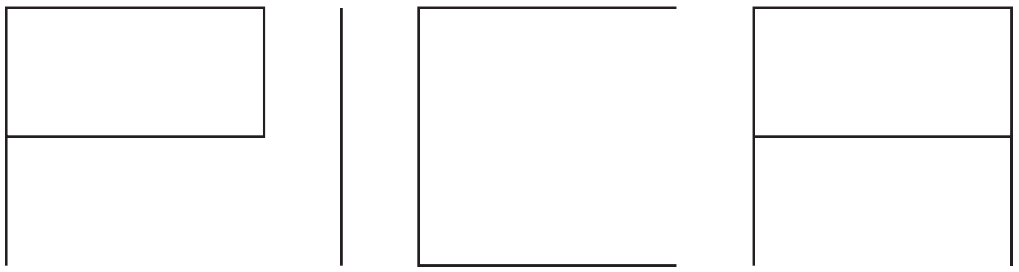
CHAPTER 3

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.



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 supplies 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 manufactures 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.)

■**Note** 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.

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.*

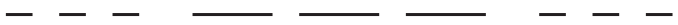
```
| pica |
pica := Bot new.
pica go: 30.
pica jump: 30.
pica go: 30.
```

Experiment 3-1 (Creating and Moving a Robot)

Experiment by changing the values in the previous script.

Experiment 3-2 (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.



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`.

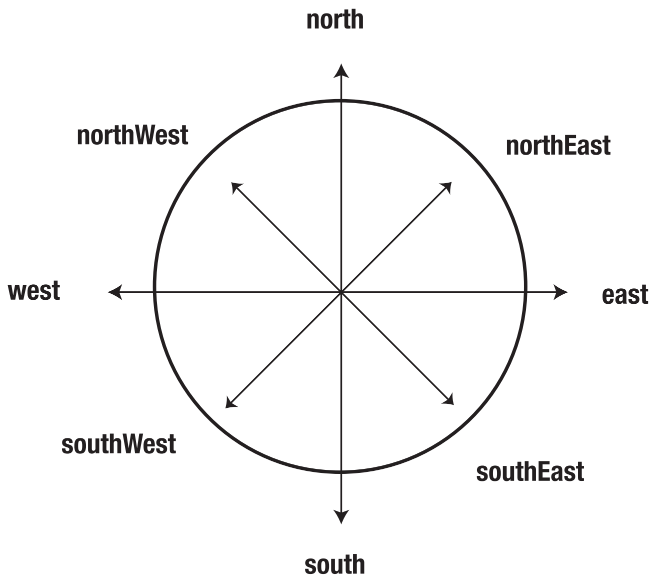


Figure 3-1. *The default absolute directions of the compass in which a robot can point.*

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 3-3 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 3-3. *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.
```

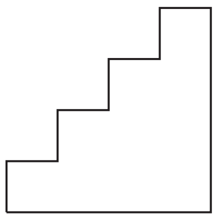
You can use these orientation methods to make more complex drawings.

Experiment 3-3 (A Square)

As a first exercise, draw a square with sides of length 50 pixels. Then draw another square of side length 250 pixels.

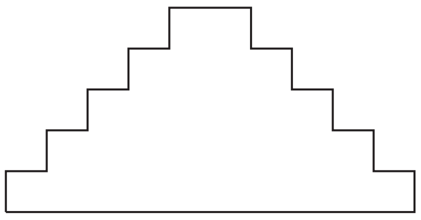
Experiment 3-4 (A Staircase)

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 reproduces this drawing.



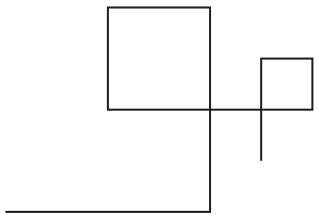
Experiment 3-5 (The Step Pyramid of Saqqara)

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 3-6 (Abstract Art)

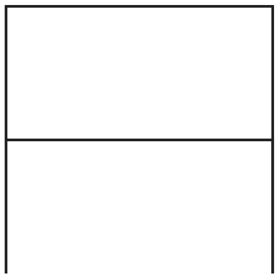
Write a script to draw the picture shown in the figure below.



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 3-4 draws a rather primitive letter "A."

Script 3-4. *The letter A is drawn.*



```
| 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 3-7 (PICA)

Draw the name "PICA" as shown at the start of the chapter. To separate the individual letters, you should use the command `jump:`.

Remark 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.

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.

