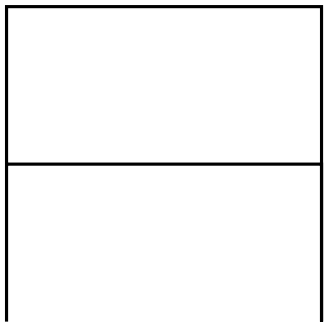


3.4 The ABC of Drawing

Even though you don't yet have much control over the direction of a robot's line segments, you can start programming pica to write letters. Script 3.1 draws a rather primitive letter "A."

Script 3.1 (*The letter A)*

```
| pica |
pica := Bot new.
pica north.
pica go: 100.
pica east.
pica go: 100.
pica south.
pica go: 100.
pica north.
pica go: 50.
pica west.
pica go: 100
```



Drawing a letter “C” is no more difficult. You can even write a script to spell out “pica.”

Experiment 3.5 (*Pica)*
Draw the name "PICA" as shown at the start of the chapter. To separate the individual letters, you should use the command jump:.

Note One could argue that Script 3-4 could be improved. For example,the bottom half of the right- hand vertical line of the "A" is drawn twice,since the robot goes back over this segment—once going south, once going north—in order to get into position to draw the horizontal bar. Deciding on the best approach to solving a programming problem can be a difficult proposition. There are many issues to be considered,such as speed,complexity,and readability of the code,and these questions will have different answers depending on the programming language and the methods used. However,one approach you might consider is to start off by choosing the simplest solution. Then if you are dissatisfied because the program is too slow or doesn’t have the particular bells and whistles you want,you can always modify it to speed it up or add other enhancements.

3.5 Controlling Robot Visibility

You can control whether a robot is to be displayed using the messages beInvisible and beVisible. The message beInvisible hides the receiver of the

message. A hidden robot acts exactly like a normal one; it just doesn't show where it is. Be careful not to use the method hide, which is defined by Squeak for its own purposes and can damage the robot environment if used improperly. The message beVisible makes the robot receiving the message visible. A newly created robot is visible by default.

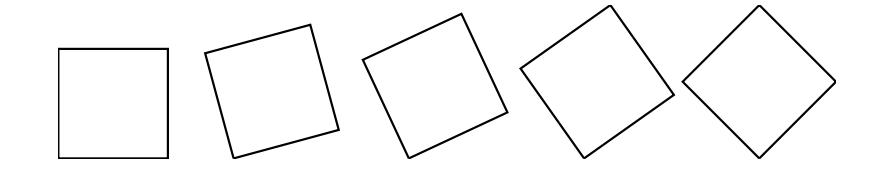
3.6 Summary

The following table summarizes the expressions and messages encountered in this chapter.

Expressions / Messages	**Description**	**Example**
Bot new	Create a robot.	pica := Bot new
| x y |	Declare variables to be used in the *script*	| pica |
jump: *anInteger*	Tell a robot to move forward a given number of pixels without leaving a trace.	pica jump: 10
go: *unEnter*	Tell a robot to move forward a given number of pixels while leaving a trace.	pica go: 10
beInvisible	Diu al robot que s'ha de tornar invisible	pica beInvisible
beVisible	Diu al robot que s'ha de tornar visible	pica beVisible
east, northEast, north, northWest, west, southWest, south, southEast.	Tell a robot to point in the given direction.	pica north
Color *colorName*	Create the color *colorName*	Color blue
color: *aColor*	Ask a robot to change its color.	pica color: Color red

Chapter 4

Directions and Angles



By now, you should be getting tired of drawing figures only in fixed directions. In this chapter you will learn how to change the direction in which a robot points, allowing the robot to point inanydirection, to turn through any angle relative to its current position, and therefore to draw lines in any direction. If you already understand clearly what an angle is and how to measure angles in degrees, you may skip the section “The Right Angle of Things” and then proceed to the examples and experiments in the section “Simple Drawings.” I will begin by presenting the elementary messages for changing direction that robots understand. I am going to hide the robots from the illustrations using the message beInvisible so that you can get clearer pictures.

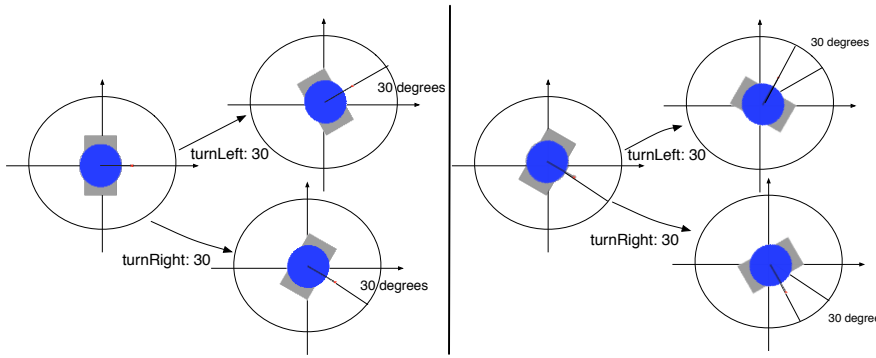


Figure 4.1: Left: A robot facing east turns left or right through 30 degrees. Right:A robot facing in some other direction turns left or right through 30 degrees.

4.1 Right or Left?

In the previous chapter, you learned that a robot can be made to face in different directions using the messages east, north, northEast, northWest, south, southEast, southWest, and west. However, with these messages you cannot change the direction of your robot through an arbitrary angle, such as 15 degrees. In addition, you cannot turn a robot through, say, a quarter turn relative to its current direction.

To turn a robot through a given angle you should use the two methods turnLeft: and turnRight:, which tell a robot to turn to the left or the right. As the colon at the end of each method name indicates, these two methods expect an argument. This argument is the angle through which the robot should turn relative to its current position. That is, the argument is the difference between the robot's direction before the message is sent and its direction after the message is sent. This angle is given in degrees. For example the expression pica turnLeft: 15 asks pica to turn to the left fifteen degrees from its current direction, and pica turnRight: 30 turns pica to the right thirty degrees from its current direction. Figure 4.1 illustrates the effect of the messages turnLeft: and turnRight:, first when a robot is pointing to the east, and second when a robot is pointing in some other direction.

As you practice turning robots through various angles, keep in mind that when a new robot is created, it always points to the east, that is, to the right of the screen.

Experiment 4.1
Mystery Scripts Scripts 4.1 and 4.2 present problems in which you are to guess what the created robot will do. After studying these two scripts, experiment with them by changing the angle values, for example to determine what angle turns the robot through a quarter circle, a half circle, or a full circle. If you need to review the notion of angle, read the section "The Right Angle of Things" before continuing.

Script 4.1: *What does pica do? (Problem 1)*

```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 45.
pica go: 50.
pica turnLeft: 45.
pica go: 100
```

Script 4.2: *What does pica do? (Problem 2)*

```
| pica |
pica := Bot new.
pica go: 100.
pica turnRight: 60.
pica go: 100.
pica turnLeft: 60.
pica go: 100
```

4.2 A Directional Convention

In mathematics, it is a general convention that rotation through a negative angle is construed as clockwise, while one with a positive angle is in the counterclockwise direction. You can also make use of this mathematical convention by using the message turn:. Hence, the message turnLeft: aNumber is equivalent to the message turn: aNumber, while the message turnRight: aNumber is equivalent to turn: –aNumber, where –aNumber is the negative of aNumber. This relationship is depicted in Figure 4.2.

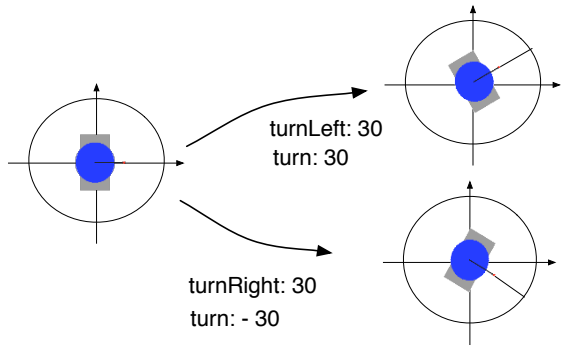
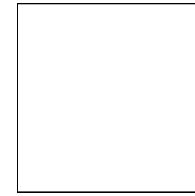


Figure 4.2: Turning through a 30-degree angle starting from the direction east.

4.3 Absolute Versus Relative Orientation

You should now feel confident that you can ask a robot to execute any drawing consisting of straight lines. Before going further, be certain that you understand the difference between orienting a robot absolutelyusing the methods north, south, southEast, east, etc., and using the methods turn:, turnLeft:, and turnRight: to orient the robot relative to its current orientation. Experiments 4.2, 4.3, and 4.4 will help you to solidify your understanding of this difference.

Experiment 4.2
A relative square Write a script to draw a square using the method turnLeft: or turnRight:.

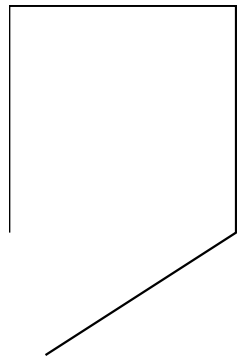


Experiment 4.3 (*Tilting the Square)*
Modify your script from Experiment 4.2 by adding the line pica turnLeft: 33. before the first line containing the message go: 100. You will obtain a square again, but it is tilted 33 degrees from the previous one.

Experiment 4.4 (*A broken square)*
Finally, execute Script 4.1, which attempts to draw a tilted square using the methods north, south, east, and west that we presented in the previous chapter.

Script 4.1 (*A Broken Square)*

```
| pica |
pica := Bot new.
pica north.
pica go: 100.
pica east.
pica go: 100.
pica south.
pica go: 100.
pica north.
pica go: 50.
pica west.
pica go: 100
```



Do you still obtain a square? No! The first side drawn by the robot is slanted, whereas the other sides are either horizontal or vertical. The script that you wrote for Experiment 4.4 and Script 4.1 demonstrate the crucial difference between *relative* and *absolute* changes in direction:

- The methods north, south, east, and west change direction in an absolute manner. The direction in which the robot will point *does not depend* on the current direction in which it is pointing.

- The methods turnLeft:and turnRight: change direction in a relative manner. The direction in which the robot will point *depends* on its current direction.

Figure 4.3 shows the equivalence between relative moves starting with a robot pointing to the east and absolute moves. As you know, this equivalence is valid only if the robot is point- ing east and not if it is pointing in any other direction. By the way, note that turning the robot 180 degrees points it in the opposite direction; this trick is often used in scripts.

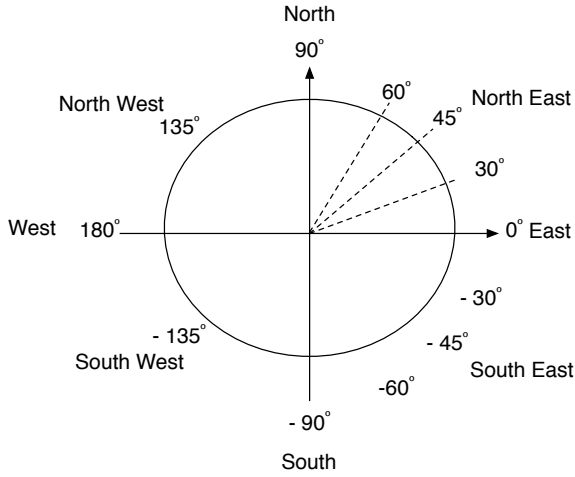


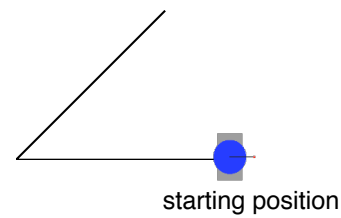
Figure 4.3: Comparing absolutes and relative angles starting from the east direction.

4.4 The Right Angle of Things

As you know by now, a newly created robot is pointing east, that is, toward the right-hand side of the screen. If we ask this robot to turn left by 90 degrees, it will end up heading north. If instead, we ask it to turn right by 90 degrees, it will end up heading south. Script 4-4 illustrates the result of a turn left by 45 degrees. To help you in following the script, the accompanying figure shows the robot's starting position.

Script 4.2 (*Moving through angles (1)*)

```
| pica |
pica := Bot new.
pica west.
pica go: 100.
pica east.
pica turnLeft: 45.
pica go: 100.
```



The first part of Script 4.2, up to the line pica east, draws a horizontal line, which will act as a reference line to indicate the easterly direction. The last part draws a line in the direction 45 degrees to the left of the easterly direction. You can vary the value of the angle to see what sort of angles

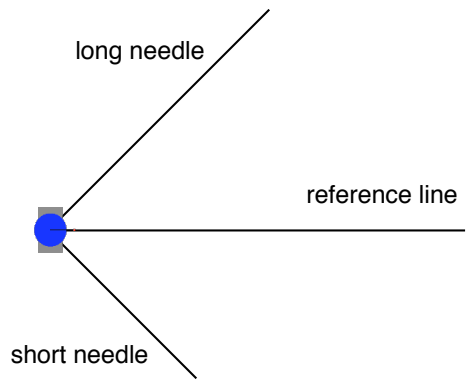
other numbers of degrees represent. Try the values 60, 120, 180, 240, 360, and 420. In particular, note that a turn by 180 degrees amounts to turning the robot in the opposite direction from which it is pointing.

Do you see any difference between arguments of 60 and 420? They represent the same angle! Any two angle values whose difference is 360 or any multiple thereof are equivalent because 360 degrees represents a complete circle. Try an angle value of 1860 (1860 = 60 + 360 ×5). The result is the same as you obtained with angle values 60 and 420. So keep in mind in dealing with angles that a robot's orientation does not change by adding one or more full turns to the orientation.

Now let us have some fun with the method turnRight:. Script 4-5 draws the hour and minute hands of a clock together with a reference line. It uses two robots, which you can use to investigate the correspondence between a left turn and a right turn. I have added comments surrounded by quotation marks and have employed a variety of font effects to help you to identify the different parts of the script. Note that you do not have to type these comments, since they are not executed.

Script 4.3 (*Moving through angles (2)*)

```
| pica daly |
pica := Bot new.
pica jump: 200.
"drawing the reference line"
pica turnLeft: 180.
pica go: 200.
pica turnLeft: 180.
pica color: Color blue.
pica turnLeft: 45.
"drawing the minute hand"
pica go: 150.
daly := Bot new.
daly color: Color red.
daly turnRight: 45.
"drawing the hour hand"
daly go: 100.
```



In Script 4.3, the code in italics draws the reference line—that is, the line representing the direction of the robot before a turn method is executed—using the fact that a turn through 180 degrees amounts to turning around to point in the opposite direction. The reference line is also the longest line drawn. Thus, the reference line will still be visible if the lines drawn by the robots fall on top of it. The text in normal roman font following the italics is the code that draws the minute hand (using pica) and in

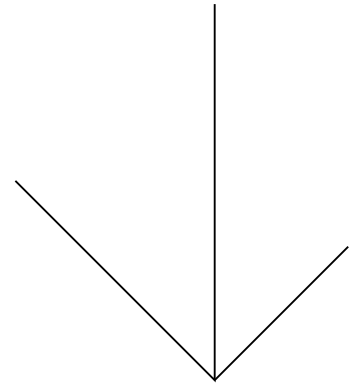
bold, the code drawing the hour hand using the robot daly.

Experiment 4.5 (*Moving Clock Hands)*
Experiment with different angle values for each of the two robots; that is,change the angle values for the two turn methods. Then, compare the effect of the method turnLeft: 60 (for pica) and turnRight: 300 (for daly). You can see that turning left 60 degrees yields the same result as turning right 300 degrees. This is so because the sum of the two values is 360 degrees, that is, a full circle.

Now let us see what happens when the robot turns from another direction. Here is the same script as Script 4-4 but showing the effect of turning from the north. In this script we are replacingdalyby another robot, berthe, who honors the French impressionist painter Berthe Morisot.

Script 4.4 (*Moving through angles (3))*

```
| pica berthe |
pica := Bot new.
pica north.
pica jump: 200.
pica turnLeft: 180.
pica go: 200.
pica turnLeft: 180.
pica color: Color blue.
pica turnLeft: 45.
pica go: 150.
berthe := Bot new.
berthe north.
berthe color: Color red.
berthe turnRight: 45.
berthe go: 100.
```



Experiment 4.6 (*Changing the Reference Direction)*
Continue to experiment with Script 4.4 by changing the reference direction. For the comparison to be meaningful, you have also to orient berthe in the same direction as pica after creating her. Try any angle values you like and try to predict what the resulting drawing will look like before executing the script. Continue experimenting with the script until your predictions are accurate.

Note that you should always be able to predict what is going to happen

before executing a script, because a computer blindly executes all valid statements, even the silliest ones.

4.5 A Robot Clock

I have mentioned that the lines drawn in Script 4-6 are akin to the hands of a clock. The anal- ogy between time and angles is a good one, for the notion of degrees is strongly correlated with that of hours. Ancient civilizations discovered the notion of time by measuring the angle of the sun (or a star) relative to a reference direction. However, a script like Script 4-6 allows you to place the hands in a position that does not indicate a real time of day. For example, you could draw a clock with the hour hand pointing north and the minute hand pointing south. But on a real clock, when the minute hand is pointing south, it is half past the hour, and so the hour hand should be halfway between two numbers on the clock's face.

Now you will study the relationship between the hour hand and the minute hand on a realclock that represents a realtime of day.

Experiment 4.7 (*A "real" Clock)*
Modify Script 4.4 as follows:

- Keep the direction of reference to the north (this is how Script 4-6 is written). This reference line indicates 12:00 noon or midnight.
- Use the method turnRight: for both robots. After all,the hands of a clock move clockwise, which is to the right.
- You can ask pica to draw the minute hand by multiplying the number of minutes after the hour that you wish to indicate by 6 (since during the 60 minutes in an hour, the minute hand travels the $6 * 60 = 360$ degrees in a full circle). For example,to represent the minute hand for 20 minutes after the hour, you should use the expression turnRight: 120 (since $120 = 6 * 20$).
- You can ask berthe to draw the hour hand by multiplying the number of the hours you want to indicate by 30 (12 hours times 30 degrees per hour equals 360 degrees) and then adding one-half (0.5) of a degree for each minute after the hour,since in 60 minutes,the hour hand moves 30 degrees.For example, the hour hand is positioned for 2 o'clock with the message turnRight: 60 $(60 = 30 * 2)$, while the time 4:26 requires the hour hand to be positioned with the message turnRight: 133 $(133 = 30 * 4 + 26 * 0.5)$.

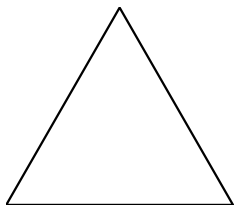
Try to indicate a few times of your choice with this modified script.

4.6 Simple Drawings

To begin with, here is a script for drawing a triangle with three equal sides:

Script 4.5 (*An equilateral triangle)*

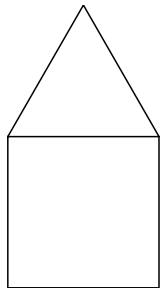
```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 120.
pica go: 100.
pica turnLeft: 120.
pica go: 100.
pica turnLeft: 120.
```



The last line of code is not necessary for drawing the triangle; it serves to point pica back in his initial position.

Now, you are ready to draw a house.

Experiment 4.8
A House Draw a house as shown in the figure. Try to draw houses of different shapes.

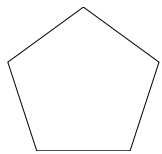


4.7 Regular Polygons

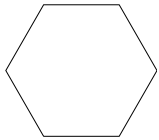
A regular polygon is a figure composed of line segments all of the same length and all of whose angles are equal. An equilateral triangle is a regular polygon with three sides. A square is a regular polygon with four sides. For example, Script 4.5 draws an equilateral triangle whose side length is 100 pixels. It is obtained by telling pica to go forward 100 pixels and then turn 120 degrees left, and then repeating these two messages two more times so that they are executed three times altogether.

You can program a robot to draw a regular polygon with any number of sides by asking it to move a certain length and then turn left or right by 360 degrees divided by the number of sides; this sequence must be repeated as many times as there are sides. Note that the last turn by the robot can be omitted, since the robot has drawn the last line of the polygon.

Experiment 4.9
A House Draw a regular pentagon (a regular polygon with five sides), as shown in the figure,with sides of length 100 pixels.

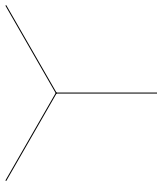


Experiment 4.10
A House Draw a regular hexagon (a regular polygon with six sides), as shown in the figure, with sides of length 100 pixels.



If you are just curious to see how far you can go with this process, you can use the cut and paste feature of the Bot workspace to generate a regular polygon with a large number of sides. If you are in the mood, go on increasing the number of sides. However, in Chapter 7, I will show you how you can type a sequence of expressions once and then have them repeated over and over.

Experiment 4.11
3 spaces Draw the three-spoked figure shown below.



4.8 Summary

- A robot can be oriented relative to its current direction using the methods turnLeft: and turnRight:.
- The parameter given to the methods turnLeft:andturnRight:is given in degrees.
- Turning 360 degrees corresponds to a turn through a full circle.
- Turning 180 degrees corresponds to a turn through a half circle.
- Angle values whose difference is a multiple of 360 degrees are equivalent.

Here is a list of the methods that you have learned about in this chapter.

Method	**Syntax**	**Description**	**Example**
turnLeft:	turnLeft: aNumber	Tell the robot to change its direction by a given number of degrees to the left.	pica turnLeft: 30
turnRight:	turnRight: aNumber	Tell the robot to change its direction by a given number of degrees to the right.	pica turnRight: 30
turn:	turn: aNumber	DTell the robot to change its direction to a given number of degrees following the mathematical convention that a turn is to the left if the number is positive and to the right if it is negative.	pica turn: 30
beInvisible	beInvisible	Hide the receiver.	pica beInvisible
beVisible	beVisible	Show the receiver.	pica beVisible

Chapter 5

Pica's Environment

In this chapter, I will present pica's environment and show you how to obtain tools and save your scripts. I will also return to the notion of messages and show that you can ask the environment not only to execute a message, but also to print the result of the message execution.

5.1 The Main Menu

When you click on the background you get the main menu of the environment, as shown in Figure 5.1.

