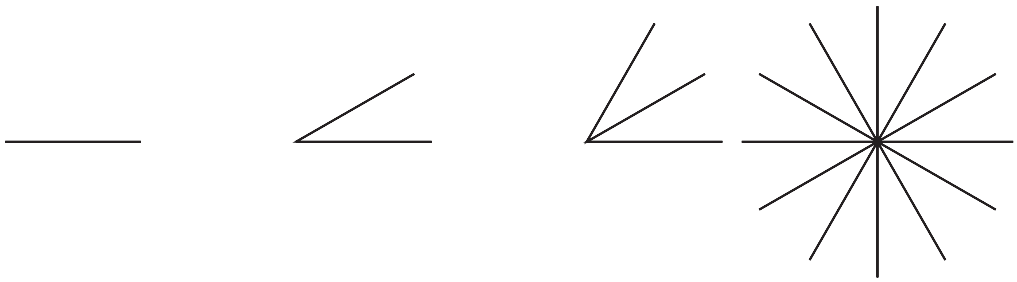
PART 2

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 *loops* to 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.

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.

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. *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.

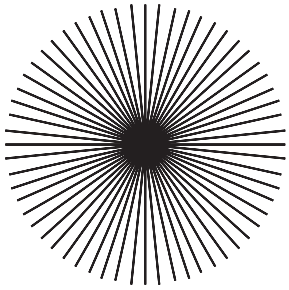
Loops at Work

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.



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.

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

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

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.

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*

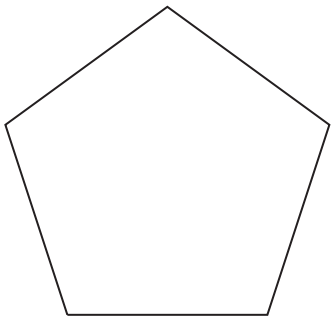
```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90
```

Experiment 7-2 (A Square Using a Loop)

Transform Script 7-4 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.

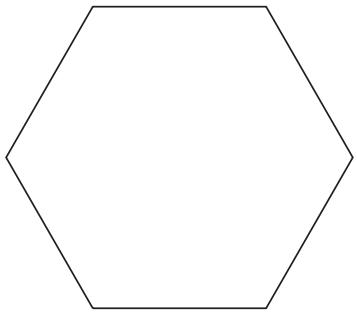
Experiment 7-3 (A Regular Pentagon)

Draw a regular pentagon using the method `timesRepeat:`.



Experiment 7-4 (A Regular Hexagon)

Draw a regular hexagon using the command `timesRepeat:`.

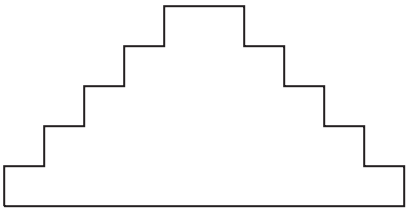


Once you have gotten the hang of it, try drawing a regular polygon with a very large number of sides. You may have to reduce the side length to make the figure fit on the screen. When the number of sides is large and the side length is small, the polygon will look like a circle.

Rediscovering the Pyramids

Recall how you coded the outline of the pyramid of Saqqara in Experiment 3-5. You can simplify your code by using a loop, as shown in Script 7-5.

Script 7-5. *A looping pyramid script*

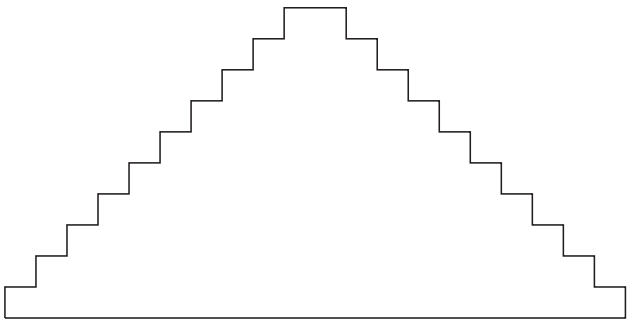


```
| pica |
pica := Bot new.
5 timesRepeat:
   [pica north.
   pica go: 20.
   pica east.
   pica go: 20].
5 timesRepeat:
   [pica go: 20.
   pica south.
   pica go: 20.
   pica east].
pica west.
pica go: 200.
```

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 7-5.



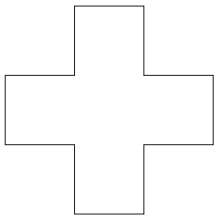
You may now want to generate pyramids with even larger numbers of terraces. The size of the terraces will have to be adjusted if you want them to fit on the screen.

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.

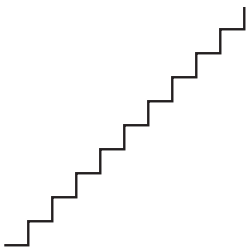
Experiment 7-6 (A Swiss Cross)

Draw the outline of the Swiss cross shown on the right using `turnLeft:` or `turnRight:` and `timesRepeat:`.



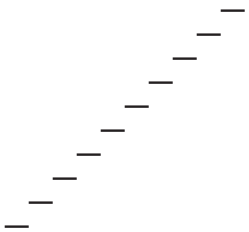
Experiment 7-7 (A Staircase)

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.

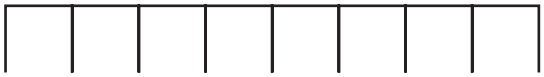


Experiment 7-9 (A Staple)

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 illustrated at the start of Chapter 4.

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 ]`