Summary

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

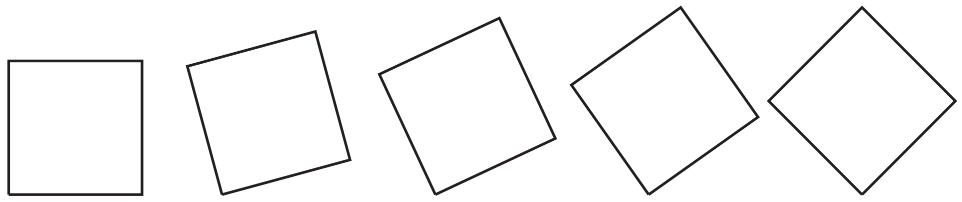
Expressions / Messages	Description	Example
`Bot new`	Create a robot	`pica := Bot new`
`| x y |`	Declare variables to be used in the script	`| pica |`
`jump:` *anInteger*	Tell a robot to move forward a given number of pixels without leaving a trace	`pica jump: 10`
`go:` *anInteger*	Tell a robot to move forward a given number of pixels while leaving a trace	`pica go: 10`
`beInvisible`	Tell a robot to be invisible	`pica beInvisible`
`beVisible`	Tell a robot to be 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 in *any* direction, 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.



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.

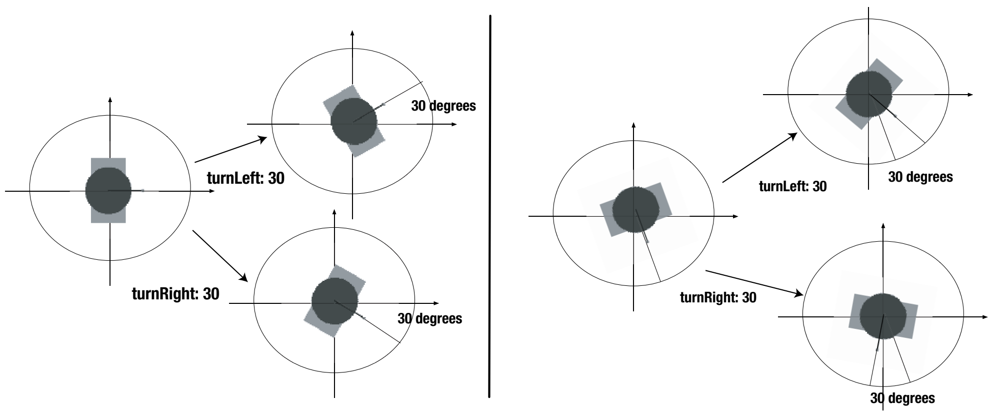


Figure 4-1. *Left: A robot facing east turns left or right through 30 degrees. Right: A robot facing in some other direction turns left or right through 30 degrees.*

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
```

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.

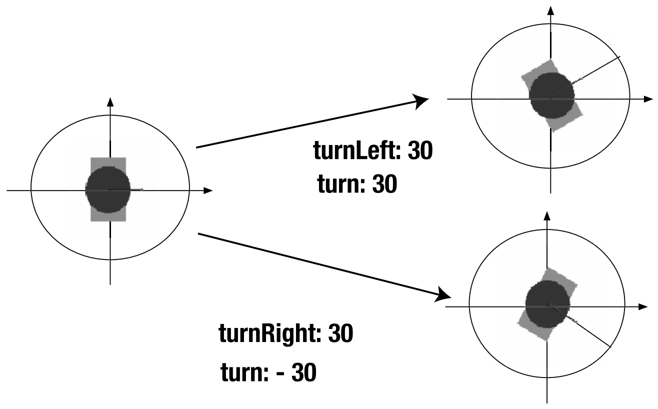


Figure 4-2. *Turning through a 30-degree angle starting from the direction east*

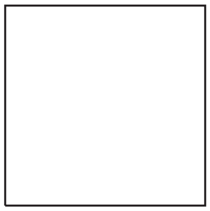
Absolute Versus Relative Orientation

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 *absolutely* using 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:`.



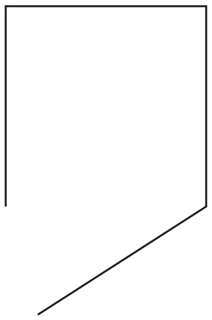
Experiment 4-3 (Tilting the Square)

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 4-3, which attempts to draw a tilted square using the methods `north`, `south`, `east`, and `west` that we presented in the previous chapter.

Script 4-3. *A broken square*



```
| pica |
pica := Bot new.
pica turnLeft: 33.
pica go: 100.
pica north.
pica go: 100.
pica west.
pica go: 100.
pica south.
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 4-3 and Script 4-3 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 pointing 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.

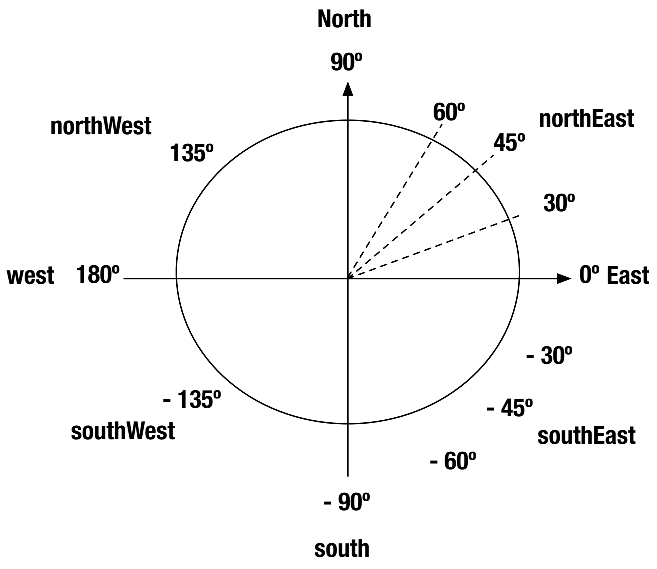
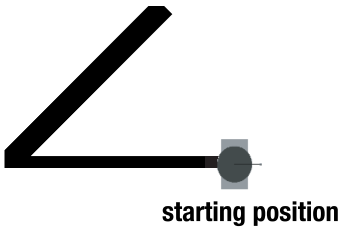