If you want to know what a particular menu item does, simply move the mouse pointer over it for a second, and voilà! a balloon should pop up describing the item. The main menu gives access to five main groups of functionalities: access to tools, screen capture, access to some robot behavior, appearance, and saving the environment. The submenus are grouped as follows:

- The `open` menu collects several tools such as the robot code browser, the Bot workspace, a file browser, and other tools that I will present as needed.
- The `BotsInc actions` menu collects several actions such as indicating the version of the environment and clearing all robots and their traces, as well as some actions to reinstall the environment if needed: reinstalling default preferences resets the preferences that you may have modified using the appearance menu to their default values.
- The `appearance` menu collects actions that change the appearance

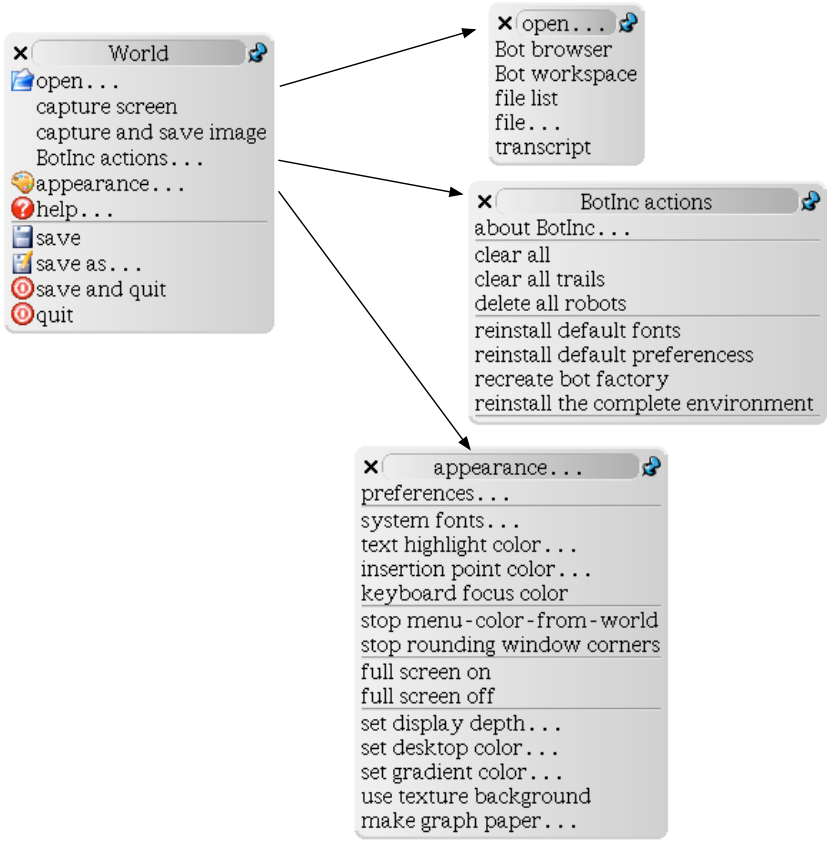


Figure 5.1: Menu options of the environment

of the environment such as fonts used, full-screen mode, and background color.

5.2 Obtaining a Bot Workspace

If you happen to close the default Bot workspace, don't worry. You can get a new one easily from the dark blue flap, as shown (though not in blue) on the left side of Figure 5.2, or from the main menu, as shown in Figure 5.1. To install a new Bot workspace in the working flap, open the working flap (bottom flap) and drop the Bot workspace from the blue flap

into the bottom flap.

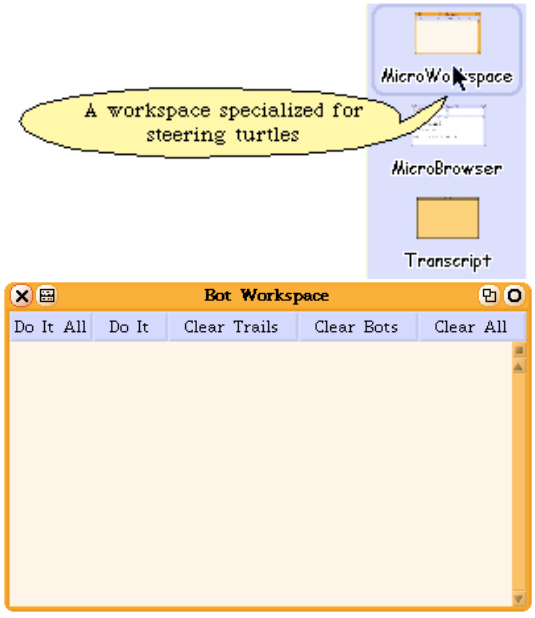


Figure 5.2: Obtaining a new Bot workspace from the flaps.

The dark blue flap contains other tools that we are going to use in the future. The second tool is basically a code browser that you will use when you define new robot methods.

The environment contains a simple tool (Figure 5.3) that lists the most important messages that a robot can understand. You can obtain access to this tool via the `open` vocabulary menu or the help menu (open vocabulary). The vocabulary pane lists the messages, grouped according to type. For example, the messages `east`, `north`, and so on are listed under absolute directions.

5.3 Interacting with Squeak

Interaction with Squeak is based on the assumption that you have a three-button mouse, though there are button equivalents for a Windows two-button mouse or Macintosh one- button mouse, as shown in Table below. Each button is associated with a logical set of operations. The left button is

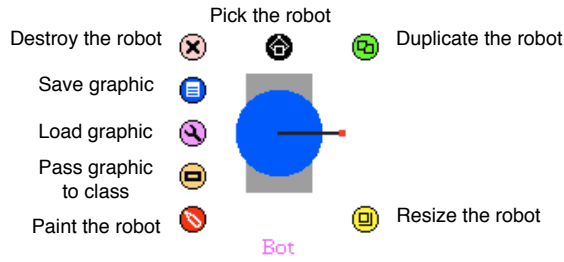


Figure 5.3: Right clicking on a robot brings up its halo of handles.

for obtaining contextual menus and for pointing and selecting, the middle button is for window manipulation (bringing a window to the front or moving it), and the right button is for obtaining handles, which are small colored and round buttons floating around graphical elements (as shown in Figure 5.7). Collectively, the handles are called a *halo*. The handles are useful, for they allow you to interact directly with the robot. I will present them in detail in the next chapter.

	Pointing and Selecting	**Context-Sensitive Menus**	**Open the Halo**
Three Buttons:	Left click	Center click	Right click
Windows 2-button equivalent:	Left click	Alt-left click	Right click
Mac 1-Button equivalent:	Click	Option-click	Command-click

Mouse Button and Key Combinations

5.4 Using the Bot Workspace to Save a Script

The Bot workspace has five buttons and a menu that allow you to save scripts. The button `Do It All` executes the entire script contained in the workspace. The button `Do It` executes the part of the script in the workspace that is currently selected. The button `Clear Trails` clears only the robot trails without removing the robots themselves. The button `Clear Robots` removes only the robots without clearing their trails. The button `Clear All` removes all the robots and their trails.

Once you have written a script, you may wish to save it to a file for

future use. The Bot workspace provides a way of saving and loading files via the workspace menu. Click on the contents of the workspace to bring up its associated menu, as shown in Figure 5.4. The menu item save contents will save the complete contents of the workspace into a file. Selecting this menu item brings up a dialog box, as shown in the figure. Note that the system checks whether a file with the same name already exists. If such a file already exists, the system gives you the choice of overwriting the file or saving it under another name.

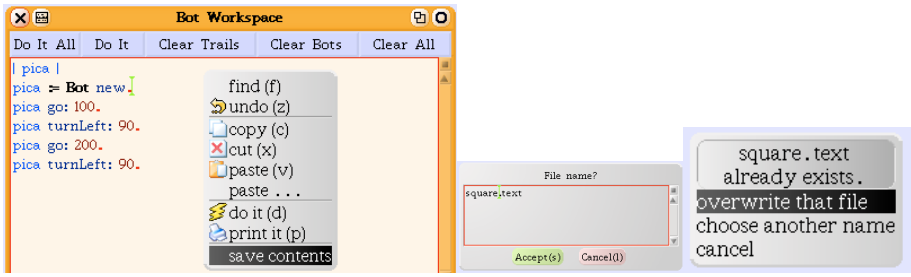


Figure 5.4: Left: Bot workspace menu options. Middle: Specifying the name of the file in which the script is to be saved. Right: If a file already exists,you can overwrite it or rename it.

5.5 Loading a Script

To load a script, you have to use a file list, a tool that allows you to select and load different files into Squeak. You can obtain a file list by selecting the menu item open file list from the main menu. A file list comprises several panes. The top left pane allows you to navigate through volumes and folders; each time you select an item in this pane, the top right pane is updated. It shows all the files contained in the folder that you selected in the left pane. When you select a file in the right pane, the bottom pane automatically displays its contents. Figure 5.5 shows that we are in the folder Bot testing, in which the file square.text is selected.

To load a script, you simply have to copy the contents of the bottom pane using the menu item copy and paste it into the Bot workspace using paste, just as you would in any text editor.

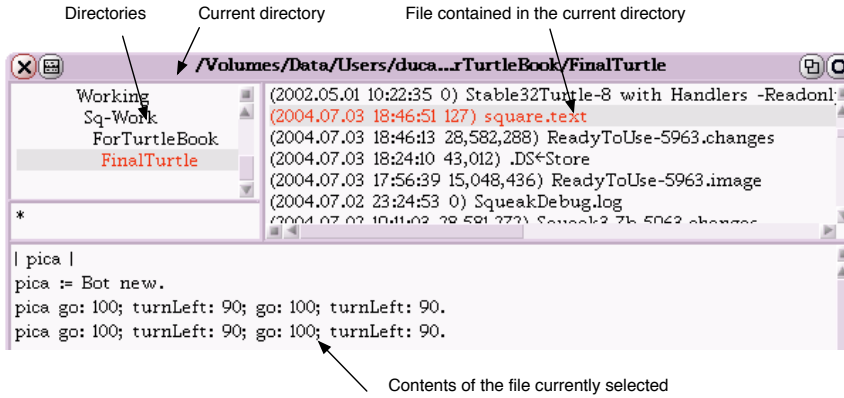


Figure 5.5: The file list is open to the script square.text.

5.6 Capturing a Drawing

To keep a record of your drawings, you can use the screen capture feature of your computer. However, with some computers, screen capture is problematic. To avoid such problems, the environment offers a simple screen capture mechanism that works on any computer. Bring up the main menu by clicking on the background of the environment. The menu offers two items for capturing, named `capture screen` and `capture and save image`, as shown in Figure 5.6.

The easier of the two options is to use the capture and save image menu item. When you select this item, Squeak shows that it is ready to capture by changing the cursor's shape to that of a corner, as shown on the right-hand side of Figure 5.6. Place the cursor at the corner of the rectangular region you want to capture, click, and drag the mouse to delimit the region you want. The region is displayed in the bottom left corner of the Squeak window, and Squeak prompts you for the name of the file without extension that it will save.

If you want to capture a region of the screen, use the menu item capture screen. In this case, Squeak will not prompt you to save the file, but instead, it creates a picture on the Squeak desktop, which you can save by first calling up the handles by right clicking on the screenshot. A number of different handles should appear around the image, as shown in Figure 5.7. Once the halo, that is, the group of handles, has appeared around your image, click on the red handle, which opens a menu of actions that you can

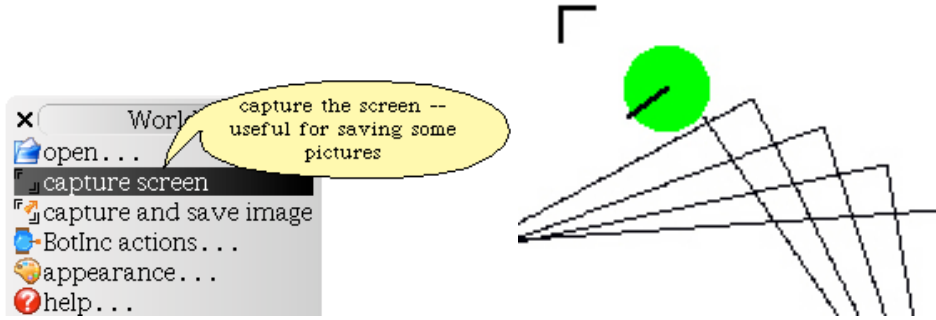


Figure 5.6: Left: Two possibilities for capturing and saving the capture. Right: The cursor has changed,indicating that Squeak is ready for the capture. Now click to position one corner of the rectangular region you want to capture.

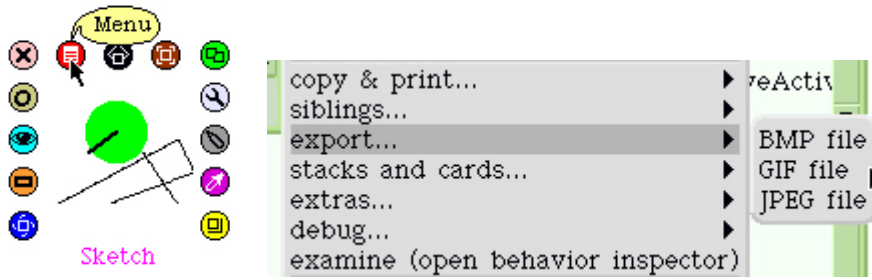


Figure 5.7: Call up the halo and choose the red handle menu item `export` to save the image to disk.

apply to the image. Select `export` and the format in which the image is to be saved. Squeak will prompt you for the name of the file. Note that you can import these files into Squeak by dropping them from the desktop onto the Squeak desktop.

5.7 Message Result

In Smalltalk, objects communicate only by sending and receiving messages to and from other objects. Once an object receives a message, it executes it, and additionally, it returns a result. A result is an object that the receiving

object has returned to the sender. Communication between objects by means of messages is similar to communication between people by sending letters: Some letters that we receive require us to perform certain actions (such as a warning from the dogcatcher to keep our dog on a leash), while others might require us to sign an acknowledgment that we have received the letter (a certified letter).

In Squeak, the receiver of a message always returns a result, which by default is the receiver of the message. However, this result is often not of interest. For example, sending the message `go: 100` to a robot tells the robot to move 100 pixels in its current direction. But we have no use for the result returned, which in this case is the robot itself, so in this case, we ignore the result. In many cases, though, the result of a message execution is important. For example, the expression `2 + 3` sends the message `+ 3` to the object 2, which returns the object 5. Sending the message `color` to a robot returns its current color. The result of a message can be used as part of another message in a compound message. For example, when the expression `(2 + 3) * 10` is executed, the expression `(2 + 3)` is executed, whereby the message `+ 3` is sent to the object 2, and this returns 5. The result 5 is then used as the object to which a second message, `* 10`, is sent. Thus 5 is the receiver of the message, and it then returns the result 50. The Squeak environment allows you to execute messages without dealing with the message's result, and it also allows you to execute messages and print the returned message value. The following section will illustrate this difference in detail.

Note A result is an object that the receiving object returns to the object that sent a message. For example, `2 + 5` returns 7 and `pica color` returns pica's color, a color object.

In Figure 5.8, the expression `50 + 90` is selected, then using the menu the expression is executed, and the result, 140, is printed on the screen.

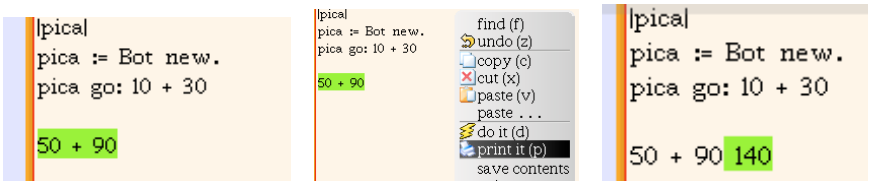


Figure 5.8: Left: Selecting the expression 50 + 90. Middle: Opening the menu.Right:Executing the message and having the result printed.

5.8 Executing a Script

There are three ways of executing a script.

1. Using the buttons of the Bot workspace editor. In Chapter 2 you saw a simple way to execute your first script by pressing the Do It All button of the Bot workspace. But to execute a script, you can also select the text you want to execute with the mouse (the selection turns green) and then press the Do It button of the Bot workspace.

2. Using the menu. Select the part of your script you want to execute, as shown, for example, in Figure 5.9. Then open the menu by pressing the middle button of your mouse (or press the option key while clicking with the left button), and then choose the do it (d) or the print it (p) menu item as shown in Figure 5.8.

3. Using keyboard shortcuts. Select a piece of text, then press command+D on a Mac or alt+D on a PC.

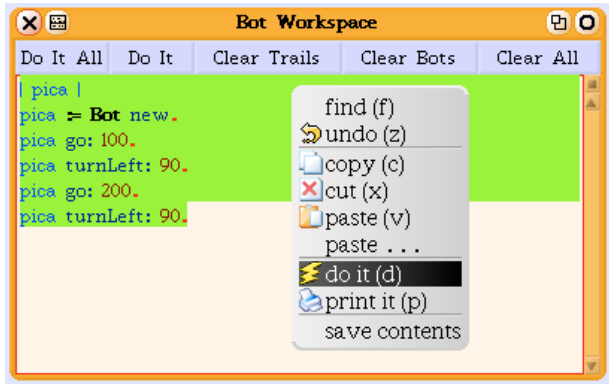


Figure 5.9: Selecting a piece of a script and executing it explicitly using the menu.

5.9 Hints

To automatically select all the text of a script, you can simply click at the start of the text (before the first character), at the end of the text, or on the line after the last expression. If you want to select a word, you can double

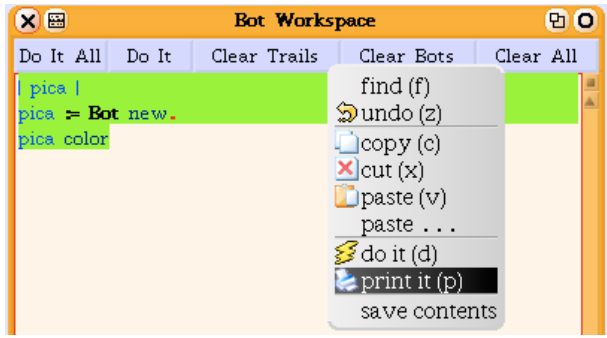


Figure 5.10: Open the menu and select the item Print it (d)to execute the selected piece of code.

click anywhere on the word. If you want to select a line, just dou- ble click at the beginning (before the first character) or end (after the last character) of the line.

5.10 Two Examples

When you execute the expression `pica color`, which asks the robot its color using the do it (d) menu item, the message color is sent and executed. However, you have the impression that nothing happens. This is because you have not asked the system to do anything with the result of the message execution. If you are interested in the result of a message, you should use the menu item `print it (p)`, as shown in Figure 5.10. This has the effect of both executing the piece of code selected *and* printing the result returned by the last message in the code. In the figure, the expression `Bot new` is executed, and then the message color is sent to the newly created robot. The message `color` is executed, and the color of the receiving robot is returned and printed, as shown in Figure 5.11. The text `(TranslucentColor r: 0.0 g: 0.0 b: 1.0 alpha: 0.847)` tells us that the color of the robot is a transparent color composed of the three color components red, green, and blue.

Let's look at a final example to make sure that you understand when to use print it. When you execute the expression `100 + 20` using the menu item do it (d), the message `+ 20` is sent to the object 100, which adds 20. However, you do not see anything. This is normal, because in such a case the execution of the message `+ 20` returns a new number representing the sum, but you did not ask Squeak to print it. To see the result, you have to print the result of the message execution using the menu item print it. From now on, we will write "-Printing the returned value:" to indicate that we

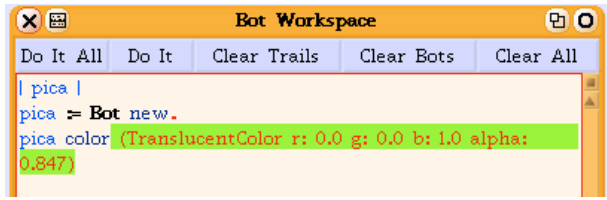


Figure 5.11: The result of the message is printed as a textual representation of a color.

are using the print command to execute an expression and print its result, as shown in Script 5.1. Note that we will use this convention only when the result is important.

Script 5.1: *Printing the result of executing an expression*

```
(100 + 20) * 10
–Printing the returned value: 1200
```

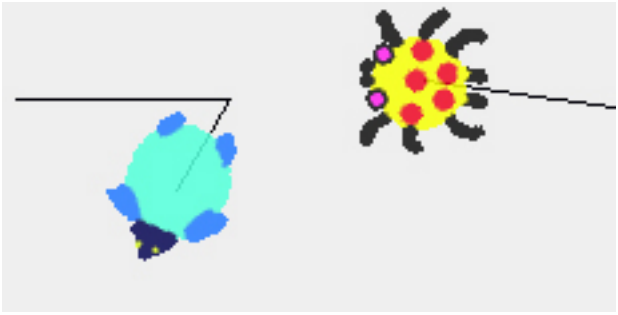
There are two ways of executing an expression: (1) using the Do It menu item to execute an expression, and (2) using the `Print it` menu item to execute it and print the returned result.

5.11 Summary

- To execute an expression, select a piece of text representing one or several expressions and press the `Do It` button or select the menu item do itfrom the execution menu.
- A result is an object that you obtain from a message. For example, `pica color` returns the color of the robot.
- There are two ways of executing an expression, (1) using the `do it` menu item to execute an expression, and (2) using the `print it` menu item to execute it and print the returned result.

Chapter 6

Fun with Robots



The basic look of a robot is rather simple. Wouldn't it be nice to be able to create robots that had a bit more pizzazz to them? Fortunately, you can create customized robots, and in this chapter, I will show you how you can change the shape, the pen size, and the color of your robots. You can make your robot look like an animal, a monster, or even the famous robot R2D2 from the movie Star Wars.

6.1 Robot Handles

You have learned about opening a message balloon for sending a message to a robot by click- ing on that robot. Now you are going to learn about

obtaining access to other robot functions such as duplicating, moving, and changing the look of a robot. These extra functionalities are available via the halo of handles, which, as was mentioned briefly in Chapter 5, you can beam up by right clicking (command clicking on a Mac) on a robot. The handles are the small, round icons that surround the robot like a halo, as shown in Figure 6-1. I will explain the functions of the different handles as they are needed. You can get information about a handle by letting your mouse rest over a handle; then a balloon pops up and explains the handle's purpose. For now, try to make a copy of the robot by clicking on the green ("duplicate the robot") handle, move the robot by clicking on the black ("select the robot") handle and dragging the robot, or destroy the robot using the pale pink ("destroy the robot") handle with the "X."

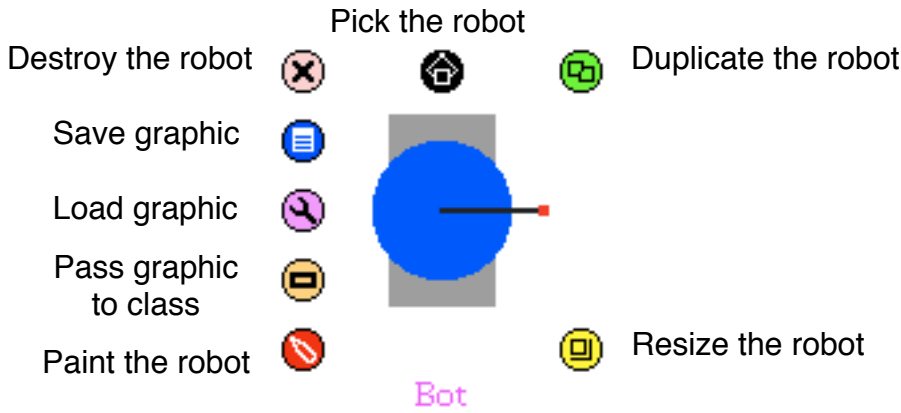


Figure 6.1: Right click (command click) on the robot to bring up the halo of handles.

6.2 Pen Size and Color

When our robots moved around the screen in previous chapters, they left a black trace in their wakes. But you are not limited to the default color black. You can change the color of a robot's pen by sending a robot the message penColor: with a color object as argument. One of the ways of obtaining a color object is to send a message with the name of a color to the class Color, which is a factory that makes color objects. For example, Color blue yields a blue color object, and Color yellow yields a yellow color object. Thus you can change the pen color of the robot pica to the color blue with the

message send pica penColor: Color blue. I will explain more about colors in the following section.

You can also change the thickness of the robot's pen by sending the message penSize: with a number as argument. For example, pica penSize: 5 orders pica to change his pen size to be 5 pixels wide. Script 6.1 draws a thick blue line of width 5 pixels.