Figure 4-3. *Comparing absolute and relative orientation starting from the easterly direction*

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 the starting position.

Script 4-4. *Moving through angles (1)*



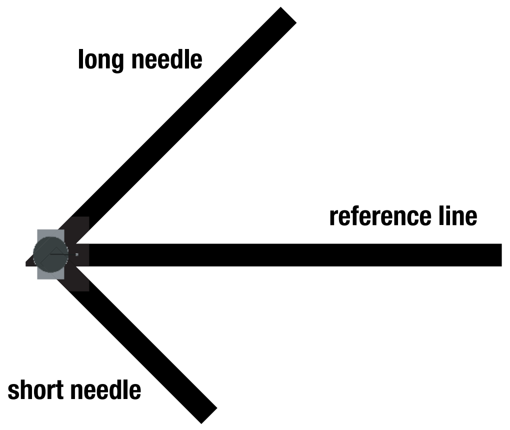
```
| pica |
pica := Bot new.
pica west.
pica go: 100.
pica east.
pica turnLeft: 45.
pica go: 100.
```

The first part of Script 4-4, 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 ($1860 = 60 + 360 \times 5$). 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 4-5. *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.
```

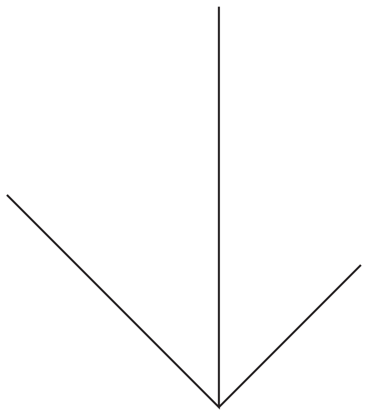
In Script 4-5, the code in italics draws the reference line—that is, the line representing the direction of the robot before a turn method is executed—using the fact that a turn through 180 degrees amounts to turning around to point in the opposite direction. The reference line is also the longest line 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 replacing `daly` by another robot, `berthe`, who honors the French impressionist painter Berthe Morisot.

Script 4-6. *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 4-6 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.

A Robot Clock

I have mentioned that the lines drawn in Script 4-6 are akin to the hands of a clock. The analogy 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 *real* clock that represents a *real* time of day.

Experiment 4-7 (A "Real" Clock)

Modify Script 4-6 as follows:

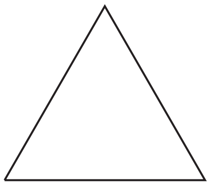
- Keep the direction of reference to the north (this is how Script 4-6 is written). This reference line indicates 12:00 noon or midnight.
- Use the method `turnRight:` for both robots. After all, the hands of a clock move clockwise, which is to the right.
- You can ask `pica` to draw the minute hand 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.

Simple Drawings

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

Script 4-7. *An equilateral triangle*



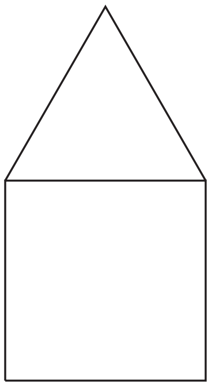
```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 120.
pica go: 100.
pica turnLeft: 120.
pica go: 100.
pica turnLeft: 120.
```

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

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



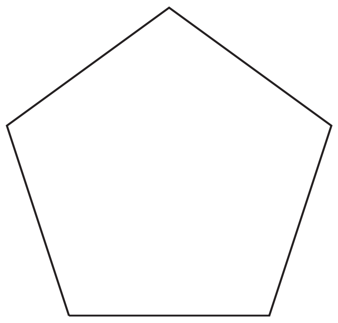
Regular Polygons

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 4-7 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.

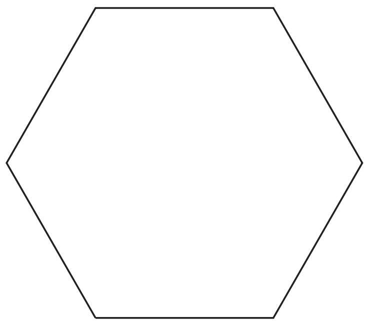
Experiment 4-9

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



Experiment 4-10

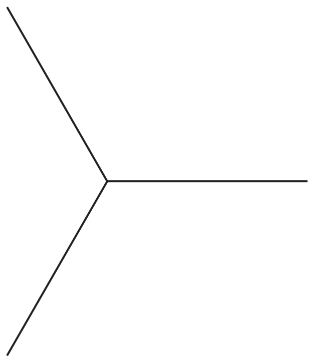
Draw a regular hexagon (a regular polygon with six sides), as shown in the figure, with sides of length 100 pixels.



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

Draw the three-spoked figure shown below.



Summary

- A robot can be oriented *relative* to its *current* direction using the methods `turnLeft:` and `turnRight:`.
- The parameter given to the methods `turnLeft:` and `turnRight:` 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`	Tell 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.

The Main Menu

When you click on the background you get the main menu of the environment, as shown in Figure 5-1.

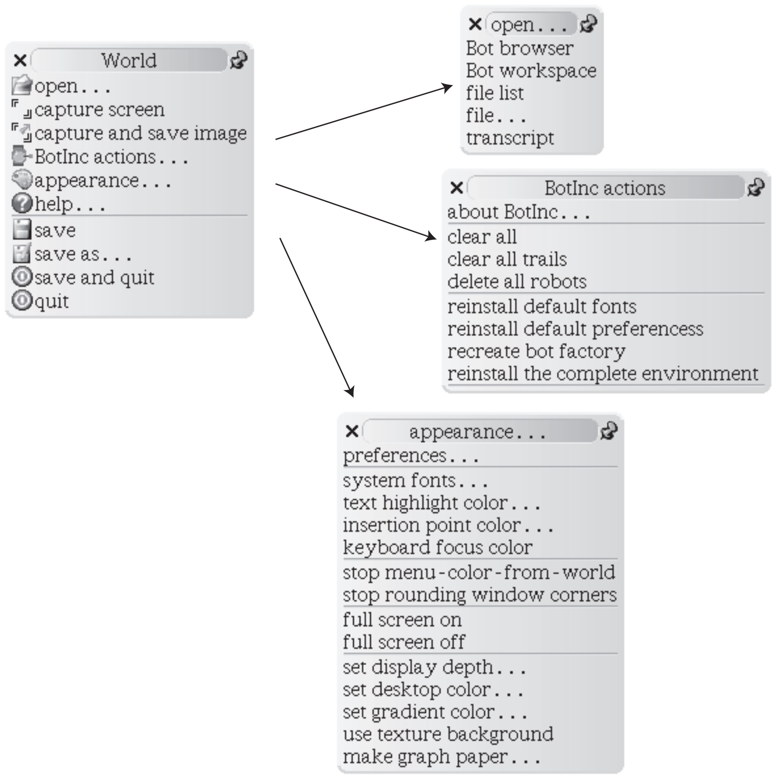


Figure 5-1. *Menu options of the environment*

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

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.

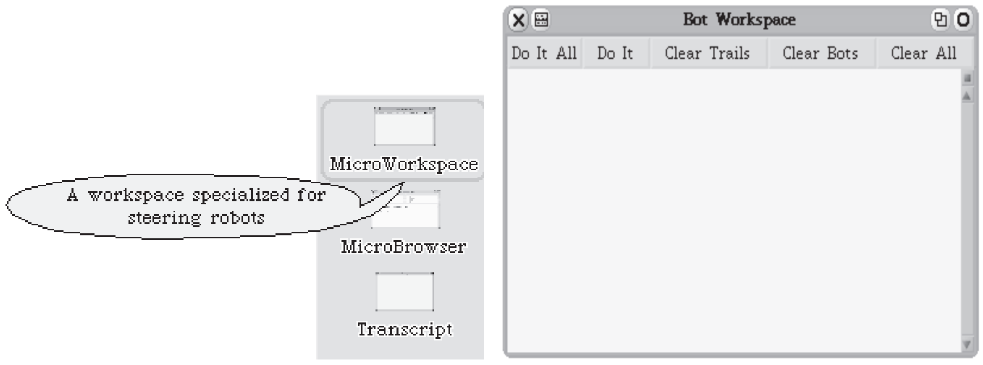
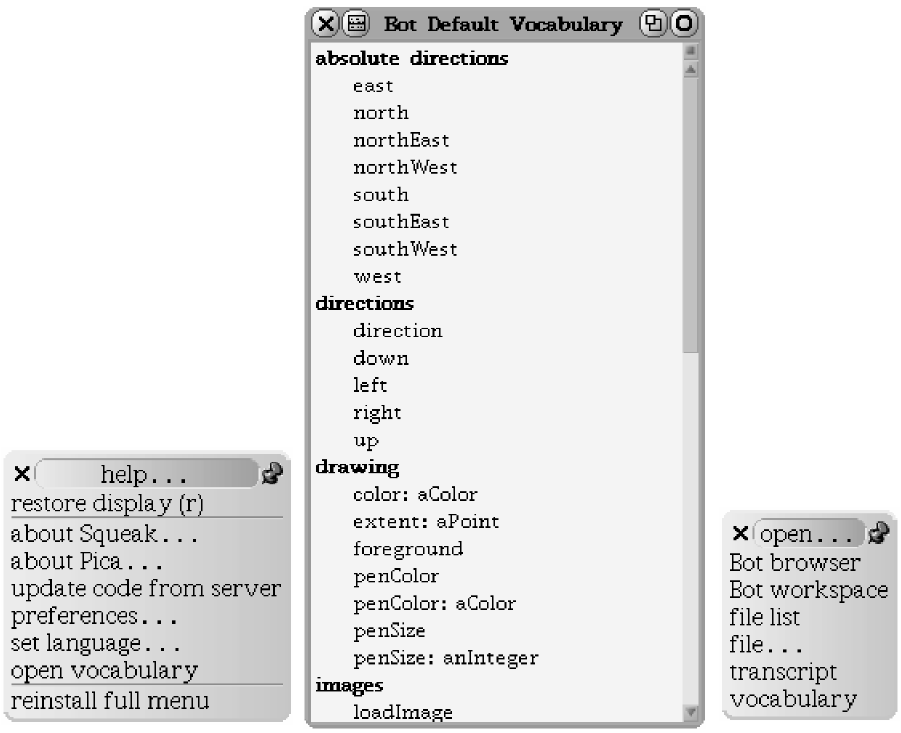


Figure 5-2. *Obtaining a new Bot workspace from the flaps*

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**.



Bot Default Vocabulary