Script 6.1: *Pica can draw a thick blue line.*

```
| pica |
pica := Bot new.
pica penColor: Color blue.
pica go: 100.
pica penSize: 5.
pica go: 100
```

Script 6.1 (*Pica draws a spyglass.)*

```
| pica |
pica := Bot new.
pica go: 40.
!pica penSize: 2.!
pica go: 40.
!pica penSize: 4.!
pica go: 40.
!pica penSize: 6.!
pica go: 40.
```

You can change the color of the robot itself using the method color:. For example, the message send berthe color: Color yellow changes the robot berthe 's color to yellow. Script 6.2 tells berthe to change her color to yellow and then go forward 100 pixels, while pica is left behind with his default color and without moving.

Script 6.2: *Berthe changes her color and goes for a walk,while pica is left behind.*

```
| pica berthe |
pica := Bot new.
berthe:= Bot new.
berthe color: Color yellow.
berthe go: 100.
```

6.3 More about Colors

As previously mentioned, Squeak is an environment that is built from objects and that uses objects. Therefore, programming in Squeak amounts to creating objects and sending them messages. In particular, a color is an object created by the class Color. To obtain a color object, you send a message to the class Color.

Some color messages are named for the color they represent. For example, Color red causes the class Color to create a red color object. Here is the list of the predefined message selectors that you can send to the class Color to create that color: black, veryVeryDarkGray, veryDarkGray, darkGray , gray, lightGray, veryLightGray, veryVeryLightGray , white, red, yellow, green, cyan, blue, magenta, brown, orange, lightRed, lightYellow, lightGreen, lightCyan, lightBlue , lightMagenta, lightBrown, lightOrange, paleBuff, paleBlue, paleYellow, paleGreen, paleRed, veryPaleRed, paleTan, paleMagenta, paleOrange, and palePeach.

The Color class is like a real-life paint factory. Not only can it make a large number of standard colors, it can also create a customized color for you by combining different amounts of red, green and blue. Table 6-1 shows a few examples of how to create colors this way using the message r:redAmount g: greenAmountb: blueAmount. The arguments taken by the mes- sage selector r:g:b: should be decimal numbers between 0 and 1 representing the amounts of red, green, and blue to be combined. For example, the expression Color r: 1 g: 0 b: 0 creates the same pure red color that you get from Color red. Using the same amount of each of the three colors produces a shade of gray. All ones produces white, and all zeros produces black.

Table 6.1: Creating Colors with Color r:g:b:

Color	r: (Red)	g: (Green)	b: (Blue)
red	1	0	0
light gray	0.1	0.1	0.1
yellow	1	1	0
white	1	1	1
black	0	0	0
gray	0.5	0.5	0.5
pale green	0.8739	1	0.8348

Finally, the method fromUser lets you pick a color from a palette on

the screen, and then shows you that color's ingredients, as illustrated in Figure 6.2 (though you will have to imagine the colors). For that, you need to execute the expression Color fromUser using the print it menu to get the result of the selection printed.

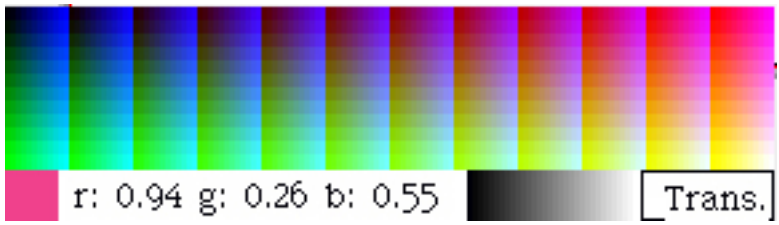


Figure 6.2: Choose your color from a color palette with the message send Color fromUser.

6.4 Changing a Robot's Shape and Size

You can change a robot's shape as well as its color. In addition to the default robot shape, two shapes, a circle and a triangle, are built into the Bot factory (but you can also draw the robot shape with a drawing tool, as shown in the next section). The message lookLikeTriangle gives a triangular shape to a robot. The message lookLikeCircle gives a circular shape to a robot. The default shape is produced by sending the message lookLikeBot.

Another aspect you can change is the size of a robot using the message extent: widthAndHeight, where the values of widthAndHeight represent the width and height of the rectangle in which the robot is drawn. The argument widthAndHeight is a pair of numbers, also called a point in Squeak. It is composed of two numbers separated by the @ symbol. For example, the point 50@100 represents a rectangle 50 pixels wide and 100 pixels tall.

Thus to create a robot named bigpica in the shape of a triangle that fits inside a square with dimensions 150@150, you would first send bigpica the message lookLikeTriangle and then the message extent: 150@150.

Figure 6.3 shows some robot shapes created using the built-in triangle and circle shapes, and Script 6.3 shows how to create robots of these sizes and shapes and move them into position as shown in the figure.

Script 6.3: *Creating robots of different sizes and shapes (circles and triangles)*

```
| pica daly bigpica |
pica := Bot new.
pica lookLikeTriangle.
```

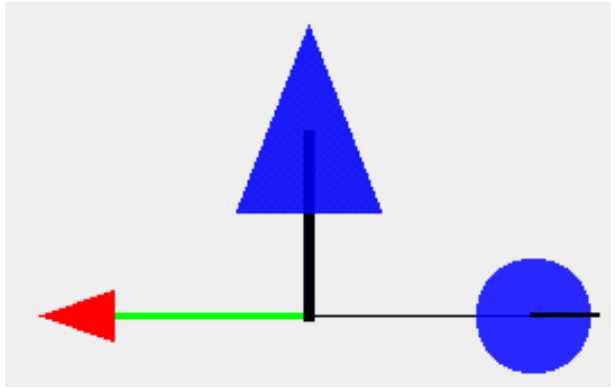


Figure 6.3: Robots can come in different shapes and sizes.

```
pica west.
pica color: Color red.
pica penColor: Color green.
pica penSize: 3.
pica go: 100.
daly := Bot new.
daly extent: 60@60.
daly east.
daly go: 100.
bigpica := Bot new.
bigpica lookLikeTriangle.
bigpica extent: 150@150.
bigpica penSize: 5.
bigpica north.
bigpica go: 80.
```

6.5 Drawing Your Own Robot

Squeak lets you draw a customized robot. You can even create a robot that looks like one of the figures shown at the beginning of this chapter. I will now describe step by step how to draw your own robot.

Step 1:Open the painting tool via the red handle. The first step is to open the painting tool that is included in Squeak. Right click (or command click for Mac) to beam up the halo around the robot that you want to paint,

as shown in Figure 6.4. Click on the red handle, the one with the icon of a pen inside. This will open the painting editor, which is depicted in Figure 6.5. Do not worry about the other handles. Note that if you have already drawn a graphic, that graphic will be shown inside the painting tool.

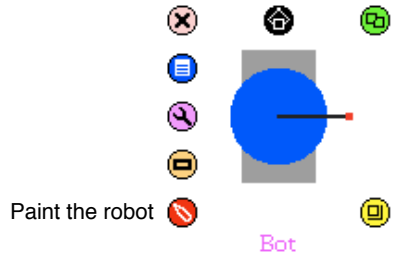


Figure 6.4: Right click (or command click) to obtain the halo.Choose the painting editor from the red handle.

Step 2: Draw the new robot graphic. The second step is to draw a new graphic for your robot. Draw your robot pointing to the right, as shown in Figure 6.6. The painting editor has the usual features of graphics programs, such as selecting the brush size, filling a region, repeating a selected region, and selecting the paint color. The painting tool also has two buttons (shown in Figure 6.7) to rotate and zoom your drawing.

Step 3:Preserving your graphic. Once you are satisfied with your drawing, you should press the button keep. This closes the painting tool. Now your robot looks like the graphic that you created.

6.6 Saving and Restoring Graphics

If you have spent lot of time drawing a robot and you would like to save it for future reference, you can save it to a file. Once it has been saved, you will be able to load it into different environments and share it with your friends. You can begin to build a library of robot graphics over time. Now I will show you how to save and load a graphic. Then I will show how you can associate a graphic with a single robot or even to a class (robot factory), so that all newly created robots will look like the graphics that you have

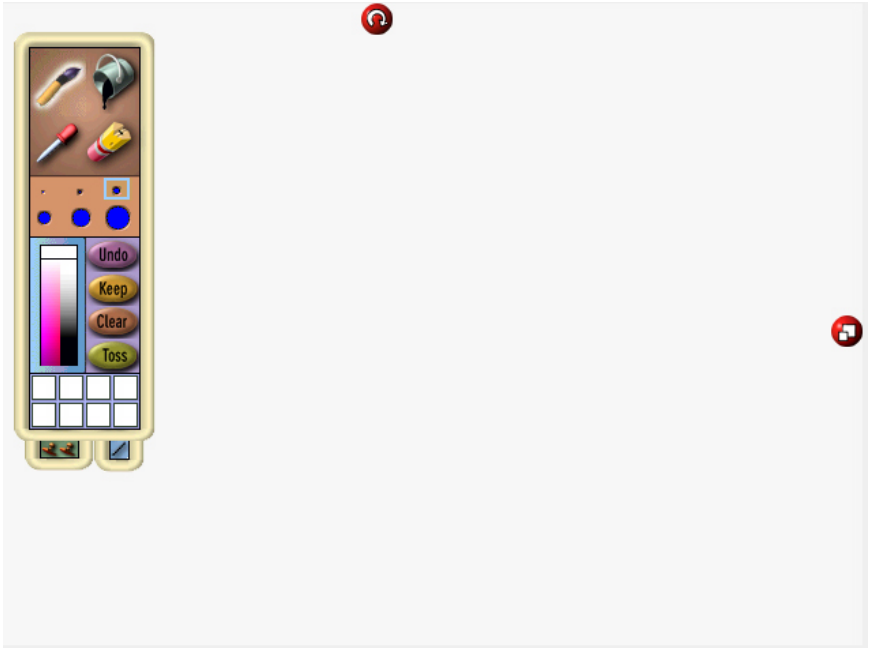


Figure 6.5: The painting editor.

drawn. I will start by showing you how to perform all these manipulations by interacting with the robots directly, and then how to write scripts to do these things automatically.

The "Save Graphics"Handle

To save a graphic, simply click on the blue handle, the one with the file icon (Figure 6.8). I chose the color blue to make you think of a frozen lake: saving the graphic "freezes" your robot's shape to preserve it. The system will then ask you to give a name to the saved graphic, as shown in Figure 6-9. This operation saves your graphic to a file, in the same folder as the Squeak image, with the name you entered and with the extension .frm.

You can reverse the operation and load a graphic by clicking on the pink handle in the robot's halo, the one with an icon that looks like a tool that a robot might use. I chose the color pink to make you think of bringing your robot back to life. When you click on the pink handle, the system asks you for the name of the graphic you want to load. Your robot will take on the appearance of the graphic that you choose.

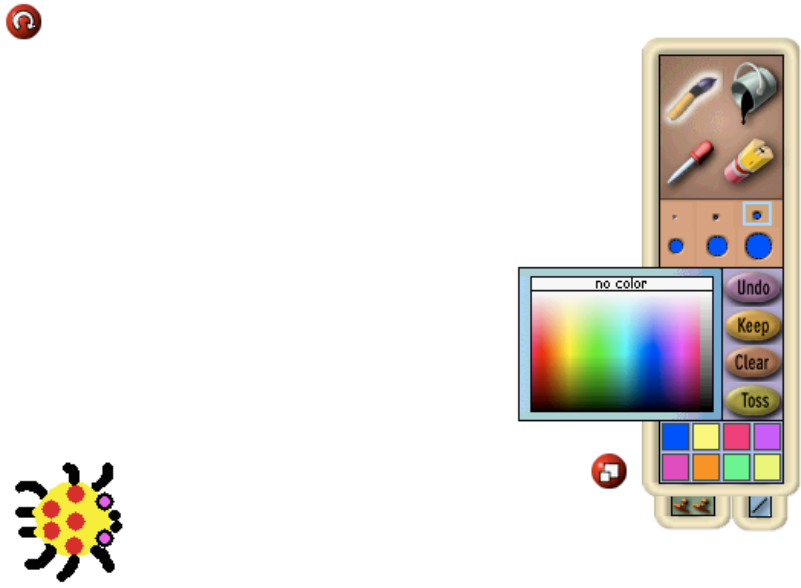


Figure 6.6: This robot looks like a spotted spider.

Figure 6.7: The zoom and rotate buttons.

6.7 Retooling the Robot Factory

You have drawn and saved a beautiful spotted spider, and you would like the robot factory to make you a robot with this graphic, but when you tell the Bot class to create a new robot, it creates one using the default graphic. In order for the Bot class to be able to create your spider robot, you have to tell your robot to pass its graphic to the class using the message passImageToClass. After you have sent this message, if you create a new robot and ask it to look like the image, it will look like the graphic that you

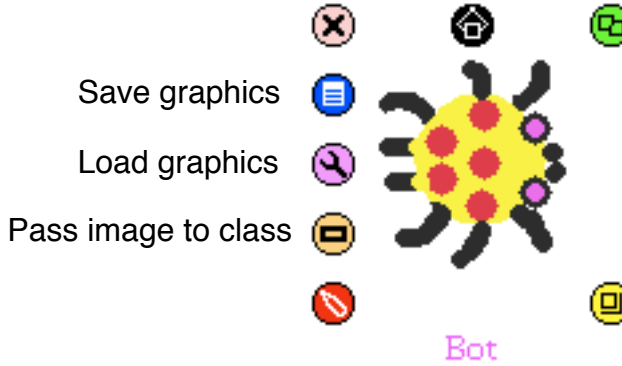


Figure 6.8: The robot now looks like a spider.It is pointing to the right.

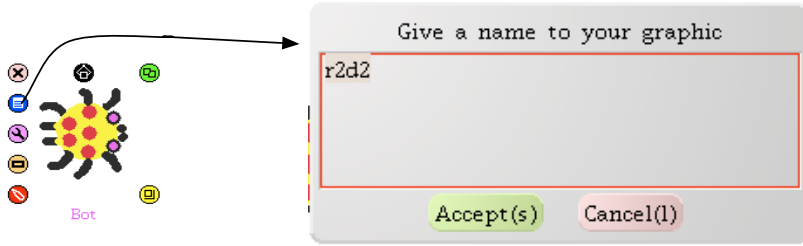


Figure 6.9: Clicking on the blue handle produces a prompt for a name.

just drew.

Another way of obtaining the same result is to send lookLikeImage or any of the lookLike messages to the Bot class itself. Then the class will be configured to create new robots according to the new configuration. For example, if you send the message lookLikeCircle to the class Bot, all newly created robots will look like a circle. Thus if you want to have the Bot class create robots that look like spiders, you have to (1) get a robot, (2) draw the spider or load a previously saved spider graphic, (3) pass the spider image to the class, and (4) tell the class to make robots with the image by sending it the message lookLikeImage. Then all your newly created robots will look like a spider, as shown in Figure 6.10.

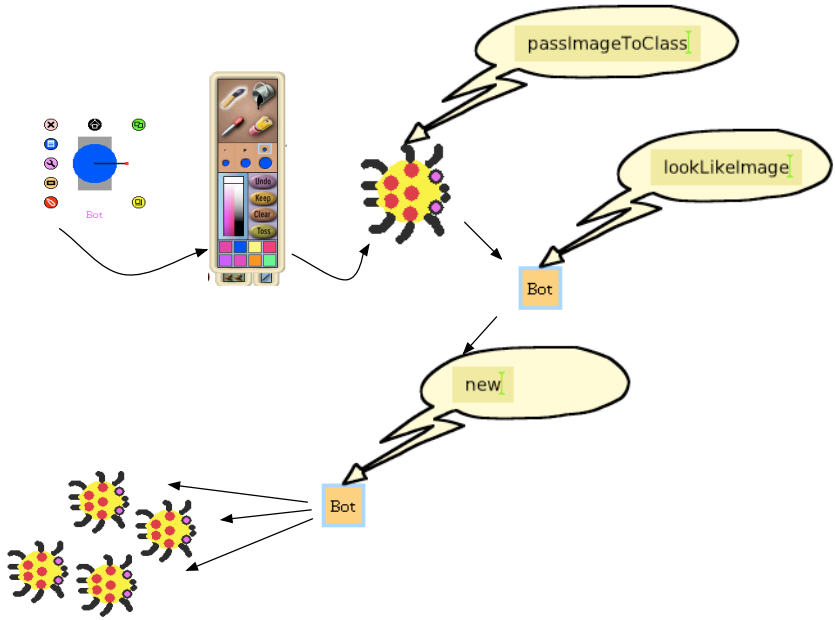


Figure 6.10: Passing an image to the Bot class and sending the message lookLikeImageresults in all newly created robots looking like that image.

6.8 Graphics Operations Using Scripts

You can also write a script to load and save graphics and associate them with a single robot or with a class.

Script 6.4 creates two robots and loads a graphic for each of them using two different methods. After picais created, he is sent the message loadImage , which results in a prompt to the user for the name of the image to load. Then bertheis created, and she is sent the message loadImage: 'spider', which gives her the image contained in the image file spider.frm.

Note the important distinction between the messages loadImageandloadImage: 'fileName'. The first of these has no parameter, and the user is prompted for the name of the graphics file to load. The second message does have a parameter, where here 'fileName' represents the name of the image file inside single quotation marks (and without the file extension).

You can save the image using the message saveImage or saveImage: '

fileName'. First berthe is sent the message saveImage, which prompts the user for the name under which the image is to be saved. Finally, pica is sent the message saveImage: 'spider2', which saves his image under the file name spider2.frm.

Script 6.4: *Two ways of loading and saving robot graphics*

```
| pica berthe |
pica := Bot new.
pica loadImage.          "The user is prompted for the name of the image to load"
berthe := Bot new.
berthe loadImage: 'spider'   "A parameter gives the name of the file to be loaded"
berthe saveImage.          "The user is prompted to name the saved image file"
pica saveImage: 'spider2'   "A parameter gives the name of the saved image"
```

Just as you can load and save graphics associated with an individual robot, you can load and save graphics that are to be associated with a class, such as the Botclass. The same mes- sages are used for the class as were used for the individual robots. They are just sent to Bot instead of to picaor berthe. Script 6.5 first associates the image spider.frmwith the Botclass. Then the image is saved under another name, spiderBot.frm.

Instead of the methods loadImage: and saveImage:, you can use loadImage and saveImage (no colon, no argument), which prompt the user for a file name. The expression Bot clearImage resets the Bot class to its condition the first time you used it. This restores the default robot image to the class, which means that when you run the script, you can reproduce a predictable scenario.

Script 6.5: *Loading and saving a graphic associated with the* Bot *class*

```
| pica berthe |
Bot clearImage.          "clears any graphic that was previously associated
with the class Bot."
berthe := Bot new.       "berthe looks like a default robot"
Bot loadImage: 'spider'.  "The image in spider.frm is associated with the Bot class"
Bot lookLikeImage
pica := Bot new.         "The robot pica looks like a spider"
Bot saveImage: 'spider3'  "The spider image is saved under the name spider3.frm"
```

The following scripts (Script 6.6 and 6-8) assume that three image files, luth.frm, spider.frm, and airplane.frm are located in the directory containing the Squeak image. These files are included in the distribution of the environment used in this book.

Script 6.6 uses the method loadImage: to associate an image with a robot, and the method lookLikeImage to instruct a robot to look like the image with which it is associated. After the robot pica is created, he is asked to

look like his associated image (pica lookLikeImage). Since no graphic has been associated with pica, asking him to look like an image produces no change. But then the image from the file luth.frm is associated with pica by sending him the message loadImage:'luth'. Now when pica is sent the message lookLikeImage, his appearance is changed, and he looks like a luth sea turtle. In the last line of the script, a different image file is associated with pica. Once it has been given the message lookLikeImage, a robot will look like whatever image is associated with it. Thus when the expression pica loadImage: 'spider' is executed, pica will look like a spider.

The script continues with a new robot, berthe, being created. Since the class Bot does not have any image associated with it, berthe will have the graphic of a default robot, and if you send her the message lookLikeImage, nothing changes, since she does not have an associated image.

Script 6.6: *Changing the image of a robot*

```
| pica berthe |
Bot clearImage.
pica := Bot new.
pica lookLikeImage.
"No image loaded or created, so nothing changes"
pica loadImage: 'luth'.
pica lookLikeImage
"Load an image and ask the robot to look like it"
pica loadImage: 'spider'.
"load another image, and since the message lookLikeImage has already been
sent, pica will look like the new image"
berthe := Bot new.
berthe lookLikeImage.
"When berthe is created, she looks like a default robot, and since no image
has been loaded into the class, the message lookLikeImage causes nothing to
change"
```

Script 6.7 shows how to notify the Bot class that all newly created robots should have a particular graphic. In contrast to the situation described in Script 6.6, the message loadImage: 'fileName' is sent to the class Bot itself and not to a particular robot. Just as a swimmer and a billiards player have different reactions to the word "pool," different objects and classes have different understandings of the same message. That is because every object or class has its own method that responds to a given message, and these methods may be different for the same message. In the case at hand, loadImage: has different behavior depending whether it is received by the Bot class or by a robot, which is an instance of the class. When received by the Bot class, the message loadImage: 'fileName' leads to the class loading and associating the graphic from the file, so that newly created instances (robots)

can use the new graphic. When received by a robot, only the particular robot receiving the message can use this graphic.

Script 6.7: *Associating a graphic with the* Bot *class*

```
| berthe daly pica yertle |
Bot loadImage: 'spider'.
berthe := Bot new.
berthe lookLikeImage.
"berthe, as an instance of the Bot class, now looks like a spider"
daly := Bot new.
daly lookLikeImage.
"daly also now looks like a spider"
pica := Bot new.
pica loadImage: 'luth'.
pica lookLikeImage.
"But a specific robot can still change its own graphics;
pica now looks like a turtle"
pica getImageFromClass.
"pica gets his image from the Bot class; now he looks like a spider again"
Bot loadImage: 'luth'.
Bot lookLikeImage.
yertle := Bot new.
"Now the class will create robots that look like luth turtles"
```

Script 6.7 starts by loading a new graphic from a file and associating it with the Bot class itself. Then the new robot berthe is created, and she is sent the message that tells her to use the new graphic. Creating another robot, daly, and sending him the message lookLikeImage makes him also look like the image associated with the class.

All robots created can be made to look like a spider. However, a particular robot, such as the robot pica in the script, can be given his own image by sending him the message loadImage: 'fileName'. The robot's associated image overrides the class image. The message getImageFromClass makes it possible to restore the graphic associated with the class. The last sequence of messages shows that we can associate a new graphic to a class, replacing the currently associated image. Sending the message loadImage: 'fileName' to the class Bot associates the graphic in the file fileName.frm to the class. Then sending the message lookLikeImage ensures that newly created robots will by default look like the graphic now associated with the class. Hence the robot yertle looks like a turtle.