```
absolute directions
    east
    north
    northEast
    northWest
    south
    southEast
    southWest
    west
directions
    direction
    down
    left
    right
    up
drawing
    color: aColor
    extent: aPoint
    foreground
    penColor
    penColor: aColor
    penSize
    penSize: anInteger
images
    loadImage
```

Figure 5-3. *The most important messages for robots*

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 5-1. Each button is associated with a logical set of operations. The left button is 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 5-4). 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.

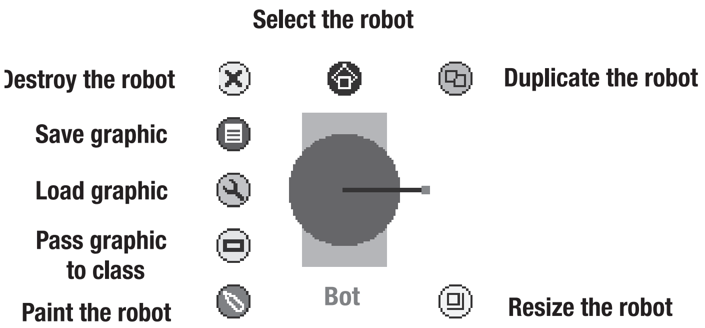


Figure 5-4. *Right clicking on a robot brings up its halo of handles.*

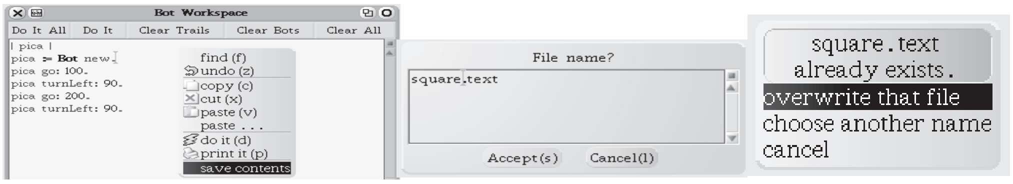
Table 5-1. *Mouse Button and Key Combinations*

	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

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 5-5. 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.



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

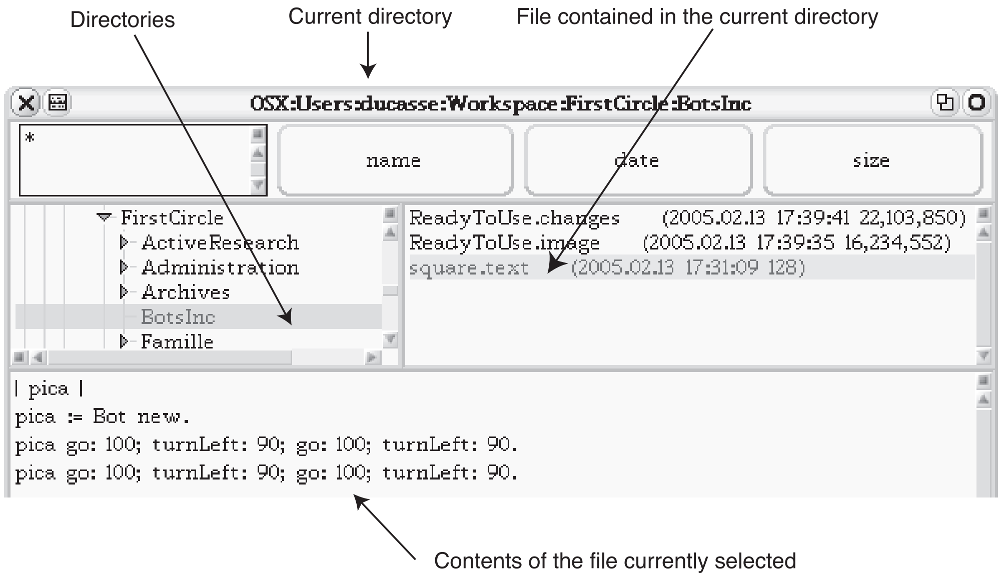
square.text

square.text already exists.

Figure 5-5. *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.*

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 5-6 shows that we are in the folder `Bot testing`, in which the file `square.text` is selected.


Directories
Current directory
File contained in the current directory
OSX:Users:ducasse:Workspace:FirstCircle:BotsInc
*
name
date
size
FirstCircle
ActiveResearch
Administration
Archives
BotsInc
Famille
Contents of the file currently selected