6.9 Summary

Method	Description	Example
lookLikeCircle	Change the shape of the receiver to a circle.	Bot new lookLikeCircle
lookLikeBot	Change the shape of the receiver to a robot.	Bot new lookLikeBot
lookLikeTriangle	Change the shape of the receiver to a triangle.	Bot new lookLikeTriangle
lookLikeImage	Change the appearance of the receiver to the graphic you painted.	Bot new lookLikeImage
lookLikeCircle	Sending to the class results in newly created robots having the shape of a circle.	Bot lookLikeCircle
lookLikeBot	Sending to the class results in newly created robots having the shape of a robot.	Bot lookLikeBot
lookLikeTriangle	Sending to the class results in newly created robots having the shape of a triangle.	Bot lookLikeTriangle
lookLikeImage	Sending to the class results in newly created robots having the shape of the graphic you painted or loaded.	Bot lookLikeImage
loadImage: '*fileName*'	Load the image file *fileName.frm* into the class or the robot.	Bot loadImage: 'spider' or berthe loadImage: 'spider'
loadImage	Prompt the user for the name of an image file to be loaded into the class or the robot.	Bot loadImage or berthe loadImage
saveImage: '*fileName*'	Save the image of the class or the robot to the file named *fileName.frm*.	Bot saveImage: 'spider' or berthe saveImage: 'spider'
saveImage	Save the image of the class or the robot by prompting the user for a file name.	Bot saveImage or berthe saveImage
penColor: *aColor*	Change the color of the pen.	berthe penColor: Color blue

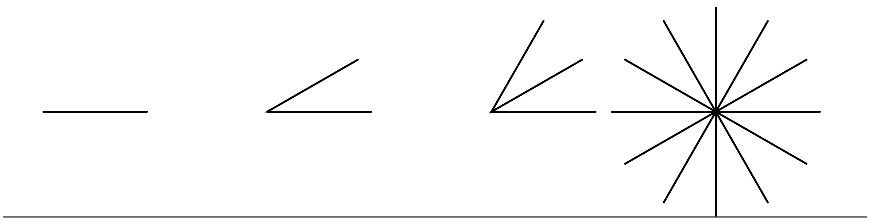
Method	Description	Example
penSize: *aNumber*	Change the size of the pen. The default size is 1.	berthe penSize: 3
color: *aColor*	Change the color of the receiver to the specified color.	berthe color: Color yellow
extent: *aPoint*	Change the size of the receiver to dimensions given by aPoint where aPoint is given by w@h, where w is the width and h is the height.	berthe extent: 80@100
passImageToClass	Pass the graphic of the receiver to the class. After this message, robots created by the class will have as graphic the graphic of the current robot.	berthe passImageToClass
getImageFromClass	Get the graphic of the class. After this message, the receiver will look like the robots that would be created by the class.	berthe getImageFromClass

Part II

Elementary Programming Concepts

Chapter 7

Looping



By now, you must think that the job of robot programmer is quite tedious. You probably have a number of ideas for interesting drawings, but you just don't have the heart to write the scripts to draw them, since it appears that the number of lines that you have to type gets larger and larger as the complexity of the drawing increases. In this chapter, you will learn how to use loopsto reduce the number of expressions given to a robot. Loops allow you to repeat a sequence of expressions. With a loop, the script for drawing a hexagon or an octagon is no longer than the script for drawing a square.

7.1 A Star Is Born

We would like to instruct a robot to draw a star, similar to the one shown in the picture at the beginning of this chapter. We will instruct pica to draw a star in the following way: starting at what will be the center of the star, draw a line, return to the center, turn through a certain angle, draw another

line, and so on until the star is finished. Script 7.1 creates a robot that draws a line of length 70 pixels and then returns to its previous location. Note that after it has returned to its starting point, the robot makes an about-face, so that it is pointing in its original direction.

Script 7.1: *Drawing a line and returning*

```
| pica |
pica := Bot new.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
```

To draw a star, we have to repeat part of Script 7.1 and then instruct the robot to turn through a given angle. Let's draw a six-pointed star, and so the angle will be 60 degrees, since turning 60 degrees each time will result in 360/60 = 6 branches. Script 7.2 shows how this should be done to obtain a star having 6 branches without using loops.

Script 7.2: *A six-pointed star without loops*

```
| pica |
pica := Bot new.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
```

```
pica turnLeft: 60.
pica go: 70.
pica turnLeft: 180.
pica go: 70.
pica turnLeft: 180.
pica turnLeft: 60.
```

As you can see, after pica is created, he repeats the same five lines of code six times (shown in alternating roman and italic type). It seems wasteful to have to type the same code segment over and over. Imagine the length of your script if you wanted a star with 60 branches, like the one shown in Experiment 7.1. What we need is a way of repeating a sequence of expressions.

7.2 Loops to the Rescue

The solution to our problem is to use a loop. There are different kinds of loops, and the one that I will introduce here allows you to repeat a given sequence of messages a given number of times. The method timesRepeat: repeats a sequence of expressions a given number of times, as shown in Script 7.3. This script defines the same star as the one in Script 7.2, but with much less code. Notice that the expressions to be repeated are enclosed in square brackets. Script 7.3.

Script 7.3: *Drawing a six-pointed star using a loop*

```
| pica |
pica := Bot new.
6 timesRepeat:
   [ pica go: 70.
   pica turnLeft: 180.
   pica go: 70.
   pica turnLeft: 180.
   pica turnLeft: 60 ]
```

Important! n timesRepeat: [sequence of expressions] repeats a sequence of expressions n times.

The method timesRepeat: allows you to repeat a sequence of expressions, and in Smalltalk, such a sequence of expressions, delimited by square brackets, is called a *block*.

The message timesRepeat: is sent to an integer, the number of times the sequence should be repeated. In Script 7.3 the message timesRepeat: [...] is sent to the integer 6. There is nothing new here; you have a message being sent to an integer when we looked at addition: the second integer was sent to the first, which returned the sum.

Finally, note that the number receiving the message timesRepeat: has to be a *whole number*, because in looping as in real life, it is not clear what would be meant by executing a sequence of expressions, say, 0.2785 times.

The argument of timesRepeat: is a block, that is, a sequence of expressions surrounded by square brackets. Recall from Chapter 2 that an argument of a message consists of information needed by the receiving object for executing the message. For example, [pica go: 70. pica turnLeft: 180. pica go: 70.] is a block consisting of the three expressions pica go: 70, pica turnLeft: 180, and pica go: 70.

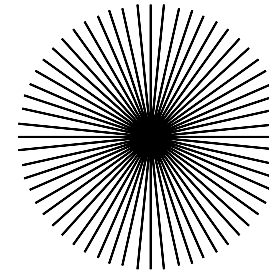
Important! The argument of timesRepeat: is a block, that is, a sequence of expressions surrounded by square brackets.

7.3 Loops at Work

If you compare Script 7.1 with the expressions in the loop of Script 7.3, you will see that there is one extra expression: pica turnLeft: 60, which creates the angle between adjacent branches. There is a simple relationship between the number of branches and the angle through which the robot should turn before drawing the next branch: For a complete star, the relation between the angle and the number of repetitions should be $angle * n = 360$.

To adapt Script 7.3 to draw a star with some other number of branches, you have to change the number of times the loop is repeated by replacing 6 with the appropriate integer. Note that the angle 60 should also be changed accordingly if you want to generate a complete star.

Experiment 7.1 (*A Star with Sixty Branches)*
Write a script that draws a star with 60 branches.



7.4 Code Indentation

Smalltalk code can be laid out in a variety of ways, and its indentation from the left margin has no effect on how the code is executed. We say that indentation has no effect on the syntactic "sense" of the program. However, using clear and consistent indentation helps the reader to understand the code.

I suggest that you follow the convention that was used in Script 7.3 in formatting timesRepeat: expressions. The idea is that the repeated block of expressions delimited by the characters [and] should form a visual and textual rectangle. That is why the block begins with the left bracket on the line following timesRepeat: and we align all the expressions inside the block to one tab width. The right bracket at the end indicates that the block is finished. Figure 7.1 should convince you that indented code is easier to read than unindented code.

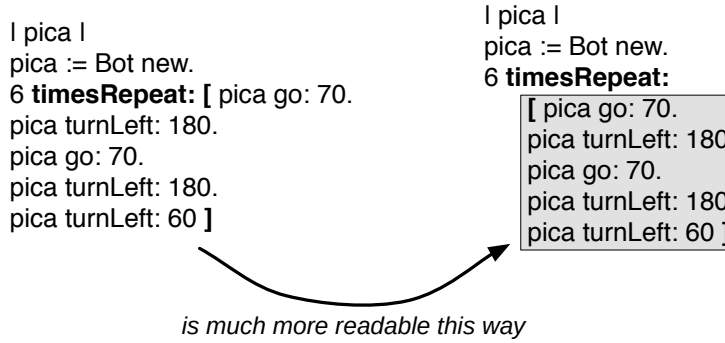


Figure 7.1: Indenting blocks makes it much easier to identify loops. Left: unindented. Right: indented

Code formatting is a topic of endless discussion, because different people like to read their code in different ways. The convention that I am proposing is focused primarily on helping in the identification of repeated expressions.

7.5 Drawing Regular Geometric Figures

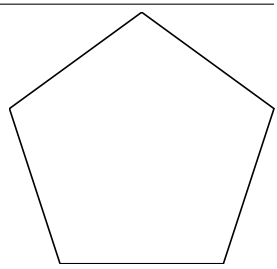
Many figures can be obtained by simply repeating sequences of messages, such as the square that was drawn in Chapter 4 (repeated here as Script 7.4).

Script 7.4: *Pica's first square*

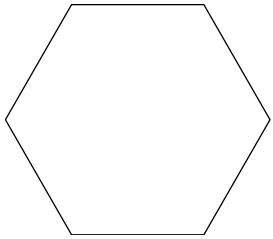
```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90
```

Experiment 7.2 (*A Square Using a Loop)*
Transform Script 7.2 so that it draws the same square using the command timesRepeat:. Now you should be able to draw other regular polygons, even those with a large number of sides.

Experiment 7.3 (*A Regular Pentagon)*
Draw a regular pentagon using the method timesRepeat:.



Experiment 7.4 (*A Regular Hexagon)*
Draw a regular hexagon using the command timesRepeat:.



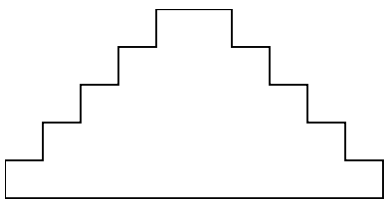
Once you have gotten the hang of it, try drawing a regular polygon with a very large number of sides. You may have to reduce the side length to make the figure fit on the screen. When the number of sides is large and the side length is small, the polygon will look like a circle.

7.6 Rediscovering the Pyramids

Recall how you coded the outline of the pyramid of Saqqara in Experiment 3.3. You can simplify your code by using a loop, as shown in Script 7.1.

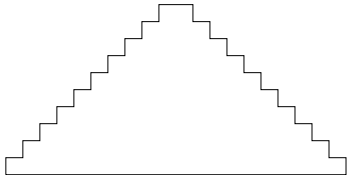
Script 7.1 (*A looping pyramid script)*

```
| pica |
pica := Bot new.
5 timesRepeat:
[pica north.
pica go: 20.
pica east.
pica go: 20].
5 timesRepeat:
[pica go: 20.
pica south.
pica go: 20.
pica east].
pica west.
pica go: 200.
```



Now you should be able to generate pyramids with an arbitrary number of terraces using the same number of expressions, merely by changing the numbers in the script.

Experiment 7.5 (*A Ten-Step Pyramid)*
Draw a pyramid with 10 terraces using a variation of Script 7.1.

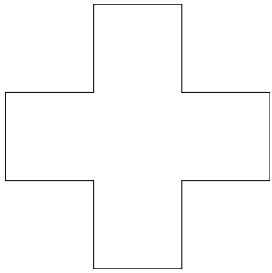


You may now want to generate pyramids with even larger numbers of terraces. The size of the terraces will have to be adjusted if you want them to fit on the screen.

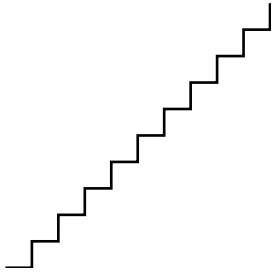
7.7 Further Experiments with Loops

As you have seen, generating a step pyramid involves the repetition of a block of code that draws two line segments. Once you have identified the proper repeating element, you can produce complex pictures from elementary drawings through repetition. The following experiments illustrate this principle.

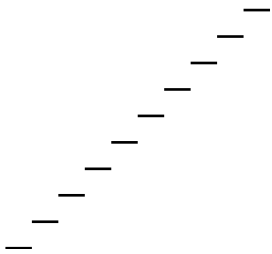
Experiment 7.6 (*A Swiss Cross)*
Draw the outline of the Swiss cross shown on the right using turnLeft: or turnRight: and timesRepeat:.



Experiment 7.7 (*A Staircase)*
Draw the staircase illustrated in the figure.

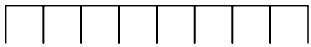


Experiment 7.8 (*A Staircase Without Risers)*
Draw the stylized staircase—with treads but without risers—illustrated in the figure.

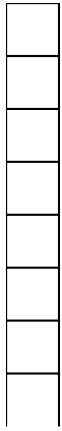


Experiment 7.9 (*A Staple)*
Draw the illustrated graphical element that looks like a staple.

Experiment 7.10 (*A Comb)*
Transform the graphical element that you produced in Experiment 7-9 to produce the comb shown in the figure.



Experiment 7.11 (*A Ladder)*
Transform the graphical element from Experiment 7-9 to produce a ladder.



Experiment 7.12 (*Tumbling Squares)*
Now that you have mastered loops using timesRepeat:,define a loop that draws the tumbling squares illus- trated at the start of Chapter 4.

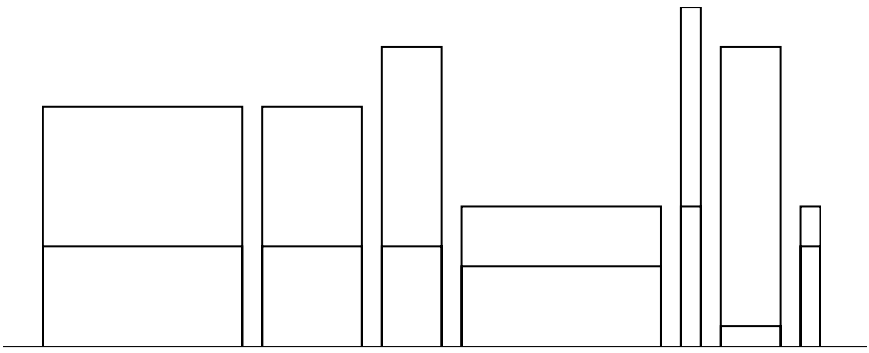
7.8 Summary

In this chapter you learned how to program loops using the method *n* timesRepeat:.

Method	Syntax	Description	Example
timesRepeat:	n timesRepeat: [a sequence of expressions]	repeats a sequence of expressions n times	10 timesRepeat: [pica go: 10. pica jump: 10]

Chapter 8

Variables



People are always giving names to things. For example, we give names to people, to dogs, and to cars. When we do this, we are *associating* some object, being, or idea with a word or a symbol. Once this association has been made, we may then use the word or symbol to *refer* to or interact with the object associated with it. A name can last a lifetime, or it can be discarded after a short period of time. Sometimes, names refer to other names. For example, an actor generally has several names: a given name, a stage name, and the name of the character that he or she is currently playing on stage or screen. In a programming language, we also need to be able to name things, and *variables* are used for this purpose.

In this chapter, you will learn about variables, which are placeholders for objects, and how variables help to simplify programs. Indeed, variables are often necessary in programming. Finally, as the complexity of the

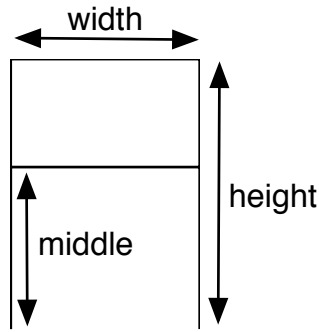


Figure 8.1: The shape of a letter A is characterized by its height, width, and midheight.

problems that you encounter increases, you will see that you will need to express dependencies between variables. For example, the width of a rectangle might be two-thirds its length. In this chapter I will show you how to use variables to express dependencies between different quantities.

8.1 Brought to You by the Letter A

As you did in Chapter 3, suppose you want to use a robot to write letters of the alphabet. The rather primitive letter A that we are going to draw is characterized by its *height*, its *width*, and a *midheight*, which isthe height at which the midline of the A should be drawn, as shown in Figure 8.1.

Experiment 8.1
Write a script that draws a letter A of height 100 pixels,width 70 pixels,and midheight 60 pixels.

Variations on the Theme of A

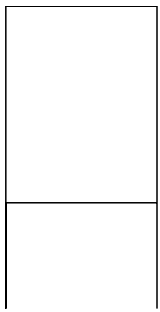
The script you wrote for Experiment 8-1 should resemble Script 8.1.

Script 8.1: *An A for Experiment 8.1.*

```
| pica |
pica := Bot new.
```

```
pica north.
pica go: 100.
pica east.
pica go: 70.
pica south.
pica go: 100.
pica west.
pica jump: 70.
pica north.
pica go: 60.
pica east.
pica go: 70
```

Experiment 8.2 (*frAnkenstein)*
Modify Script 8.1 to draw a monstrous letter A of height 200 pixels,width 100 pixels,and midheight 70 pixels, as shown in the figure below.



In modifying Script 8.1 for Experiment 8.2, you found that in order to produce an A of a different size, you had to change the numbers that represent the height, the width, and the midheight of the letter *everywhere* they occur and *synchronously*. By synchronously, I mean "in the correct order"; that is, 100 should become 200, 70 should become 100, and 60 should become 70 without any mixups.

Experiment 8.3 (*A Variety of A's)*
Modify Script 8.1 to draw other A's of different sizes of your choice.Try to reproduce some of the A's that appear in the picture at the start of this chapter.

In doing Experiment 8.3 you undoubtedly quickly came to the conclusion that changing the values of an A's characteristics everywhere is tedious. Moreover, it should be obvious that in making such changes, you run the risk of becoming confused and forgetting to change a value or making a change to an incorrect value. The result can be nothing like what you had in mind for your script. You can imagine that in complex programs, changing

values one by one in this way can become highly problematic.

8.2 Variables to the Rescue

Making large numbers of changes in producing letters of different sizes and shapes is both tiresome and error-prone, and so we need a solution that will both keep us from mixing up the numbers representing the various characteristics of a letter and enable us to make alterations without having to change all the values everywhere. In fact, we would like to be able to do the following:

- *Declare* the height, width, and midheight of a letter A once for the entire script.
- *Refer* to these values as needed.
- *Change* the values if necessary.

These three things are exactly what a variable allows us to do! Amazing isn't it? A variable is a *name* with which we *associate a value*. We must *declare* it and *associate* a new value with it. Then we can *refer* to a variable and obtain the *value* associated with this variable. It is also possible to *modify* the value associated with a variable and assign it a new value. The value of a variable value can be a number, a collection of objects, or even a robot. We now illustrate how to declare, associate a value, and use a variable.

Important! Important! A variable is a *name* with which we associate a *value*. We declare a variable and associate a value with it.Then we can *refer* to a variable and obtain its *value*.It is also possible to *modify* the value associated with a variable and associate a new value with it.

Declaring a Variable

Before using a variable, we have to *declare* it; that is, we must tell Squeak the name of the vari- able that we want to use. We declare variables by enclosing them between vertical bars ||, as shown in the following example, which declares the three variables height, width, and midheight:

```
| height width midheight |
```

To be precise, vertical bars || declare *temporary* variables, which are variables that exist only during the execution of the script.

Assigning a Value to a Variable

Before using a variable it is almost always necessary to give it a value. Associating a value is called *assigning* a value to a variable. In Smalltalk, the symbol pair := is used in combination to assign a value to a variable. In the following script, after declaring three variables we assign 100 to the variable height, 70 to the variable width, and 60 to the variable midheight. When we assign a value to a variable for the first time, we say that we are *initializing* it:

```
| height width midheight |
height := 100.
width := 70.
midheight := 60
```

Important! Important! The symbol := assigns a value to a variable. For example, height := 120 assigns the value 120 to the variable height, while length := 120 + 30 assigns the result of the expression 120 + 30, that is, 150, to the variable length.

When we assign a value to a variable for the first time,we say that we are *initializing* it.

Referring to Variables

To refer to the value assigned to a variable—we also say *use* a variable—simply write its name in a script. In the following script, after being *declared* in line 1, the variable height is *initialized* with the value 100 in line 3 and *used* in line 5 to tell the created robot to go forward the number of pixels associated with the variable height, which here is 100.

```
| pica height |
pica := Bot new.
height := 100
pica north.
pica go: height
```

Important! Important! In general, a variable must be *declared* and *initialized* before being used.

And What About Pica?

You guessed it! pica is also a variable. It just happens to be a variable whose value is a robot. Hence, | pica | declares a variable named pica. The expression pica := Bot new initializes the variable with a value, here a new robot. Then we use this robot by sending messages to it via the variable pica, for example, pica go: 100.

8.3 Using Variables

Now let us explore the benefits of using variables, and I will show you some powerful properties that variables possess. In particular, I will show you the power that comes from expressing relationships between variables.

By introducing variables into Script 8.1, we obtain Script 8.2.

Script 8.2: *An A with variables.*

```
| pica height width midheight|
pica := Bot new.
height:= 100.          "initializes the variables"
width:= 70.
midheight:= 60.
pica north.
pica go: height.       "then we use the variables"
pica east.
pica go: width.
pica south.
pica go: height.
pica west.
pica jump: width.
pica north.
pica go: midheight.
pica east.
pica go: width
```

You will agree that changing variable values once is easier than changing numbers scattered throughout the script. Change some values to convince yourself. You should be able to draw all the A's that appear in the figure at the head of this chapter. Now if you want to change the characteristics of your letter A, you need only reinitialize the variables by changing the values in lines 3, 4, and 5, as shown in Script 8.3. The resulting drawing is presented in Figure 8.2.

Script 8.3: *A modified letter A*

```
| pica height width midheight |
pica := Bot new.
height:= 30.      "initializes the variables"
width:= 200.
midheight:= 10.
...
```

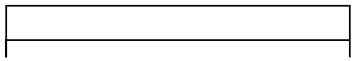


Figure 8.2: A short and stout letter A simply created with height = 30, width = 200, and midheight = 10.

Using variables, you can easily create many different letters, and in the future, you will be able to write programs to solve many challenging problems. Let us step back for a moment and consider the power provided by variables.

The Power of Variables

The experiments in the remainder of the chapter illustrate the power of variables. Variables let you name a thing, whether a robot, a length, or practically anything else. Then you can use the names instead of having to repeat the values that you associated with the names. Variables make your scripts much easier to change, since you can simply reinitialize your variables to different values.

In addition, a variable can hold a wide variety of types of values. Up to now, you have assigned robots and numbers to variables, but you can also assign colors (for example, `robotColor := Color yellow`), a sound, or indeed any Squeak object.

Note also that variables make your scripts much more readable and easier to understand. To convince yourself of this, just compare Scripts 8.1 and 8.2. The simple fact of using variables with names such as "width" and "height" helps you to understand how the letter is drawn.

8.4 Expressing Relationships Between Variables

In your experiments with the letter A, you probably found some of your letters easier to recognize than others. Letters of the alphabet should generally adhere to certain proportions to keep them readable. In particular, the

dimensions that describe a particular letter are not chosen at random, but maintain certain proportions between them.

In our simple letter A, let us decide that the width should be two-thirds of the height, and the midheight should be three-fifths of the height. We can express these relationships using variables, as shown in Script 8.4. As you can see, the value of a variable does not have to be a simple number, but can be the result of a complex computation.

Script 8.4: *Relations between variables:a first approximation.*

```
| pica height width midheight |
pica := Bot new.
height := 120.
width := 120 * 2 / 3.
midheight := 120 * 3 / 5.
```

In looking over Script 8.4, you will soon realize that it is not optimal. The relationships between the variables are expressed not between the variables themselves, but in terms of the value 120 (in lines 3, 4, and 5). This value would have to be changed manually whenever you wanted to produce a different letter A with the same proportions. You want to be able to change the value of height and have the values of width and midheight change automatically. The solution is to use the variable heightinstead of 120, as shown in Script 8-5. In this script, the values of the variables widthandmidheighttruly depend on the value of height. What makes this work is that the value of a variable can be expressed in terms of other variables. The expression width := height * 2 / 3 expresses that the width of the letter is equal to two-thirds of its height.