```
ReadyToUse.changes     (2005.02.13 17:39:41 22,103,850)
ReadyToUse.image       (2005.02.13 17:39:35 16,234,552)
square.text            (2005.02.13 17:31:09 128)
```

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

Figure 5-6. *The file list is open to the script* `square.text`

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.

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 5-7.

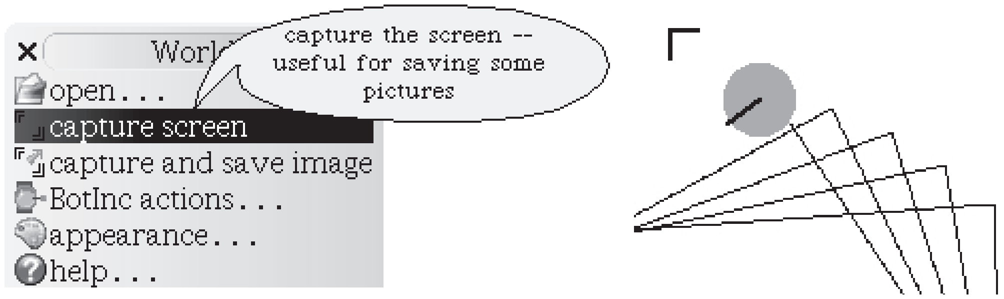


Figure 5-7. *Left: Two possibilities for capturing and saving the capture. Right: The cursor has changed, indicating that Squeak is ready for the capture. Now click to position one corner of the rectangular region you want to capture.*

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 5-7. 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 5-8. 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 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.

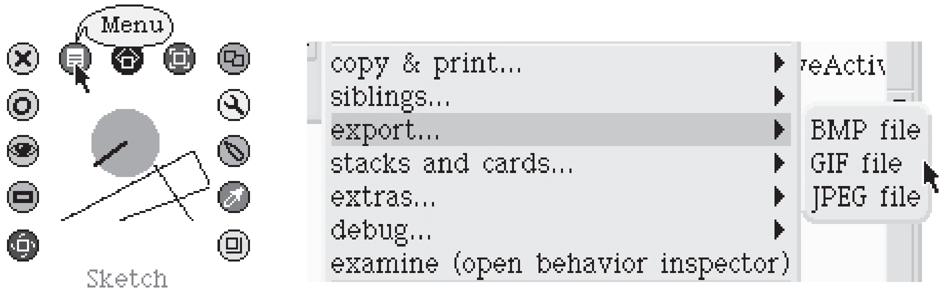


Figure 5-8. *Call up the halo and choose the red handle menu item* ***export…*** *to save the image to disk.*

Message Result

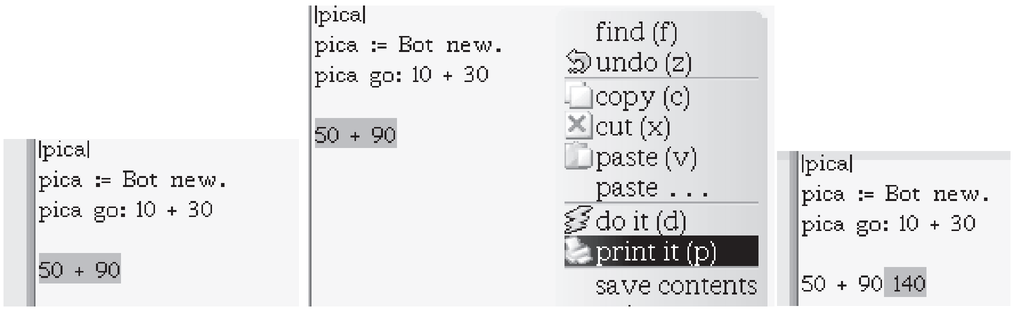
In Smalltalk, objects communicate only by sending and receiving messages to and from other objects. Once an object receives a message, it executes it, and additionally, it returns a result. A result is an object that the receiving object has returned to the sender. Communication between objects by means 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; pica color` returns pica's color, a color object.

In Figure 5-9, the expression `50 + 90` is selected, then using the menu the expression is executed, and the result, `140`, is printed on the screen.