Script 8.5: *Relations between variables:The variables* width *and* midheight *depend on* height.

```
| pica height width midheight |
pica := Bot new.
height:= 120.
width := height * 2 / 3.
midheight := height * 3 / 5.
pica north.
...
```

Initialize Before Using!

The only constraint that you have to consider in expressing relationships between variables is that a variable used in the definition of another variable should have a value. For example, in Script 8.5, the variable height has its

initialized value 120, which is used by width and midheight in computing their initialized values. To see what can go wrong, in Script 8.6 the variable height has not been initialized, and so when an attempt is made to initialize the variable width to height * 2/3, this leads to an error, because there is no value of height to be used in the computation. I will have more to say about errors in Chapter **??**.

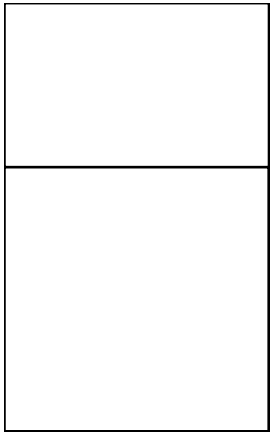
Script 8.6: *Problematic* width *initialization.*

```
| height width midheight |
width := height * 2 / 3.
height := 120.
midheight := height * 3 / 5.
```

8.5 Experimenting with Variables

The experiments that follow will help you to gain experience with variables.

Experiment 8.4 (*Golden Rectangle)*
A golden rectangle is a rectangle one of whose sides is approximately 1.6 times the length of the other.The number 1.6 is an approximation of the "golden ratio":the number. A nice property of such a rectangle is that if you cut off a square inside the rectangle,as shown in the figure below,then the part of the rectangle left over is again a golden rectangle.You can then cut a square off of this smaller golden rectangle and obtain an even smaller golden rectangle,and so on ad infinitum.The dimensions of a golden rectangle are pleasing to the eye,and since ancient times,artists and architects have used the golden ratio in their work.Write a script that draws a golden rectangle. You can express the number in Smalltalk as 1 + 5 sqrt / 2.



Experiment 8.5 (*Scripts That Don't Work)*

```
| pica height |
pica := Bot new.
height := 120.
pica north.
pica go: 100.
pica east.
pica go: 70.
pica south.
pica go: 100.
pica west.
pica jump: 70.
pica north.
pica jump: 50.
pica east.
pica go: 70
```

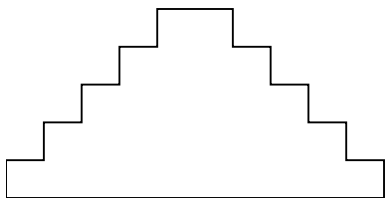
```
| pica height |
   pica := Bot new.
  pica north.
pica go: height.
  pica east.
pica go: 70.
 pica south.
 pica go: height.
  pica west.
pica jump: 70.
  pica north.
pica jump: 50.
  pica east.
pica go: 70
```

The Pyramids Rediscovered

In Script 7.1, in Chapter 7, we defined the outline of the step pyramid of Saqqara as in Script 8.1.

Script 8.1 (*The pyramid of Saqqara)*

```
| pica |
pica := Bot new.
5 timesRepeat:
[ pica north.
pica go: 20.
pica east.
pica go: 20 ].
5 timesRepeat:
[ pica go: 20.
pica south.
pica go: 20.
pica east ].
pica west.
pica go: 200.
```



Experiment 8.6 (*A Pyramid with a Variable Number of Terraces)*
Modify Script 7.1, introducing the variable terraceNumber to represent the number of terraces that the pyramid should have.

Experiment 8.7 (*A Pyramid with a Variable Number of Terraces)*
Modify the script from Experiment 8.6 by introducing the variable terraceSize to represent the size of a terrace.

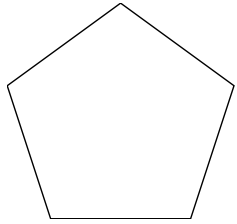
8.6 Automated Polygons Using Variables

The use of variables greatly simplifies the definition of scripts in which some of the *variables* depend on other variables. In this section, you will see how the use of variables provides great leverage in dealing with loops. Chapter 10 will go more deeply into the power that the combination of variables and loops can give to your programs.

Let us look again for a moment at Experiments 7.3 and 7.4, in which a Bot was asked to draw a regular pentagon and a regular hexagon. The requisite code appears here as Scripts 8.2 and 8.2.

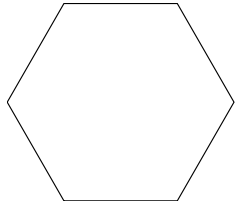
Script 8.2 (*A regular pentagon)*

```
| pica |
pica := Bot new.
5 timesRepeat:
[ pica go: 100.
pica turnLeft: 72 ]
```



Script 8.3 (*A regular hexagon)*

```
| pica |
pica := Bot new.
6 timesRepeat:
[ pica go: 100.
pica turnLeft: 60 ]
```



In order to transform the first script into the second, you must change the number of sides (let us call it s) *and* the magnitude of the turn (let us

call it T) such that the product $s * T$ is equal to 360. Wouldn't it be nice if we could write a script in which we would have to change only a single number, let us say the number of sides, since this is the easiest parameter to choose? This can be done by introducing variables. Try to come up with your own solution.

Script 8.7 solves the problem. It makes it possible to draw a regular polygon with any number of sides by changing a single number. Try it before I discuss it further.

Script 8.7: *Drawing a regular polygon.*

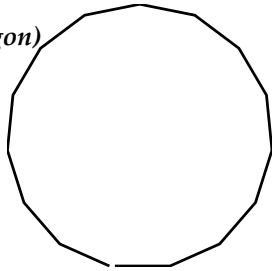
```
| pica sides angle |
pica := Bot new.
sides := 6.
angle := 360 / sides.
sides timesRepeat:
   [ pica go: 100.
   pica turnLeft: angle ]
```

This script introduces two new variables, sidesandangle, which are used to hold the number of sides and the size of the angle. Then, the expression sides := 6 assigns the value 6 to the variable sides, and the expression angle : = 360 / sides assigns a value to the variable angle, which is the result of 360 divided by the value held in the variable sides. The value of the variable angle is then used as the argument of the command turnLeft: given to the robot in the repeating block.

8.7 Regular Polygons with Fixed Sizes

You will notice that if Script 8.7 is executed with a large number of sides, the resulting figure does not fit on the screen. The next experiment asks you to fix this problem by reducing the length of the sides as the number of sides increases.

Experiment 8.8 (*Controlling the Sides of the Polygon*) Modify Script 8.7 so that the size of the regular polygon stays roughly constant as the number of sides changes. Hint:Introduce a variable totalLength that is set to a fixed length,and let each side of your polygon have length equal to totalLength divided by the number of sides.



8.8 Summary

- A variable is a *name* with which we *associate a value*. We must *declare* it and *assign* a value to it. Then we can *refer* to a variable and obtain the *value* associated with this variable. It is also possible to *modify* the value associated with a variable and assign a new value to it.
- A variable can be used at any place where its value can be used.
- The first time that we assign a value to a variable, we say that we are *initializing* it.
- The symbol := assigns a value to a variable. For example, `height := 120` assigns the value `120` to the variable `height`, while `length := 120 + 30` assigns the result of the expression `120 + 30`, that is, `150`, to the variable `length`.
- A variable must generally be *declared* and *initialized* before being used.

Chapter 9

Digging Deeper into Variables

In the previous chapter I introduced variables. In this chapter I am going to delve a bit deeper into the subject so that you can learn more about how variables are used. Since this chapter is a bit technical in nature, you might want to omit it on a first reading.

Before illustrating in detail how variables work, I want to stress again the importance of choosing good names for variables.

Naming Variables

You are free to choose practically any name for a variable. However, giving your variables meaningful names is very important, because doing so will help you both in writing your programs and in understanding the programs that you have written. To illustrate this point, read Script 9.1, which is in fact Script 8-10 rewritten using meaningless variable names.

Script 9.1: *Meaningless variable names make a program difficult to understand.*

```
| x y z|
x := Bot new.
y := 6.
z := 360 / y.
y timesRepeat:
   [ x go: 100.
   x turnLeft: z ].
```

As you may discover by trying it out, Script 9-1 is perfectly correct, and Squeak can execute it without any problem. But I am sure that you know which of the two scripts 8-10 and 9-1 is more understandable.

In Smalltalk, the name of a variable can be any sequence of alphabetic and numeric (alphanumeric) characters beginning with a lowercase letter. It is customary to use long vari- able names that clearly indicate the function of the variable in the program. Doing so helps you and other programmers to understand your scripts more easily.

Being able to understand what a program does is very important, since as you will see later, a program usually involves a combination of many scripts.[1] When the time comes that you need to understand a script written by someone else, or even one written by yourself, but perhaps many months ago, you will be glad that you adopted the habit of choosing meaningful variable names.

Now that you have been convinced of the importance of choosing meaningful names for variables, I will discuss variables in detail.

Variables as Boxes

Variables are placeholders that refer to objects. A common way to explain the notion of a variable is to use a graphical notation in which variables are represented as boxes. Let us illustrate this idea in Script 9-2 and Figure 9-1.

In Script 9-2 (step a in the script and in the figure), two variables, pica and pablo, are declared. Then in step (b) we create a robot and assign it to the variable pica; that is, the variable pica now refers to the newly created robot. Then in (c) the variable pablo is assigned the value of the variable pica, and therefore pablo now points to the same object as the variable pica, that is, to the newly created robot. When we send a message using either of the two variables, we are actually sending that message to the same object, namely the robot created in step (b), since both variables refer to the same object. Therefore, in (d), the message sent to pica causes him to move 100 pixels, while the message in (e), which addresses pablo, causes the same robot to change its color to yellow.

Another way of saying this is that the robot has two names: pica and pablo. It is just as if the artist Pablo Picasso's mother had said, "Picasso, come here" (pica go: 100), and then said, "All right, Pablo, here is your yellow shirt. Put it on" (pablo color: yellow).

[1] This is actually a simplification. Soon, you will learn about methods, which are the true building blocks of programming with objects.

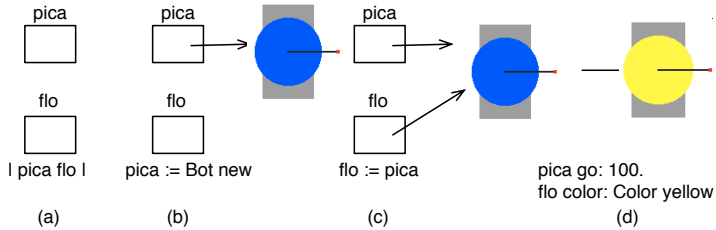


Figure 9.1: Two variables, pica and pablo, are declared. (b) A robot is created and the variable pica is associated with this new object. (c) The variable pablo is assigned the value of the variable pica, and therefore pablo now points to the same object as pica. (d) When we send the message go: 100 using pica,the robot moves. (e) When we send the message color: Color yellow to pablo, the same robot changes color. In sum,if we send a message to either of the two variables, we are actually sending that message to the same object.

Script 9.2: *Two variables point to the same robot.*

```
(a)  | pica pablo |
(b)  pica:= Bot new.
(c)  pablo := pica.
(d)  pica go: 100.
(d)  pablo color: Color yellow.
```

Assignment: The Right and Left Parts of :=

There are two very different ways that a variable name is used in a script. Sometimes, the name is used to refer to its value, as in expressions such as walkLength + 100 and pica go: 100. At other times, the variable name is used to refer to the placeholder itself in order to initialize it or change its value, as in walkLength := 100 and pica := Bot new.

The key thing to understand about variables is that using a variable name always refers to the value associated with the variable, except in the case that the variable is on the left-hand side of an assignment expression, that is, to the left of the symbol :=. In this one case, the variable name represents the placeholder itself and not the value of the variable. Another way of saying this is that the value of a variable is always read, except

when it appears to the left of :=, in which case it is written, that is, changed. Script 9.3 shows an example.

Script 9.3: *The variable* walkLength *is written in line 3 and then read in line 4.*

```
(1)  | walkLength pica |
(2)  pica := Bot new.
(3)  walkLength := 100.
(4)  walkLength + 150.
(5)  pica go: walkLength
```

In line 3 of Script 9-3, the variable name walkLength appears to the left of :=, so it refers to the placeholder, and the value 100 is assigned to the variable walkLength. After line 3 has been executed, the variable walkLength refers to the number 100. In line 4, walkLength + 150 is not part of an assignment expression, and so the variable name refers to the variable's value. (Note that in line 4, an addition is carried out, but the result of the addition is not used. Therefore, this line does not do anything and could be removed.) In line 5, both variables pica and walkLength are used to refer to their values, that is, the objects to which they refer. Therefore, the variable walkLength here refers to its value 100, while pica refers to the robot created earlier in the script. Thus line (5) has the effect that the message go: 100 is sent to the robot created the line 2.

Important! A variable is a placeholder for a value,that is, an object. Using a variable returns its value except when the variable is on the left-hand side of an assignment expression,that is,to the left of the symbol :=. In such a case,the value of the variable is changed to the value of the expression on the right-hand side of the assignment expression. For example, walkLength + 150 returns 150 added to the value of the variable walkLength, while walkLength := 100 changes the value of walkLength to 100.

Analyzing Some Simple Scripts

To understand better how variables are manipulated, I am going to describe a series of scripts. First, read the script and guess what it does; then evaluate the script carefully to determine its result. I suggest that you to draw a box representation if you think it would be helpful, and check your drawing against the one shown in the figure. As you will see, I give a small explanation of each script. Note that explanations from one script are not repeated in subsequent scripts, so go through them in order, and look back at the previous scripts for clarification of any points that are confusing.

Using two variables. In Script 9.4, the variables pica and walkLength are declared. A new robot is created and assigned to the variable pica. Then 100 is assigned to the variable walkLength. The robot (value assigned to the variable pica) receives the message go: with the value of the variable walkLength as argument, which in this case is 100.

Script 9.4: *Two variables are declared and initialized, and then their values are used (see Figure 9.2).*

```
| pica walkLength |
pica := Bot new.
walkLength := 100.
pica go: walkLength
```

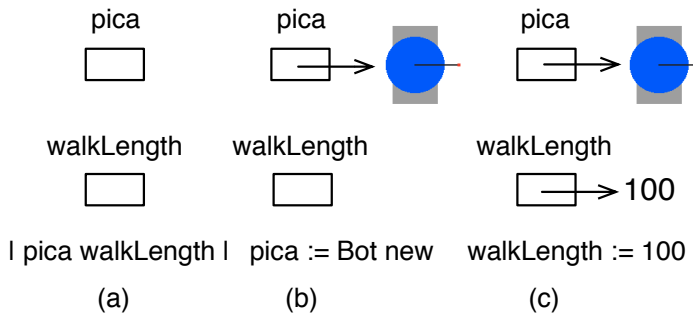


Figure 9.2: Two variables are declared and initialized, and then their values are used.

Script 9.5: *Like Script 9.4, except that a variable is used as part of an expression (see Figure 9.3).*

```
| pica walkLength |
pica := Bot new.
walkLength := 100.
pica go: walkLength + 170.
```

In Script 9.5, to determine the number of pixels that the robot should move forward, the expression walkLength + 170 is evaluated. Since walkLength was initialized to the value 100 and its value was not changed thereafter, the value of walkLength is 100. Therefore, walkLength + 170 has the value 270, and the robot moves forward 270 pixels.

Defining new variable value. The expression pica go: walkLength + 170 in Script 9.5 is equivalent to the expression pica go: walkLength2 of Script 9.6.

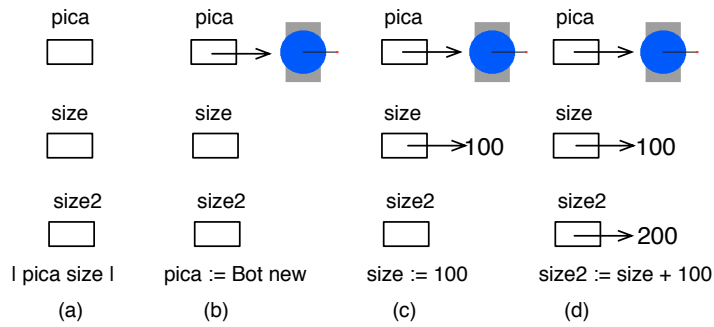


Figure 9.3: Using a new variable to hold the value of the length of pica's walk.

Indeed, the value of the variable walkLength2 is the value of the variable walkLength plus 170.

Script 9.6: *Using a new variable to hold the value of the length of pica's walk (see Figure 9.3).*

```
| pica walkLength walkLength2 |
pica := Bot new.
walkLength := 100.
walkLength2 := walkLength + 170.
pica go: walkLength2.
```

Changing the value of a variable. The value of a variable can indeed be changed using :=. In Script 9.7, first the variable walkLength is declared, then we assign 100 to it, and then we assign 300 to it (the dashed arrow in the figure pointing to 100 indicates that the variable no longer points to 100). When the variable is then used in the last expression of the script, its value is 300. As a result, the robot moves forward 300 pixels.

Script 9.7: *Changing* walkLength *twice (see Figure 9.4).*

```
| pica walkLength |
pica := Bot new.
walkLength := 100.
walkLength := 300.
pica go: walkLength
```

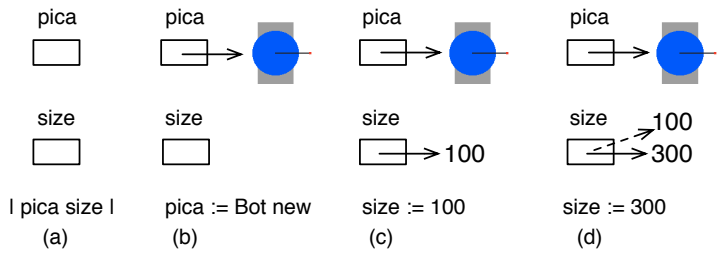


Figure 9.4: Changing walkLength value twice.

Using a variable without assigning it a value has no effect on its value. Script 9.8 shows that using the value of a variable in any way other than assigning it a new value has no effect on the variable's value. The only way to change the value of a variable is with an assignment expression. In Script 9.8, the variable walkLength is initialized with the value 100. Then the value of walkLength, here 100, is added to 200, but no assignment expression is involved, and so the variable's value is not modified. And so when the value of walkLength, which here is 100, is used in the last statement to specify how far the robot should move forward, the robot moves 100 pixels.

Script 9.8: *Using a variable without assigning it a value has no effect on its value. (see Figure 9.5).*

```
| walkLength pica |
pica := Bot new.
walkLength := 100.
walkLength + 200.
pica go: walkLength
```

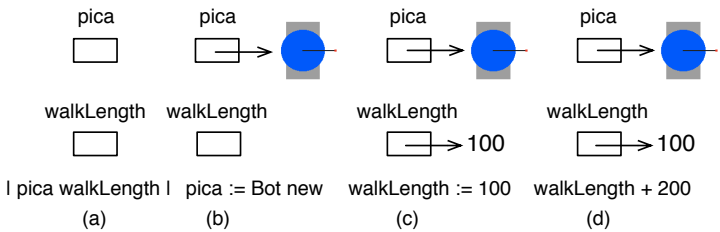


Figure 9.5: Using a variable without assigning it a value has no effect on its value.

Using the value of a variable to define its own value. In Script 9.9, the variable walkLength is initialized to 100. Then its value is changed to the value of the expression walkLength + 50. At this point, the value of walkLength is 100, so the expression walkLength + 50 returns 150, and then the value 150 is assigned to the variable walkLength. So in the last step, the robot moves forward 150 pixels. It is important to note here that in the expression walkLength := walkLength + 50, the variable name walkLength is used in two different ways: first, the expression to the right of := is evaluated, in which walkLength represents a value, and then the result of that evaluation is assigned to the variable walkLength to the left of :=, which on this side represents the variable as a placeholder.

Script 9.9: *The value of the variable* walkLength *is used to define the variable itself. (see Figure 9.6).*

```
| pica walkLength |
pica := Bot new.
walkLength := 100.
walkLength := walkLength + 50.
pica go: walkLength
```

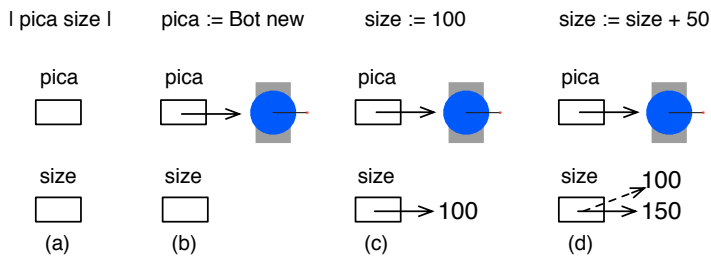


Figure 9.6: The value of the variable walkLength is used to define the variable itself.

Using twice the same variable to change its own value. In Script 9.10, the variable walkLength is initialized to 150. Then the value of walkLength is reassigned to refer to the value of the expression walkLength + walkLength. In computing the value of the expression walkLength + walkLength, the value of walkLength is 150. Therefore, the expression returns 300, which becomes the new value of the variable walkLength. And so the robot moves forward 300 pixels.

Script 9.10: *Using twice the same variable to change its own value (see Figure 9.6).*

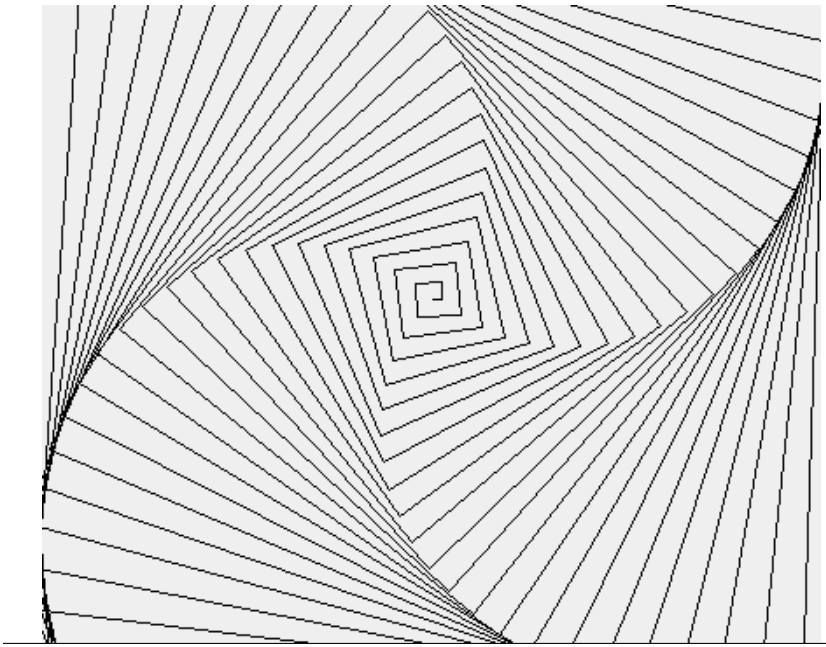
```
| pica walkLength |
pica := Bot new.
walkLength := 150.
walkLength := walkLength + walkLength.
pica go: walkLength
```

Summary

- A variable is an object that serves as a placeholder for a value. You can think of a variable as a box referring to an object.
- Using a variable returns its value except when the variable is on the left of an assignment expression :=. In such a case its value changes to become the value of the expression on the right of the assignment expression :=. For example, walkLength + 100 returns 100 added to the value of the variable walkLength. On the other hand, walkLength := 100 changes the value of walkLength to be 100.

Chapter 10

Loops and Variables



In this chapter I will show you how to use variables and loops in combination. We will begin by analyzing a simple problem that illustrates the need for using variables with loops. Then you will experiment with some other problems.

10.1 A Bizarre Staircase

Try to program a robot to draw the strange Staircase shown in Figure 10.1. All of the risers have the same height, but the treads get longer and longer as you climb the staircase. One way to start is to write a script for a normal stairway and then modify it. You will need to solve the problem of making each tread longer than the previous one.

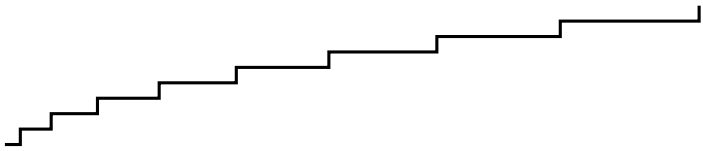


Figure 10.1: Pica draws a bizarre staircase.

A simple solution is shown in Script 10.1, where the length of each tread grows by 10 pixels. However, such a solution is not satisfactory for two reasons: First, you have to compute the length of each tread manually. And second, you have to repeat an almost identical sequence of message sends over and over.

Script 10.1: *Pica draws the bizarre staircase.*

```
| pica |
pica := Bot new.
pica go: 10.
pica turnLeft: 90.
pica go: 5.
pica turnRight: 90.
pica go: 20.
pica turnLeft: 90.
pica go: 5.
pica turnRight: 90.
pica go: 30.
pica turnLeft: 90.
pica go: 5.
pica turnRight: 90.
pica go: 40.
pica turnLeft: 90.
pica go: 5.
pica turnRight: 90.
...
```

We would like to be able to use the power of variables (to automate the increasing tread length) combined with the power of loops (so that we don't have to type so much code). To avoid repeating the sequence of message sends you can use the timesRepeat: message. And as for using variables, the key is to be found in an examination of Script 10.1, where you will see that the length of each tread after the first is the length of the previous tread plus 10 pixels. After all, 20 = 10 + 10, 30 = 20 + 10, 40 = 30 + 10, and so on, as shown in Figure 10.2.

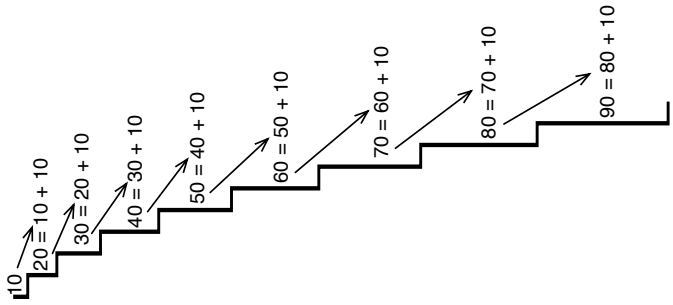


Figure 10.2: The length of a tread is the length of the previous tread plus 10 pixels.

Let us use the variable treadLength to represent the length of a tread. Then, once treadLength has been initialized to the length of the first tread, the expression treadLength := treadLength + 10 will set the length of the *next* tread to be the value of the *current* tread increased by 10. The result is that if treadLength is initialized to 10, and the first tread is drawn, that tread will of course have length 10. Then, after the expression treadLength := treadLength + 10 is executed, the next time a tread is drawn it will have length 20. And after the expression is executed again, the next tread will have length 30, and so on.

Let's combine everything! We will start with the script of a normal staircase (Script 10.2). Then, in Script 10.3, we obtain the same staircase, but using the variable treadLength. Then in Script 10.4, we add the line treadLength := treadLength + 10 to change the treadLength value in each step of the loop, and thus we finally obtain the stairway we want.

Script 10.2: *A stairway with normal steps.*

```
| pica |
pica := Bot new.
10 timesRepeat:
```

```
  [ pica go: 10.
  pica turnLeft: 90.
  pica go: 5.
  pica turnRight: 90 ]
```

Script 10.3: *A stairway with normal steps using the variable* treadLength.

```
| pica treadLength|
pica := Bot new.
treadLength := 10.
10 timesRepeat:
  [ pica go: treadLength.
  pica turnLeft: 90.
  pica go: 5.
  pica turnRight: 90 ]
```

Script 10.4: *The solution: increasing the variable* treadLength *each time through the loop produces the bizarre staircase.*

```
| pica treadLength|
pica := Bot new.
treadLength := 10.
10 timesRepeat:
  [ pica go: treadLength.
  pica turnLeft: 90.
  pica go: 5.
  pica turnRight: 90.
  treadLength := treadLength + 10]
```

Let's look more closely at the sequence of expressions in the loop in Script 10.4. The first expression draws a tread by making the robot go forward a distance given by the value of the variable treadLength (which has been initialized to 10 for the first time through the loop). Then the robot turns, draws a riser (straight up), and turns again. Then the value of the variable treadLength is increased by 10 and the loop restarts, but now the variable treadLength has a new, larger, value (20 for the second repetition). The whole process is executed 10 times.

The expression treadLength := treadLength + 10 is absolutely necessary. Without it, the value of the variable would never change.

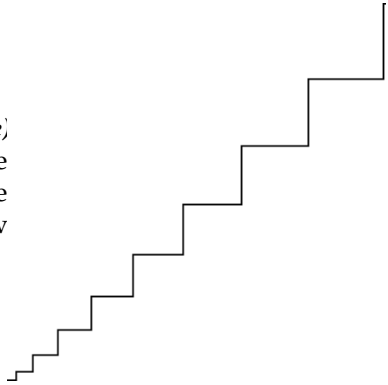
Experiment 10.1 (*Placement of the Increment in the Loop)*
Try changing the last line of the loop; for example, treadLength := treadLength + 15. Then try moving the line to different places in the loop. Can you explain what happens when you move the last line of the loop to the beginning of the loop?

If you are still uncertain about what is going on in Script 10.4, I suggest that you think carefully about the value of the variable treadLength, especially at the beginning and end of the loop. Figure out the value of treadLength in each expression for three repetitions of the loop. If necessary, read Chapter 8 again.

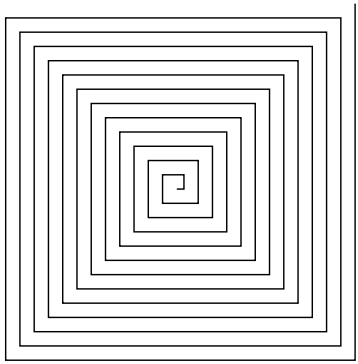
10.2 Practice with Loops and Variables: Mazes, Spirals, and More

Let's see how combining variables and loops can help you to solve some other problems.

Experiment 10.2 (*Another Bizarre Staircase*)
Modify Script 10.4 to produce the picture shown below, which represents a staircase in which both the treads and the risers grow in size.



Experiment 10.3 (*A Simple Maze)*
Write a script that reproduces the drawing shown below. In addition,by changing the angle through which the robot turns,you should be able to re-create the picture shown at the beginning of this chapter, as well as the spiral shown in Figure 10.3.



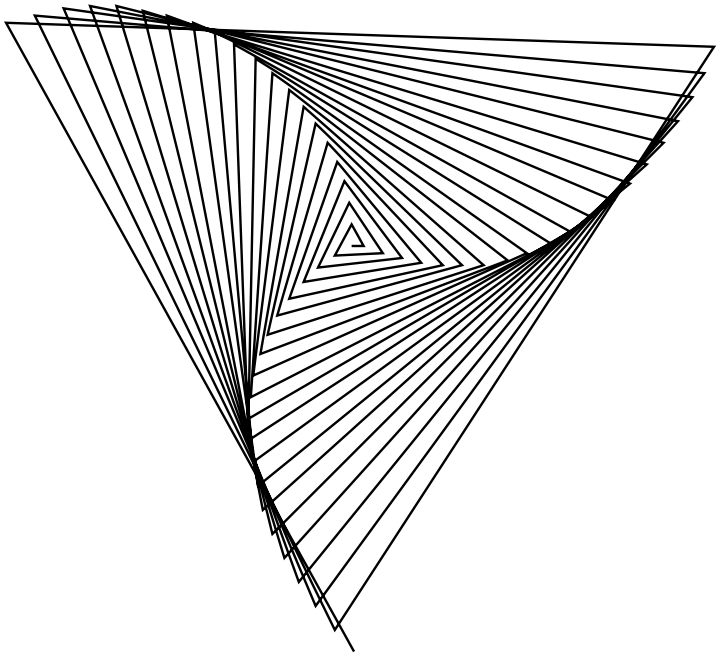
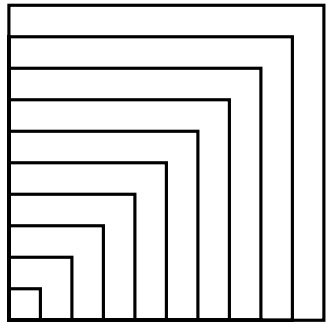
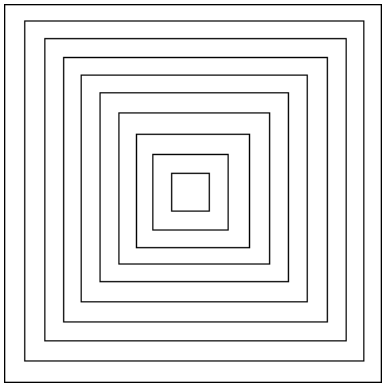


Figure 10.3: A nice spiral.

Experiment 10.4 (*Russian Squares)*
Draw the nested squares of different sizes as shown in the figure below. You might begin by defining a loop that draws the same square ten times. Then introduce a variable sideLength to represent the side length of a square, and finally,make the side length grow each time through your loop. As an additional challenge, you might try to write a new script that draws the same figure without the robot having either to jump or draw any line twice.



Experiment 10.5 (*A Long Corridor)*
The concentric (having a common center) squares of different sizes shown in the figure below represent a long corridor, which seems to get smaller as you look into the distance. Again start by drawing a square, but this time draw it starting from its center (you will need a jump message), so that when you change the square's size, the next square will automatically be drawn concentric to the first one. Now define your square inside a loop, and then introduce a variable sideLength representing the side length of the square. Finally, make the square grow each time through your loop.



10.3 Some Important Points for Using Variables and Loops

Now that you have seen the overall process of combining loops and variables and have experimented a bit, I would like to stress some important points. Script 10.5 shows a typical situation in outline form: First a variable is declared. Then it is initialized. Inside the loop, the variable is used to perform some computations, and then its value is changed for the next pass through the loop.

Script 10.5: *An outline script showing the use of a variable in a loop.*

```
| treadLengthpica |              "variable declaration"
...
treadLength := 10.               "initialization of the variable"
...
10 timesRepeat:
[ pica go: treadLength.        "variable use"
...
treadLength := treadLength + 10]   "variable change of value"
```

10.4 Variable Initialization

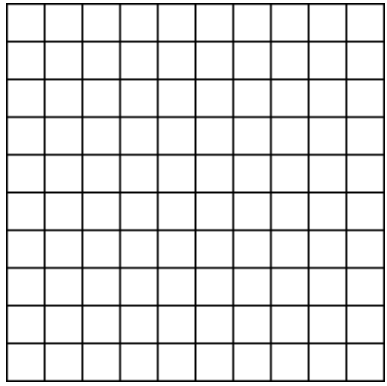
When you introduce a variable in a loop, you have to pay special attention to the first value of the variable, that is, the value assigned to the variable when it is initialized. Keep in mind that a variable cannot be used until it has been initialized. Normally, variable initialization is done outside of the loop, for otherwise, the variable's value would be reinitialized at each step of the loop, with the result that the value of the variable would not change.

10.5 Using and Changing the Value of a Variable

Inside the loop, the variable's value is often used to perform a variety of computations, such as to compute the values of other variables or perhaps to tell a robot how far to travel. Then the value of the variable is eventually changed. In the stairway example, the expression `treadLength := treadLength + 10` increases the value of the tread length based on its preceding value. What is important to understand is that the new value assigned to the variable will be its value for the next step of the loop, as illustrated in Figure 10.4.

10.6 Advanced Experiments

Experiment 10.6 (*Squares)*
Define a script that creates the checkerboard construction shown below. This experiment is a bit more complicated than the earlier experiments, in that it is difficult to see how to draw the figure using a single loop. However, there are several ways of solving the problem using two loops. For example, you could use one loop to draw all the horizontal lines, and then another loop to draw all the vertical lines.



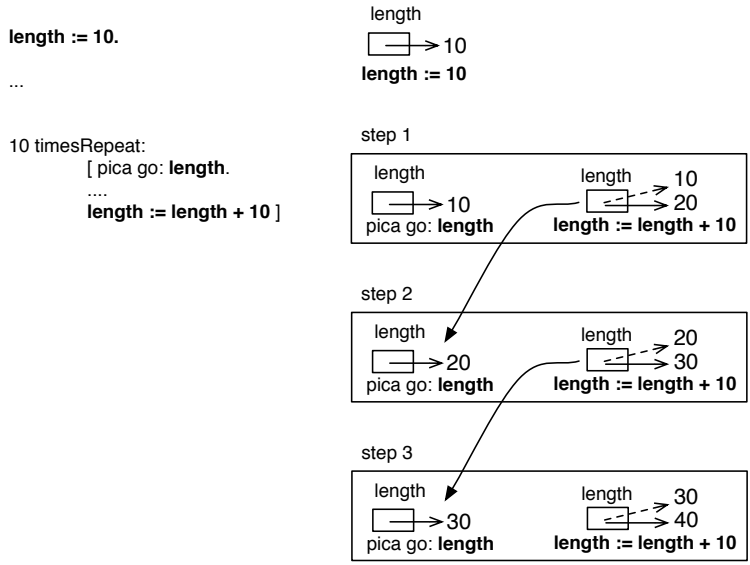
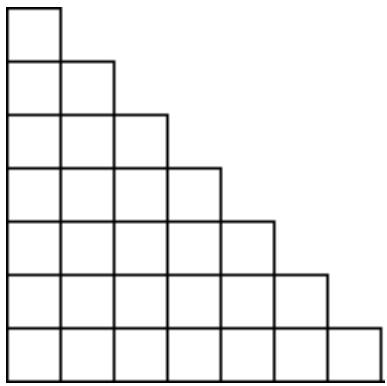


Figure 10.4: The length of a tread is the length of the previous tread plus 10 pixels. The last value that a variable has in the loop body is the value that the variable will have at the start of the loop when it is repeated.

Experiment 10.7 (*Pyramid)*
Write a script that creates the construction shown in the figure below,representing the blocks of stones forming part of the step pyramid of Saqqara, which you have seen in previous chapters. You might modify your script from Experiment 10.6 by changing the variable for the length of the lines drawn each time through the loop.



10.7 Summary

- When introducing a variable inside a loop, be careful how you initialize the variable. Keep in mind that a variable must be initialized

before its value is used. Normally, initialization occurs outside the loop, for otherwise, the value of the variable would not change when the loop is repeated.

- Keep in mind that the last value that a variable is assigned in a loop body will be the variable's value the next time the loop is executed.

Chapter 11

Composing Messages

As with any language, Smalltalk follows certain rules in executing the messages sent to objects. I have not yet presented these rules to you, and you may have wondered just what those rules are when you were experimenting with the previous scripts. Your patience will be now rewarded. This chapter explains how to read and write correctly formulated messages. This chapter may appear a bit more difficult or abstract than the previous ones. However, I have done my best to present clearly the simple rules that govern the writing of messages. Understanding such details might not be as much fun as playing with robots, but it is the price that must be paid if you are to be able to write more advanced programs. The good news is that Smalltalk is not a complex language: there are only five rules that you need to under- stand. If you are still hesitant, you may also skip this chapter on a first reading and return to it when you have questions about the structure of your programs.

As described in Chapter 2, a message is composed of the *message selector* and the optional *message arguments*. A message is sent to a *message receiver*. The combination of a message and its receiver is called a *message send*.

When you write a complex expression such as `pica go: 100 + 20`, it contains two message sends, using the message selectors `go:` and `+`, and you have to know the order in which the messages are executed if you are to understand what the result of the entire expression will be. In Smalltalk, the order in which messages are executed is determined by the type of message send. There are three types of messages: *unary*, *binary*, and *keyword-based*. Unary messages are always sent first, followed by binary messages, and finally keyword-based messages. Any messages enclosed in parentheses are executed prior to any other messages. This means that you can change the order in which message sends are executed through the use of parenthe-

ses. These rules go a long way toward making Smalltalk code easy to read. And you will soon discover that most of the time, you do not even have to think about them. However, you have to know them, because occasions will arise that will require your knowledge of them.

All of the examples presented in this chapter consist of executable code, as shown in the text. So do not hesitate to try them out and see how they function. I will begin by showing you how to identify the different types of messages, and then I will present some examples of each type. Finally, I will present the rules for message composition.

11.1 The Three Types of Messages

Smalltalk defines a small number of simple rules to determine the order in which message sends are executed. These rules, which I will present in detail later, are based on the distinction among three different types of messages:

- *Unary messages* are messages with no arguments. They are sent to an object (the message receiver) without any other information. For example, in the expression `pica color`, the message color is a unary message. It does not send any additional information; that is, there is no argument. These messages are called "unary," from the Latin *unum*, meaning "one," because a message send with a unary message involves only *one object*, the message receiver.

- *Binary messages* are messages that involve two objects: the message receiver and the message selector's sole argument. (The word "binary" is related to the Latin *bis*, mean- ing "twice.") Binary messages are mainly related to mathematical expressions. For example, in the message send `10 + 20`, the message consists of the message selector + together with the single argument 20. The message `+ 20` is sent to the object 10, which is the message receiver.

- *Keyword-based messages* are messages whose message selector contains a keyword with at least one colon character : in its name and that has one or more arguments. For example, in the message send `pica go: 100`, the keyword `go:` contains the colon character: and there is one argument, `100`, and therefore there are two objects involved in the message send: the message receiver `pica` and the argument `100`.

Important! Don't be confused by the nomenclature for unary and binary messages. The idea of "one" in the word *unary* and the idea of "two" in the word *binary* refer to the number of objects involved in a message send, not the number of arguments. Thus a unary message has no arguments, and so a unary message send involves one object, namely,the message receiver. A binary message has a single argument,and therefore a binary message send involves two objects: the argument of the message selector and the message receiver.

11.2 Identifying Messages

In order to understand the structure of an expression, the first thing you need to do is to identify the messages and their receivers. To do this, I suggest that you use a graphical notation as shown in Figure 11.1. In all the figures of this chapter, the message receivers are underlined, each message is surrounded by an *ellipse*, and the messages that make up the expression are numbered in the order in which they will be executed. When there is more than one ellipse, the first ellipse to be executed is drawn with a dashed line so that you can see at once where to begin.

Figure 11.1 shows that the expression pica color: Color yellow contains within it the expression Color yellow. Therefore, there are two ellipses, one for the entire expression pica color: Color yellow, and one for the subexpression Color yellow. You will learn a bit later that the expression Color yellow is executed first, so its ellipse is a broken line and is numbered 1.

Note that each message has a receiver. The robot pica receives the message color: ..., and Color receives the message yellow. Therefore, these two message receivers are underlined. (The three dots in the message color: ... indicate the argument of the message selector color:, which will be the result of the message send Color yellow.)

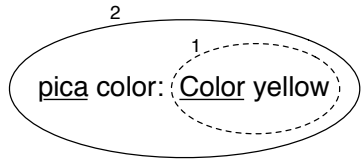


Figure 11.1: The expression pica color: Color yellow contains the subexpression Color yellow. The message yellow is sent to Color, and then the message color: ... is sent to pica.

As I have mentioned, every message is sent to an object called the message receiver. A receiver does not have to be a robot. It can be just about anything, from a number to a window. A message receiver can appear explicitly as the first element of an expression, such as pica in the expression pica go: 100 or Color in the expression Color new. However, a receiver can also be the result of other message sends. For example, in the message Bot new go: 100, the receiver of the message go: 100 is the resulting object returned by the message send Bot new. Nevertheless, in every case, a message is sent to some object, and that object is called the message receiver.

Important! In a message send, a message is always sent to an object, called the message receiver, which may be named explicitly or may be the result of other message sends.

As I have mentioned, every message is sent to an object called the message receiver. A receiver does not have to be a robot. It can be just about anything, from a number to a win- dow. A message receiver can appear explicitly as the first element of an expression, such as pica in the expression pica go: 100 or Color in the expression Color new. However, a receiver can also be the result of other message sends. For example, in the message Bot new go: 100, the receiver of the message go: 100 is the resulting object returned by the message send Bot new. Nevertheless, in every case, a message is sent to some object, and that object is called the message receiver.

Table 11.1 shows some message sends, and I suggest that in each case, you identify the type of message and then draw the graphical representation of the expression.

From Table 11.1 you should observe the following points:

- Some of the messages have arguments, while others do not. The message east, being unary, does not have an argument, while go: 100 and + 20 each have one argument, the number 100 and the number 20, respectively.

- Different messages are sent to different objects. In the expression pica east, the message east is sent to a robot, and in 100 + 20, the message + 20 is sent to the number 100.

- There are simple messages and compound messages. For example, Color yellow and 100 + 20 are simple: One message is sent to one object. On the other hand, the expression pica go: 100 + 20 contains two messages: + 20 is sent to 100, and then go: ... is sent to pica, where ... represents the result of the execution of the message send 100 + 20.

Table 11.1: *Some Examples of Message Sends, Simple and Compound.*

Expression	Type(s)	Action
pica go: 100	keyword-based	The receiving robot moves forward 100 pixels.
100 + 20	binary	The number 100 receives the message with the number 20 as argument.
pica est	unary	The receiving robot pica points to the east.
pica color: Color yellow	keyword-based, unary	The receiving robot pica changes its color to the color yellow.
pica go: 100 + 20	keyword-based, binary	The receiving robot moves forward 120 pixels.
Bot new go: 100	unary, keyword-based	The message new is sent to the Bot class, which returns a new robot to which the message go: 100 is sent, causing the robot to move forward 100 pixels.

- A message receiver can be the result of an expression that returns an object. In the expression Bot new go: 100, the message go: 100 is sent to the object (a robot) that results from evaluation of the expression Bot new.

11.3 The Three Types of Messages in Detail

Now that you are able to identify message receivers and the three types of messages, let us look at the different types of messages in detail.

Unary Messages

Among the uses of unary messages are obtaining a value from an object, such as the size of a robot's pen (pica penSize); obtaining an object from a class (Color yellow); and instructing the receiver to perform an action (pica beInvisible). Recall that a unary message does not take an argument: it is sent to the receiver without any other information. Thus the one object involved in a unary message send is the message receiver. Unary message sends are

of the form of receiver messageName. Script 11.1 presents some examples of unary messages, shown in boldface type.

Script 11.1: *Examples of unary messages*

```
| pica |
pica := Bot new.
pica color.
pica penSize.
pica east.
Color yellow.
125 factorial
```

Important! Unary messages are messages that do not take an argument. A unary message send has the form receiver messageName.

Binary Messages

Binary messages are messages that involve two objects: the receiver and a single argument. All binary message selectors are composed of one or two characters from the following list: +, *, /, |, &, =, >, <, ~, @. Therefore +, =, and * are message selectors, but so is =>, which is composed of two symbols.