```
|pica|
pica := Bot new.
pica go: 10 + 30

50 + 90
```
```
|pica|
pica := Bot new.
pica go: 10 + 30

50 + 90
```

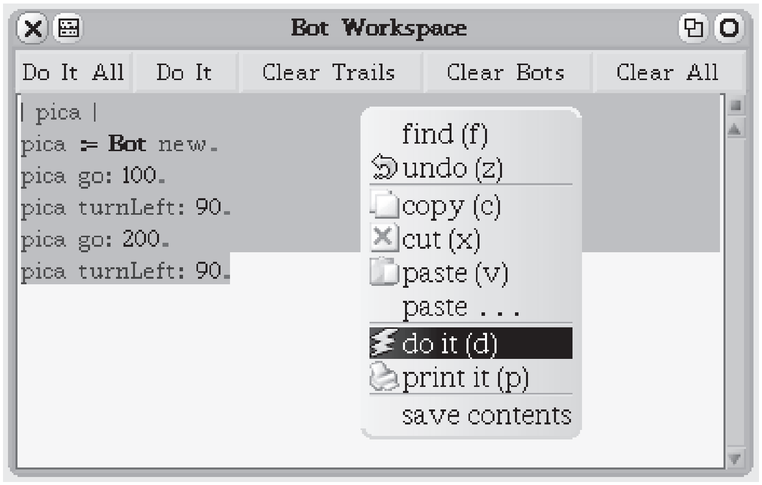
```
|pica|
pica := Bot new.
pica go: 10 + 30

50 + 90 140
```

Figure 5-9. *Left: Selecting the expression* `50 + 90`. *Middle: Opening the menu. Right: Executing the message and having the result printed.*

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 5-10. 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 5-9.



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

Figure 5-10. *Selecting a piece of a script and executing it explicitly using the menu*

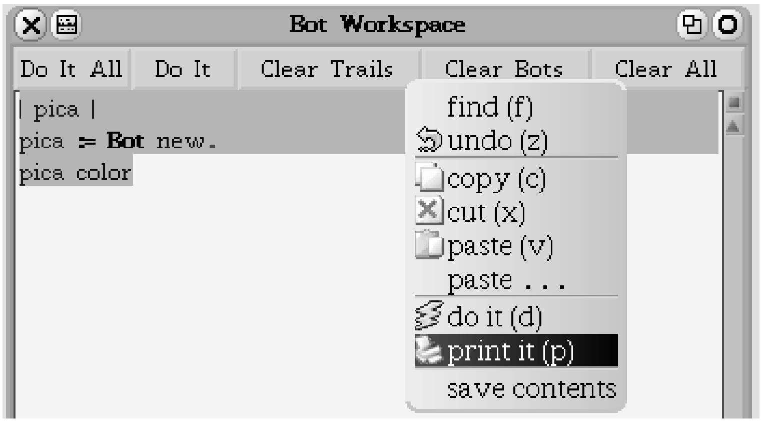
3. Using keyboard shortcuts. Select a piece of text, then press command-D on a Mac or alt-D on a PC.

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

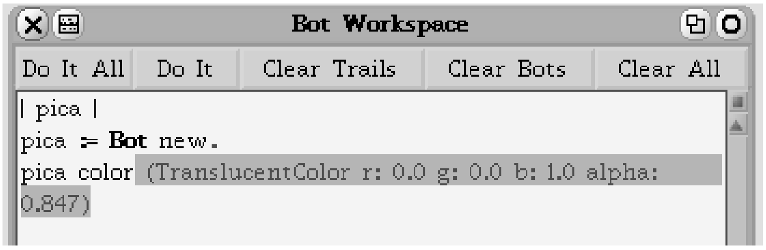
Two Examples

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 5-11. 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 5-12. 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.



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

Figure 5-11. *Open the menu and select the item **Print it (d)** to execute the selected piece of code and print the returned result.*



```
| pica |
pica := Bot new.
pica color (TranslucentColor r: 0.0 g: 0.0 b: 1.0 alpha:
0.847)
```

Figure 5-12. *The result of the message is printed as a textual representation of a color.*

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 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
```

■**Note** 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.