Table 11.2 shows some examples of binary message sends and their meaning. At this point, I would rather not go into the details of these examples, so don't worry if you are not sure about exactly what each of these message selectors does. But try executing the expressions and others like them.

Important! Binary message sends involve two objects:the receiver and a single argument.The mes- sage selector of a binary argument is composed of one or two characters from the following list: +, *, /, |, &, =, >, <, ~, @. Binary message sends have the form receiver messageName argument.

Keyword-Based Messages

Keyword-based messages are messages that take at least one argument and that contain at least one colon character :. Note that the colon is part of the message selector. Therefore, go:, not go, is the name of a keyword-based

Table 11.2: *Examples of Binary Messages with Numbers.*

Expression	Returned Value	Action
1 + 2.5	3.5	Addition of two numbers
3.4 * 5	17.0	Multiplication of two numbers
8 / 2	4	Division of two numbers
10 - 8.3	1.7	Subtraction of two numbers
12 = 11	false	Testing for equality between two numbers
12 ~= 11	true	Testing for inequality between two numbers
12 > 9	true	Is the receiver greater than the argument?
12 >= 10	true	Is the receiver greater than or equal to the argument?
12 < 10	false	Is the receiver less than the argument?
100@10	100@10	Create a point with coordinates (100, 10)

message. Script 11.2 shows some examples of keyword-based messages, shown in boldface type.

Script 11.2: *Examples of keyword-based messages*

```
| pica |
pica := Bot new.
pica go: 100.
pica penSize: 5.
pica color: Color yellow.
pica turn: 90
```

I have said that a keyword-based message has at least one argument, but we have not yet seen an example of such a message with multiple arguments. Let us look at an example now: The message send `aNumber between: lowerBound and: upperBound`checks whether the number `aNumber` is in the interval represented by the two numbers `lowerBound` and `upperBound`. This message needs two arguments, namely, the two bounds of the interval. An example is shown in Table 11-3. Note that the message selector is actually `between:and:`. It is composed of the two words `between:` and `and:`.

Table 11.3: *Keyword-Based Messages That Take More Than One Argument.*

Expression	Arguments	Returned Value	Action
5 between: 2 and: 10	2, 10	true	Is 5 between 2 and 10?
Color r:0g:1b:0	0, 1, 0	a green color object	creates a color with the given values of red, green, and blue.

Important! Keyword-based messages have at least one argument, and their message selector contains at least one colon character :. A keyword-based message send that takes two arguments is of the form receiver messageNameWordOne: argumentOne messageNameWordTwo: argumentTwo.

11.4 Order of Execution

You have seen that there are three kinds of messages: unary, binary, and keyword-based. Now I will tell you, as I promised, how to determine the order in which messages are executed. The order of message execution is determined by the type of message, as described by the follow- ing three rules:

Rule 1: Unary message sends are executed first, then binary message sends, and finally keyword-based ones.

Rule 2: As with mathematical expressions, the priority of message execution can be over- ridden by parentheses: message sends in parentheses are executed before any other types of message sends.

Rule 3: Message sends of the same type are executed from left to right.

These rules may seem complex, but they are quite natural, and once you get used to them, you will not have to think too much about them most of the time. In particular, the third rule simply states that messages of the same type are executed in the order in which they are read.

If you are ever in doubt and want to be sure that your messages are executed the way you want, you can always add extra parentheses, as shown in Figure 11.2.

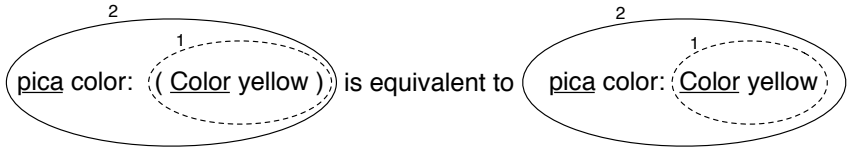


Figure 11.2: Unary message sends are executed first, so Color yellow is executed first. This execution returns a color object,which is passed as the argument of the message color: ... that is sent to pica.

In the figure, the expression pica color: Color yellow is analyzed. The message selector yellow is a unary message, while the message selector color: is a keyword-based one. Therefore, the expression Color yellow is executed first. If you are unsure about the order of execution, then you can put parentheses around Color yellow to make sure that it will be executed first. This won't change the natural order of execution. That is, pica color: Color yellow and pica color: (Color yellow) have precisely the same effect. The rest of this section illustrates each of these points.

11.5 Rule 1: Unary > Binary > Keywords

Unary message sends are executed first, then binary message sends, and finally keyword- based message sends. In programmer jargon we also say that unary messages have precedence over binary messages, and binary messages have precedence over keyword-based messages.

Important! Rule 1:Unary message sends are executed before binary message sends,which are executed before keyword-based message sends.

Example 1

In the message send pica color: Color yellow, there is one unary message, yellow, sent to the class Color, and there is one keyword-based message, color: ..., which is sent to the robot pica. Unary message sends are executed first, so the first message send to be executed is Color yellow. This execution returns a color object, represented as aColor (since we need a name to refer to it), which is passed to pica as the argument of the message color: aColor. Figure 11.2 shows graphically the order in which the messages are executed.

As an aid to your understanding, I would like to propose a textual way of representing a compound message send in step-by-step execution. In Step-by-step 11.3, the message send to be executed step by step is pica color: Color yellow. The first line shows the complete message send in boldface type.

Script 11.3: *Step-by-step decomposition of the execution of pica color: Color yellow*

```
pica color: Color yellow
(1)        Color yellow       "unary"
–returns> aColor
(2) pica color: aColor "keyword–based"
```

The lines of code represent the execution steps in numbered order in which they will occur. Thus Color yellow is the first expression to be executed. Note that the expressions have been indented to line up with their counterparts in the message send at the top.

When the execution of a message send returns a result that is used in the following execution, the line following the executed expression shows "-returns>" followed by the result. Here the expression Color yellow returns a color object that I have called aColor so that it can be referred to in the sequel. The second expression to be executed is pica color: aColor, where as I just explained, aColor is the result obtained from the previous execution step. To stress this point, the returned value or object is displayed in italic type. For further clarification, the kind of message that is currently being executed is displayed as a comment inside quotation marks. For example, Color yellow is shown to be a unary message.

Example 2

The message send pica go: 100 + 20, contains the binary message selector + and the keyword-based message selector go:. Binary messages are executed prior to keyword-based messages, so 100 + 20 is executed first: The message + 20 is sent to the object 100, which returns the number 120. Then the message pica go: 120 is executed with 120 as argument. Step-by-step script 11.4 shows how the expression is executed (Figure 11.3).

Script 11.4: *Step-by-step decomposition of the expresssion pica go: 100 + 20*

```
pica go: 100 + 20
(1)        100 + 20      "binary"
–returns> 120
(2) pica go: 120          "keyword–based"
```

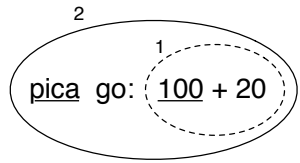


Figure 11.3: Decomposition of the expresssion pica go: 100 + 20.

Example 3

The message pica penSize: pica penSize + 2 contains the unary message penSize, the binary message with selector +, and the keyword-based message with selector penSize:. Step-by-step script 11.5 illustrates the decomposition of message execution. The unary message send pica penSize is executed first (step 1). This message returns a number, which we are calling aNumber, representing the current size of the receiver's pen. Then the binary message send aNumber + 2 is executed (step 2). The number aNumber is the receiver of the message + 2, which in turn returns another number, the sum, which here is called anotherNumber. Finally, the keyword-based message penSize: anotherNumber is sent to pica, who sets his pen size to anotherNumber (see Figure 11.4).

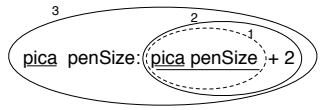


Figure 11.4: Decomposition of the expresssion pica penSize: pica penSize + 2.

Script 11.5: *Step-by-step decomposition of the expresssion pica go: 100 + 20*

```
pica penSize: pica penSize + 2
(1)          pica penSize       "unary"
–returns> aNumber
(2)          aNumber+ 2         "binary"
–returns> anotherNumber
(3) pica penSize: anotherNumber "keyword–based"
```

Altogether, the entire compound expression increases the receiver's pen size by two pixels. It does so by first asking pica for his pen size (pica penSize), increasing that number by two (aNumber+ 2), and then telling pica to change his pen size to the new number (pica penSize: anotherNumber). Note that penSize and penSize: are two different message selectors! The first is unary

and asks the receiver for its pen size, and the second is keyword- based and tells the receiver to change its pen size to the value of its argument.

Example 4

As an exercise, I will let you decompose the execution of the message `Bot new go: 100 + 20`, which is composed of one unary, one keyword-based, and one binary message (see Figure 11.5).

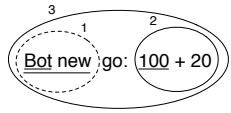


Figure 11.5: Decomposition of the expresssion `pica new go: 100 + 20`.

11.6 Rule 2: Parentheses First

The default ordering of message execution may not be suitable for what you want to accomplish in an expression, and so you should be able to change it. For this purpose, Smalltalk offers parentheses (and). Just as in mathematics, expressions in parentheses get the highest precedence, and they are executed before any others.

Keep in mind that if you find the rules for order of execution a bit complex or if you sim- ply want to clarify the structure of an expression, use parentheses to ensure that the messages are executed in the order that you wish. Figure 11.6 shows some of the expressions that we have previously looked at together with their equivalents using parentheses.

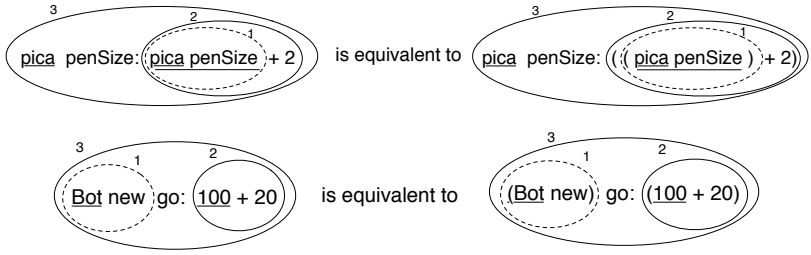


Figure 11.6: Equivalent messages using parentheses.

Important! Rule 2: As in mathematics, messages in parentheses are executed before any others. They have the highest priority.

Example 5

The message (65 @ 325 extent: 134 @ 100) center returns the center of a rectangle whose upper-left point has coordinates (65, 325), whose width is 134 pixels, and whose height is 100 pixels. Step-by-step 11.6 shows how the message is decomposed and executed. First, the message within the parentheses is executed. It is a compound expression consisting of three message sends: two binary message sends, 65 @ 325 and 134 @ 100, which are executed first and return points; and the keyword-based message extent: ..., which is then sent to the point (65, 325) and which returns a rectangle with upper-left point and dimensions as described above. Finally, the unary message center is sent to this rectangle, and a point, the rectangle's center, is returned. Try- ing to evaluate the message without parentheses leads to an error, because in that case, the unary message center would have to be executed first, and it would be sent to the object 100, but a number object does not understand the message center.

Script 11.6: *Step-by-step decomposition of the expresssion* pica go: 100 + 20

```
(65 @ 325 extent:  134 @ 100) center
(1)  65@325                          "binary"
–returns> aPoint
(2)              134@100             "binary"
–returns> anotherPoint
(3)  aPointextent: anotherPoint "keyword–based"
–returns> aRectangle
(4)  aRectanglecenter                 "unary"
–returns> 132@375
```

11.7 Rule 3: From Left to Right

Now that you know how messages are categorized according to priority of execution, the final question to be addressed is how messages with the same priority are executed. Rule 3 states that they are executed from left to right. This rule was used already, in Step-by-step 11.6, where the left-hand binary message @ 325 was executed before the right-hand message @ 100.

Important! Rule 3: Messages of the same type are executed in order from left to right.

Example 6

In the expression Bot new east, both message sends are unary messages, so the first one as you read from left to right, Bot new, is executed first. It returns a newly created robot, called aBot in Step-by-step 11.7 to which the second message, east, is sent. Figure 11.7 shows the order of execution.

Script 11.7: *Step-by-step decomposition of the expresssion* Bot new east

```
Bot new east
 (1) Bot new              "unary"
     –returns> aBot
 (2)       aBot east      "unary"
```

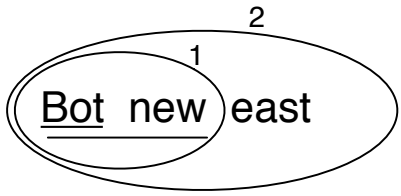


Figure 11.7: Decomposition of the expression Bot new east.

Example 7

In the expression 20 + 2 * 5, there are only the two binary message selectors + and *. According to Rule 3, since + is to the left of *, it should be executed first. In normal mathematical notation as well as in many programming languages, multiplication would take precedence over addition regardless of the order in which the arithmetic operations appear. However, in Smalltalk there is *no specific priority* for mathematical operations. The message descriptors + and * are just binary messages, and therefore they have equal status. The message selector * does not have precedence over +, and the leftmost message selector + is sent first, and then * is sent to the result, as shown in Step-by-step 11.8 and Figure 11.8.

Script 11.8: *Step-by-step decomposition of the expression* 20 + 2 * 5

20 + 2 * 5

```
(1)  20 + 2
  –returns> 22
(2)  22 * 5
  –returns> 110
```

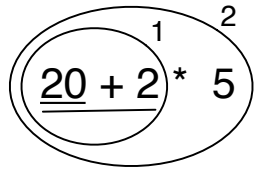


Figure 11.8: Decomposition of the expression `20 + 2 * 5`.

Important! There is no priority among binary messages. In the expression `20 + 2 * 5`, the leftmost message `+` is evaluated first, despite the fact that in normal mathematical notation, multiplication takes precedence over addition.

You can see, then, as shown in Step-by-step 11.8, that the result of this expression is not 30, which you would get if you did the multiplication first, but 110. This behavior is surprising at first, but it derives from the three simple rules for executing messages. This counter intuitive order of mathematical operations is the price that we have to pay for the simplicity of the Smalltalk model, which has only methods. If you want your expression to obey the normal priority of mathematical operations, then you should use parentheses, since when message sends are enclosed in parentheses, they are executed first. Hence the expression `20 + (2 * 5)` returns the result 30, as shown in Step-by-step 11.9 and Figure 11.9.

Script 11.9: *Step-by-step decomposition of the expression* `20 + (2 * 5)`

20 + (2 * 5)

```
(1)  2 * 5
  –returns> 10
(2)  20 + 5
  –returns> 30
```

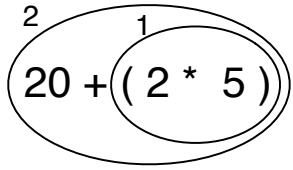


Figure 11.9: Decomposition of the expression 20 + (2 * 5).

Important! Message sends surrounded by parentheses are executed first. Therefore,in the expression 20 + (2 * 5), the message with * is executed before the one with +, which is the usual order of operations in mathematics.

Important! Note In Smalltalk,the mathematical message selectors such as +and * all have the same priority. The symbols + and * are simply message selectors for binary messages. Therefore, * does not have priority over +. If you want to force one operation to take precedence over another, then you should use parentheses. The fact that Smalltalk does not follow mathematical precedence can be confusing at the beginning. Therefore, when you have multiple binary messages representing a mathematical expression, give yourself and anyone else reading your program a break and insert parentheses to express how the computation should be performed. When you have become more accustomed to the way that messages are executed, you will probably become more laissez-faire about parentheses.

A consequence of rule 1 —which provides for the order of execution of different types of messages, with unary messages being executed before binary messages, and binary messages before keyword-based messages— is that you very often do not have to use parentheses. That is, most of the time, you do not have to worry about order of execution. Table 11.4 shows expressions written to be executed according to Smalltalk's rules and equivalent expressions using parentheses, which would be necessary if the order of precedence did not exist.

11.8 Summary

- A message is always sent to an object, called the message receiver, which may be the result of other messages.

Table 11.4: Some Expressions and Their Fully Parenthesized Equivalents.

Without Parentheses	**Equivalent Expression with Parentheses**
pica color: Color yellow	pica color: (Color yellow)
pica go: 100 + 20	pica go: (100 + 20)
pica penSize: pica penSize + 2	pica penSize: ((pica penSize) + 2)
2 factorial + 4	(2 factorial) + 4

- Unary messages are messages that take no argument. A unary message send is of the form of receiver messageName.
- Binary messages are messages that involve two objects: the receiver of the message and a single argument. The message selector of a binary message consists of one or two characters from the following list: $+$, $*$, $/$, $|$, &, $=$, $>$, $<$, $\sim$, @. Binary message sends are of the form receiver messageName argument.
- Keyword-based messages are messages that take one or more arguments and use a keyword with at least one colon character :. A keyword-based message send taking two arguments is of the form receiver messageNameWordOne: argumentOne messageNameWordTwo: argumentTwo.
- Rule 1. Unary messages are executed first, then binary messages, and finally keyword- based messages.
- Rule 2. As in mathematics, expressions in parentheses are executed before any others.
- Rule 3. When messages are of the same type, the order of execution is from left to right.
- In Smalltalk, mathematical message selectors such as + and * have the same priority, and therefore * does not have priority over +. You should use parentheses to ensure that your mathematical expressions are executed in the proper order.

Part III

Bringing Abstraction into Play

Chapter 12

Methods: Named Message Sequences

Up to now, you have been using scripts to create robots and send them sequences of messages. Using scripts has the advantage of being a straightforward approach, but it has some severe limitations. One of the major limitations is that a script cannot be called by another script. This is a serious problem, because a script cannot be reused by other scripts. You have to rewrite the same sequence of messages again and again.

Wouldn't it be nice if one could define a kind of script whose sequence of messages could be sent to any robot? In fact, this is possible, and such a sequence of messages is called a *method*. (In the context of this book we will not go into the full power of methods, since that would get us into the somewhat tricky subject of object-oriented programming.) A *method* is a named script. The name of a method can be used in a script or even in another method to invoke the method. Actually, there is nothing much new here: all the robot messages that you have used so far represent methods that you could use with any robot!

In this chapter, you will learn how to define methods. You already know most of what you need to write the code of a method. However, a method must be defined using a special editor called a *code browser*. We will start by comparing a script and a method. Then we will define a method, and finally, we will look in detail at what we have accomplished.

12.1 Scripts versus Methods

Let's look at one of the scripts that you have already written, for example, Script 12.1, which creates a robot and tells it to draw a square with side length 100 pixels.

Script 12.1: *Pica draws a simple square.*

```
| pica |
pica := Bot new.
4 timesRepeat:
   [ pica turnLeft: 90.
   pica go: 100 ]
```

The problem with this script is that each time you need to draw a square of side length 100 you need to *copy* the three last lines of Script 12.1. Furthermore, if you want another robot (for example, *daly*) to draw the square, you must change the name *pica* to *daly* everywhere. This is illustrated by Script 12.2.

Script 12.2: *Pica and daly each draw a simple square.*

```
| pica daly |
pica := Bot new.
daly := Bot new.
daly jump: 200.
daly color: Color red.
4 timesRepeat:
   [ pica turnLeft: 90.
   pica go: 100 ].
4 timesRepeat:
   [ daly turnLeft: 90.
   daly go: 100 ].
```

For all these reasons, working with scripts is not easy. In fact, I suspect that the following three statements reflect your personal experience with scripts:

- Writing long scripts is a painful task.
- Repeating long scripts is boring and error-prone.
- When one is copying complex scripts, the likelihood of making a programming error, such as omitting a line, is high. (A programming error is an error in the logic of a program. In contrast to syntax errors, which are caught quickly by the computer because they are errors in program structure, programming errors can be quite difficult to catch.)

To overcome these difficulties, we would like to *define* a sequence of messages once and for all, give the sequence a *name*, and then be able to send the *named sequence* as a single message to any robot, just as we have been able to send predefined robot messages such as go:, north, and jump:.

With this approach we could define a new *method* called square, and then write Script 12.3. But don't execute the script yet, because the method squarehas not yet been defined. Once you have the method square, you will no longer have to copy and adapt the sequence of messages defining a square. You can simply use it twice. The message send pica square will tell pica to carry out the instructions encoded in the method square.

I hope that I have convinced you that defining methods will be worth the effort.

Script 12.3: *Pica and daly draw squares using the method square.*

```
| pica daly |
pica := Bot new.
daly := Bot new.
daly go: 200.
daly color: Color red.
pica square.
daly square
```

12.2 How Do We Define a Method?

In this section I will give you a cookbook recipe for creating a method. In Squeak you can define methods on any object, but in this book you will define methods only for robots. To help you out with this, I developed a specialized code browser named Class Bot Browser just for defining methods for your robots. There is a Class Bot Browser in the working flap, or you can always create one by dragging its thumbnail from the dark blue flap or via the menu open... .

Using a Class Bot Browser to define a method requires you to (1) choose or create a method category, which is a kind of method folder; (2) type the method; and then (3) compile it. These steps will be described in upcoming sections. But first let us have a detailed look at the different parts of a Class Bot Browser.

A Class Bot Browser

Defining methods requires a new tool: the editor shown in Figure 12.1. This browser is actually a simplified version of the browser used by Smalltalk

programmers. The browser consists of three parts, or *panes*:

Categories. The upper left pane contains the *category list*. It shows the different method categories. Method categories are just names that group methods together so that you can find information faster. In Figure 12.1, the category turning is selected; it groups all the operations having to do with robots' directional changes. Other categories that group other robot methods are also listed.

Methods. The upper right pane contains the *method list*. This list shows the method names of the methods in the selected category. In Figure 12.1, five methods are listed: pointAt:, turn:, turnLeft:, turnRight:, and turnTo:. The method named turn: is currently selected.

Method Definition. The bottom pane contains the *code editor*. It shows the definition of the method whose name is selected together with optional comment text. This pane is also the place where you can type the code of a new method.

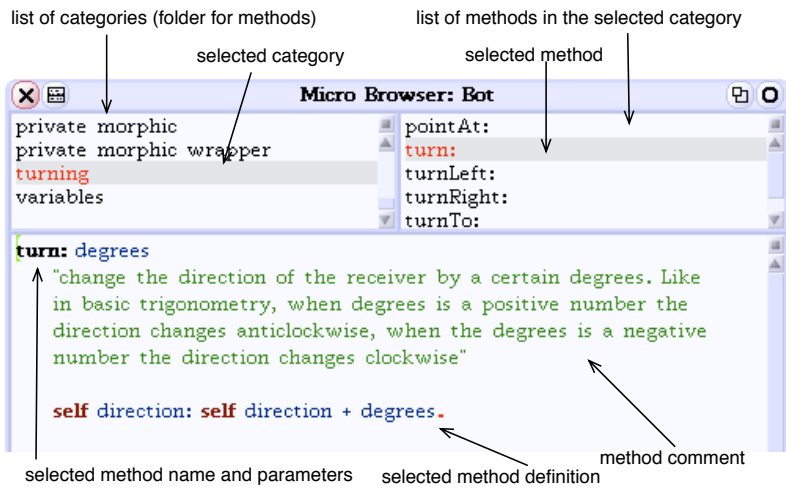


Figure 12.1: A Class Bot Browser showing the definition (bottom pane) of the method turn: (selected in the upper right pane) belonging to the category turning (selected in the upper left pane).

Creating a New Method Category

Methods are grouped by categories. A category is defined by giving it a name. To define a method, you either define a new category for it or select an existing category. Let's create a new category named regular polygons. Here is how it is done:

1. Click with the right mouse button (Alt-click or Option-click) on the category list. A menu like the one in Figure 12.2 will pop up.

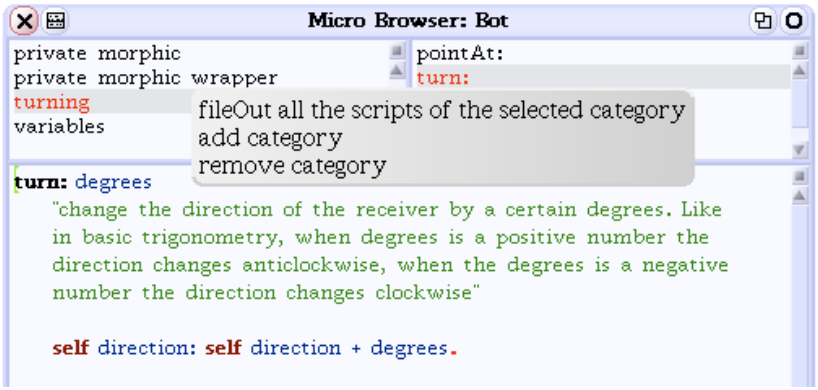


Figure 12.2: To create a new method category,open the category menu and select add category.

2. Select the option add category of that menu.

3. Type the name of the category in the dialog box that appears, as shown in Figure 12.3. You may choose any name for the category. Of course, meaningful names are better then meaningless ones when you want to share your work with other people or find your method again at a later date.

4. Click the Accept button to validate your choice.

As shown in Figure 12.4, the name of the new category appears in the category pane and is automatically selected. The editor is ready to accept a new method definition. It shows you a reminder of how to define a method, which you can remove when you start typing your method. You are now ready to define your first method.

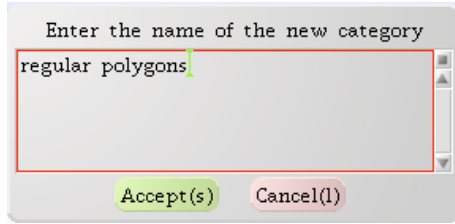


Figure 12.3: Enter a new category name in the dialog box and click the Accept button.

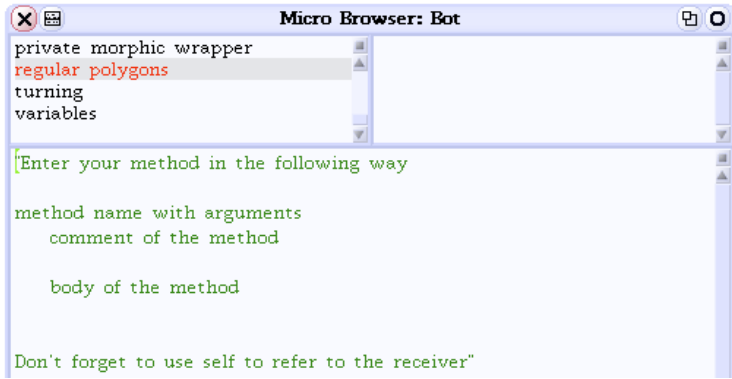


Figure 12.4: The new category is ready.

Defining Your First Method

If the category to which you want to add your method is not selected, select it. Then type the contents of Method 12.4 (following this paragraph) into the code editor pane. To do this, select all the text in the code editor and start typing your method.

Méthode 12.4: *A new method for drawing a square of side length 100*

```
square
  "Draw a square of side length 100 pixels"
  4 timesRepeat:
    [ self go: 100.
    self turnLeft: 90 ]
```

Defining a method is a three-step process:

1. **Typing the method.** Typing code into the code editor pane works exactly as with the script editor. First delete the reminder text that is

in the code editor pane. The easiest way to do this is to point your mouse at the beginning of the editor pane before the first character and click. This will select all of the code editor text. Once you finish typing the new method, your code display pane should look like Figure 12.5.

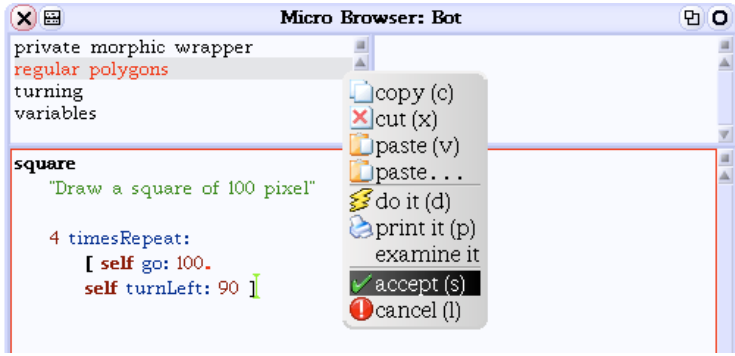


Figure 12.5: After typing in the method square, you compile it using the code editor menu.

2. **Compiling the method.** Click to bring up the menu for the code editor, as shown in the figure, and select the option accept. Doing so causes the method definition to be *compiled*, that is, transformed into a representation that the computer can understand and execute. A new method named square now appears in the method list. If you made a mistake while typing the method, Squeak will report the error as it would for a script.

   If you defined the method correctly, you should be able to compile it without Squeak reporting any errors. The browser will then reflect the fact that the compilation is complete and that robots can now understand messages containing the new method by showing the new method's name in the method list (see Figure 12.6).

3. **Testing the method.** As the saying goes, the test of whether a pudding has been properly made is in tasting it. Likewise, you have not finished creating your new method until you have tested it, because the method that you defined might not do what you had in mind for it. Now you may execute Script 12.3. You should get one black square and one red square.

Observe that a method can be used and reused, as demonstrated by Script 12.3. This is old news. Indeed, you have used this fact since the

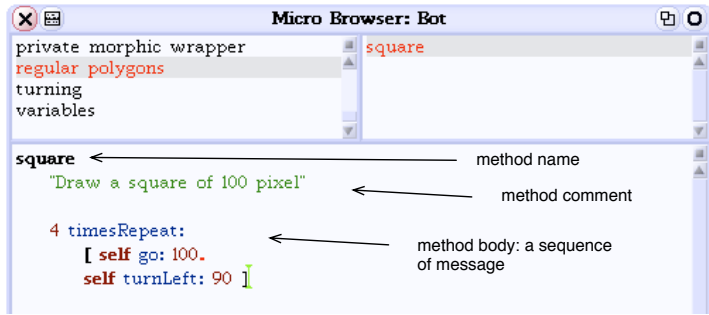


Figure 12.6: A method is composed of a name, an optional method comment, and a method body.

beginning of this book: message selectors such as go: and turnLeft: are the names of methods defined in the same way as the method square.

12.3 What's in a Method?

I asked you to type a method without much explanation. Now it is time to analyze the structure of the method.

A method is composed of a *name*, an optional *method comment*, and a *method body* (a sequence of expressions), as shown in Figure 12.6. The method name can also contain parameters (see Chapter **??**), and the method body can also define local variables using vertical bars | |.

Method name. A method name should always represent what the method does, not how it does it. When you want somebody to open a door, you don't explain all the physics and mathematics involved. It is the same for methods.

Important! A method name should always represent what the method does,not how it does it.

Method names without parameters, such as square, follow the same syntax as variable names. They are composed of alphanumeric characters (letters and digits) and start with a lowercase character. In our case, the method name is square.

Method comment. A comment consists of text enclosed between double quotes (*"This is a comment"*). The text itself cannot contain any double quotes. However, a comment can be as long as you like, and can continue over several lines.

In general, a comment explains the purpose and the effect of the method. It explains how the method can be used, but not how the method does its job. Anyone who wants to know how the method works can read the method's body.

If the method name is clear enough, the comment may be omitted. In our case the method comment is, *"Draw a square of side length 100 pixels"*.

Method body. After the comment comes the method definition itself, which is the sequence of messages that are executed in response to a message. In our case, the method body is as follows:

Méthode 12.5:

```
4 timesRepeat:
   [ self go: 100.
   self turnLeft: 90 ]
```

Important! A method is a named sequence of expressions. It is composed of a name, an optional comment, and a sequence of expressions.Once a method for robots has been defined, any robot can execute it in response to a message with the same name.

Scripts versus Methods: An Analysis

Let's compare Method 12.7 with Script 12.6. You can see three significant differences: (1) The line in the script declaring the variable picais not in the method; (2) the line creating the robot is also not in the method; (3) in the remainder of the method, the variable pica is replaced by self.

Script 12.6: *Pica draws a simple square.*

```
| pica |
pica := Bot new.
4 timesRepeat:
   [ pica turnLeft: 90.
   pica go: 100 ]
```

Méthode 12.7: *Instructions to any robot for drawing a simple square.*

```
square
"Draw a square of side length 100 pixels"
4 timesRepeat:
   [ self go: 100.
   self turnLeft: 90 ]
```

Remember that a robot method represents a sequence of expressions that can be sent to *any* robot: The robot in the script referred to by the variable pica will not necessarily be the receiver of the message square. The robot daly, or any other robot, could also be the receiver of the message square, as we saw in Script 12.6.

Therefore, it is important in defining the method squarenot to refer to any *particular* robot, since the message square will be sent to *different* robots at different times. Thus we need a name that will stand for whatever robot happens to be the message receiver of the message square. That is the purpose of self. Inside a method, self represents the object receiving the message, because that object *itself* will be executing messages such as go: and turnLeft:.

The Variable "self"

In Chapter 8 I explained that a variable is just a named placeholder for an object. In particular, I emphasized that the same variable could be used to point to different objects at different times.

In the case of a method, the variable self points to whatever object has received the message: when the expression pica square is executed, the variable self in the method square refers to the robot named pica, and when the expression daly square is executed, self refers to the robot named daly.

Important! Inside a method,the variable self represents the object that has received the message that led to the execution of that method.For example,when the expression pica square is executed, pica receives the message square and executes the robot method of the same name.The word self in the method now refers to the robot named pica, since pica is executing the method; when the expression daly square is executed, self refers to the robot named daly.

The word self in a method is a special sort of variable, because you cannot change its value. Only Squeak can assign the value of self. That is why self is not declared between vertical bars | |. Moreover, self can be

used only inside a method definition.

Important! When the code of a method needs to send a message to the receiver, the message is sent to self. For example, in the method square, the robot executing the method needs to turn *itself*, so the message turn: 90 is sent to self.

Method or Not: That Is the Question

At this stage, you may be tempted to go back and convert all the scripts you have written into methods. This is not advisable, because not every script is worth turning into a method. In general, you should define a method when you have a sequence of messages that is general enough to be used several times.

12.4 Returning a Value

A method can also return a value by using the character ^, called a *caret*. When you type a caret, Squeak prints an upward-pointing arrow ↑) in the environment. Imagine that you want to have a method that returns the greatest distance that a robot should be allowed to move at one time. You could define the method maxDistance, shown in Method 12.8. In this example, the method simply returns a number, but instead, you could have the method return the result of a complex expression, perhaps involving where the robot is positioned on the screen.

Méthode 12.8: *This method returns a value.*

```
maxDistance
    "returns the maximum distance a robot should be able to move"
    ↑ 100
```

If a method does not explicitly return a value, then it returns the message receiver by default. Method 12.9 is equivalent to the method squaredefined previously. In fact, at the end of every method there is an implicit expression ^self if there is no explicit return expression. However, in this book you do not have to worry about that.

Méthode 12.9: *This equivalent version of the squaremethod explicitly returns the message receiver.*

```
squareEquivalent
```

```
"Draw a square of side length 100 pixels"
4 timesRepeat:
   [ self go: 100.
   self turnLeft: 90].
↑ self
```

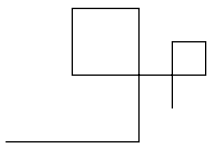
In this book, you will not use this feature much, but it is important to know that a method always returns a value.

12.5 Drawing Patterns

Now it is time to practice. As you have seen, it is quite easy to transform a script into a method. Many seasoned programmers use scripts to test ideas. When they have proven the feasibility of an idea in the form of a script, they move the code of the script into a method for later reuse. The next exercise trains you to do exactly this. Let's consider Script 12-5, which draws an abstract "art nouveau" design.

Script 12.1 (*Pica draws a simple abstract pattern.)*

```
| pica |
pica := Bot new.
pica go: 100 ;
turnLeft: 90 ;
go: 100 ;
turnLeft: 90 ;
go: 50 ;
turnLeft: 90 ;
go: 50 ;
turnLeft: 90 ;
go: 100 ;
turnLeft: 90 ;
go: 25 ;
turnLeft: 90 ;
go: 25 ;
turnLeft: 90 ;
go: 50
```

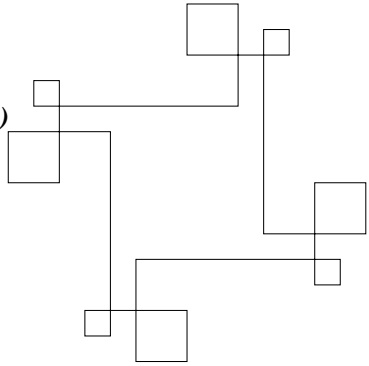


Experiment 12.1 (*A Simple Abstract Design)*
Create a method named pattern that produces the figure drawn by Script 12.1.

You now can use this method in a script to draw a more elaborate design that might be used for an art nouveau picture frame.

Script 12.2 (*An art nouveau picture frame)*

```
| pica |
pica := Bot new.
4 timesRepeat: [ pica pat-
tern ; go: 50 ]
```



At this point, the astute reader might ask, Why don't we create a method, named frame50, for example, corresponding to that of Script 12.2? This is indeed possible, since *any method created for a robot can be reused by another robot method*. Creating such methods is the topic of the next chapter.

Experiment 12.2 (*A Method for the Art Nouveau Picture Frame)*
Create a method named frame50 that produces the design produced by Script 12.2.

12.6 Summary

- A *method* is a named sequence of expressions. It is composed of a name, a comment, and a sequence of expressions. Once a method for robots has been defined, any robot can execute it in response to a message with the same name.
- A method name should always represent what the method does, not how it does it.
- A new method for a robot is created using a Class Bot Browser, which is a special editor for defining methods.
- Inside a method, the variable self represents the object that receives the message. When the method's code needs to send a message to the receiver, the message should be sent to self.

12.7 Glossary

Method categories. A method category is a folder in which methods are sorted. Categories help you to find methods more quickly.

Method. A method represents a sequence of expressions that an object can execute. A method has a name. It is executed when an object receives a message having the same name.

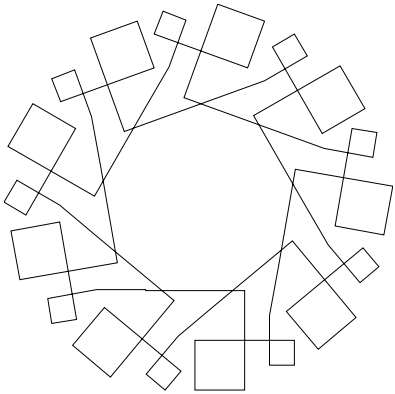
Class Bot Browser. A Class Bot Browser is a special tool for viewing and editing methods.

Comment. A comment is a piece of text surrounded by quotation marks that explains the purpose of a method.

self. The variable self is predefined by Smalltalk. It always represents the receiver of the message in a method definition.

Chapter 13

Combining Methods



In Chapter 12, you learned how to define methods. I showed that defining methods is interesting and useful because (1) methods save you from having to rewrite scripts, which is time-consuming and subject to error, and (2) methods can be used and reused by different robots. The other main advantage of using methods is the possibility of using methods in other methods, that is, calling one or more existing methods as part of the definition of a new method. The reuse of methods is what we will explore in this chapter.

Being able to reuse methods is extremely important, because we can define a method in terms of another one without having to know all the details of how the second method is defined. We just call it and ask it to do what it is designed to do.

13.1 Nothing Really New: The Square Method Revisited

Having methods call other methods (which we call composing methods) is quite natural and is not really new. In fact, it is what you did in Chapter 12 when you defined a method! The method square includes in its definition calls to the methods turnLeft:, go:, and timesRepeat: (as shown in Method 13.1). Thus even the simple method squareis defined in terms of other methods, and we did not have to know how turnLeft:, go:, and timesRepeat: are defined. We needed to know only what they do. So we are essentially done with this chapter, with nothing left to do but have some fun.

Méthode 13.1:

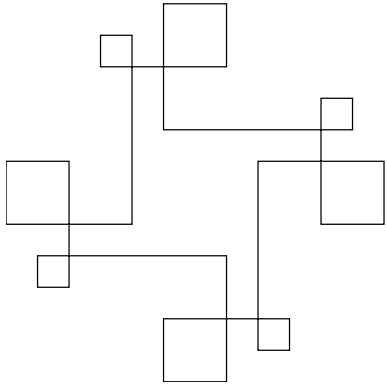
```
square
"Draw a square of 100 pixels wide "
4 timesRepeat:
[ self go: 100;
turnLeft: 90 ]
```

13.2 Other Graphical Patterns

In Chapter 12, I asked you to define the method pattern, which draws a simple abstract pattern (See script 12.1). Now I will ask you to perform some further experiments that will produce more drawings by defining more methods.

Experiment 13.1
Define a method pattern4 that calls pattern four times to produce the figure below. You will use this method later, in another script. After you have created the method pattern4, use the following three-line script to make pica draw the figure.

```
| pica |
pica := Bot new.
pica pattern4
```

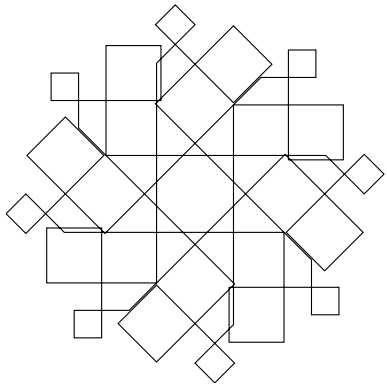


Experiment 13.2 (*A Ferris Wheel)*
Define a method called tiltedPattern that draws the picture at the beginning of this chapter, which looks somewhat like a Ferris wheel. Hint: you will have to call pattern nine times, and the angle through which to turn between calls is 10 degrees.

Experiment 13.3
Doubling the Frame Define the method doubleFrame, presented below,that draws the picture shown after the method definition.

doubleFrame

```
8 timesRepeat:
[ self pattern.
self turnLeft: 45.
self go: 100 ]
```



13.3 What Do These Experiments Tell You?

Now let's see what you can learn from the experiments you did. As you can see from the methods pattern4, tiltedPattern, and doubleFrame, the method pattern was defined only once, and then *reused several times* in different methods. Defining patternas a method allows you to (1) define it only once, (2) reuse it in various contexts, and (3) not introduce errors by copying this method over and over.

If you look at the definition of the method doubleFrame, you see that it is defined in terms of the pattern method, which is itself defined in terms of other methods, such as go: and turnLeft:. In fact, a complex method is often defined in terms of simpler methods, which themselves are defined in terms of even simpler methods, which themselves are defined in terms of even simpler methods, which themselves.... The advantage of this is that it is easier to understand and to define simple methods than complex methods, and the technique of defining methods in terms of simpler methods limits the degree of complexity in any one method. In Chapter **??**, I will show you that to solve a problem, it is advantageous to decompose it into smaller subproblems, solve these subproblems, and then use the solutions to the

smaller subproblems to solve the main problem.

It is essential to understand that in defining the method doubleFrame, you do not have to know how pattern is defined. You just need to know what it does and how to use it! When we define a method, we are giving a single name to a sequence of messages, which reduces the number of details that we have to keep track of. We just have to remember what the method does and its name, not how it does it. We say that we are building an *abstraction* over the definition details.

To make this point clear, I rewrote the method doubleFrame without calling the method patternby directly copying the definition of pattern(shown in italics). Compare doubleFrame WithoutCallingPattern (Method 13.2) with the method doubleFrame. The new version without pattern is not only longer, but for most people it is also more confusing and harder to understand.

Now imagine what would happen if I did the same with the code of turnRight:, turnLeft:, and go: — because these are methods too. It would be a nightmare! There would be so many details that we would be lost all the time.

Méthode 13.2: *Creating the double frame without the abstraction of the pattern method*

doubleFrameWithoutCallingPattern

```
8 timesRepeat:
[ self go: 100.
self turnRight: 90.
self go: 100.
self turnRight: 90.
self go: 50.
self turnRight: 90.
self go: 50.
self turnRight: 90.
self go: 100.
self turnRight: 90.
self go: 25.
self turnRight: 90.
self go: 25.
self turnRight: 90.
self go: 50.
self turnLeft: 45.
self go: 100 ]
```

Important! When you write a new method,it can call other methods.You can use a method without knowing how it is written.After you finish

writing a method,you can call it when you write another method.

13.4 Squares Everywhere

Now it is time to practice. Define the following methods using the method square.

Experiment 13.4 (*Some Boxes)*
Define methods boxandseparatedBoxthat produce the pictures shown in Figure 13.1.

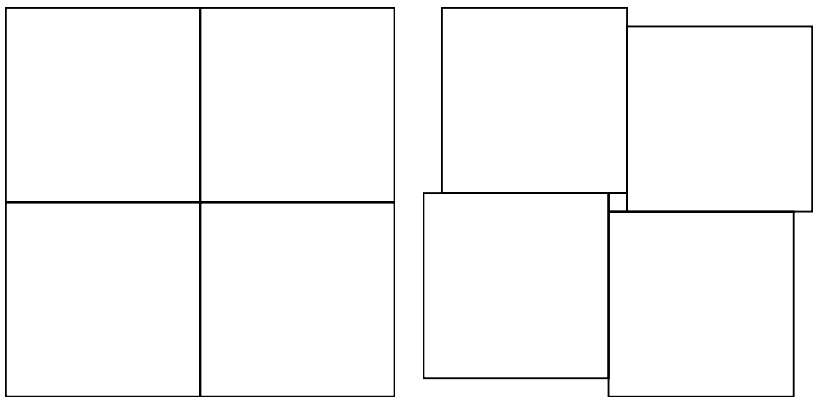


Figure 13.1: Boxes

Experiment 13.5 (*Your Choice)*
Use your previous methods to generate various figures of your choice.Have fun!

Experiment 13.6 (*A Star)*
Using the method box, experiment and define a method star that produces the right-hand picture in Figure 13.2.

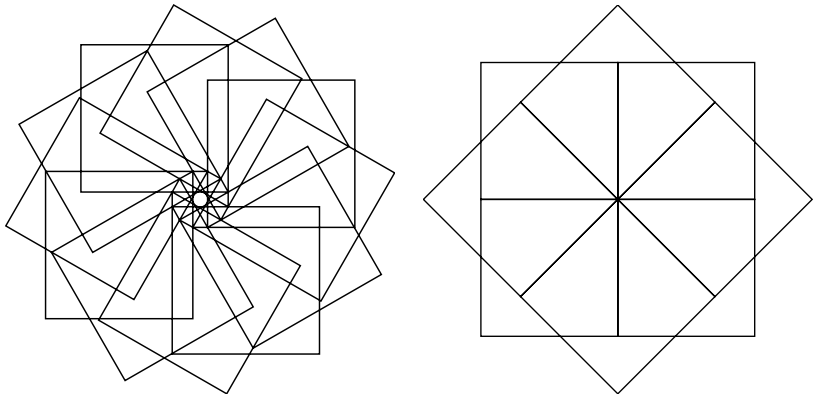


Figure 13.2: Stars

13.5 Summary

- When you write a new method, it can call other methods.
- You can use a method without knowing how it is written; you need to know only what does.
- After you finish writing a method, you can call it when you write other methods.
- Hiding the details of a method by giving it a name is called *abstraction*.