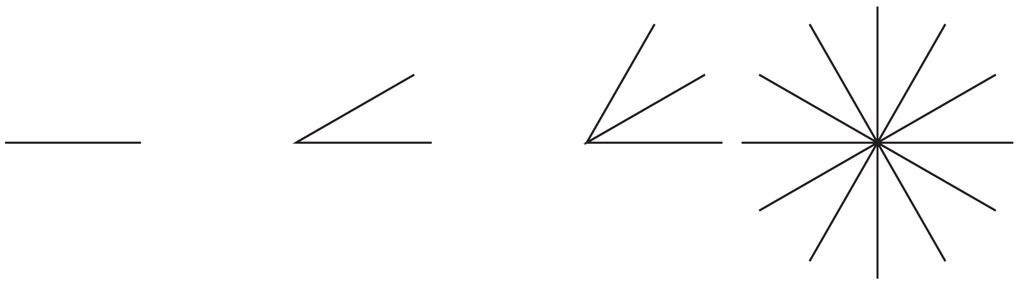
Summary

- 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 it** from 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 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 sixty 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]
```

Note *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`.

Note The argument of `timesRepeat:` is a ***block***, that is, a sequence of expressions surrounded by square brackets.

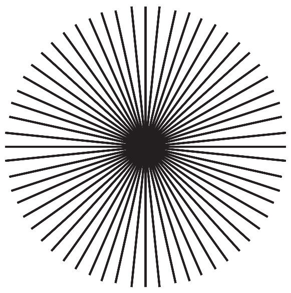
Loops at Work

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*

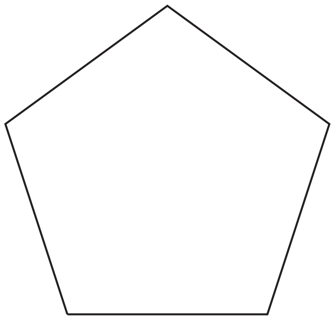
```
| pica |
pica := Bot new.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90.
pica go: 100.
pica turnLeft: 90
```

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.

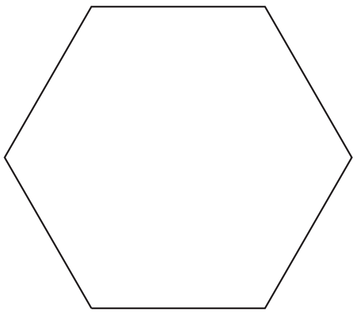
Experiment 7-3 (A Regular Pentagon)

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



Experiment 7-4 (A Regular Hexagon)

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

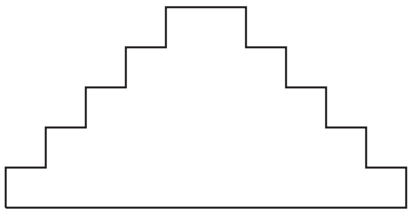


Once you have gotten the hang of it, try drawing a regular polygon with a very large number of sides. You may have to reduce the side length to make the figure fit on the screen. When the number of sides is large and the side length is small, the polygon will look like a circle.

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*

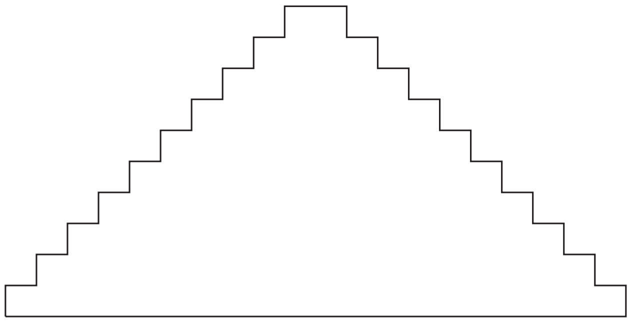


```
| pica |
pica := Bot new.
5 timesRepeat:
   [pica north.
   pica go: 20.
   pica east.
   pica go: 20].
5 timesRepeat:
   [pica go: 20.
   pica south.
   pica go: 20.
   pica east].
pica west.
pica go: 200.
```

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.



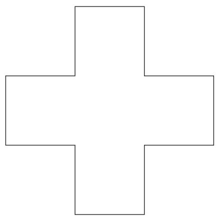
You may now want to generate pyramids with even larger numbers of terraces. The size of the terraces will have to be adjusted if you want them to fit on the screen.

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.

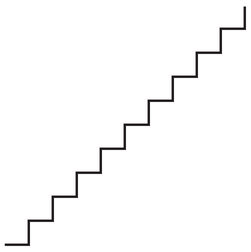
Experiment 7-6 (A Swiss Cross)

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



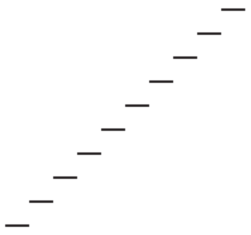
Experiment 7-7 (A Staircase)

Draw the staircase illustrated in the figure.



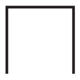
Experiment 7-8 (A Staircase Without Risers)

Draw the stylized staircase—with treads but without risers—illustrated in the figure.



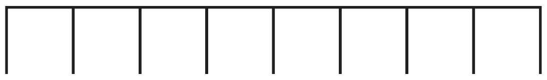
Experiment 7-9 (A Staple)

Draw the illustrated graphical element that looks like a staple.



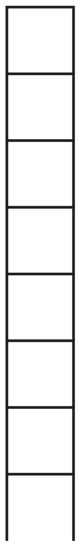
Experiment 7-10 (A Comb)

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:`	repeats a sequence of expressions *n* times	`10 timesRepeat:` `[` *a sequence of expressions* `]` `[pica go: 10.` `pica jump: 10]`

PLEASE NOTE I WASN'T SURE HOW TO LAY OUT THE ABOVE DIAGRAM.

—DIANA