CHAPTER 17

Strings, and Tools for Understanding Programs

In this chapter I present an important concept: the notion of strings. A string is a sequence of characters that can be used to represent words or sentences. One important use of strings is to communicate with users. In subsequent chapters, you will learn to use strings as a tool for understanding more complex program structures such as conditions and conditional loops. In this chapter I will present only the most important aspects of strings, providing the basic information that you will then use in subsequent chapters. I will also show you how to use strings to help you understand how programs are executed. I recommend that you also try to use the debugger for help in understanding the experiments that are proposed in this chapter.

Strings

Strings are used to represent information and present it to the user. Strings are delimited by single quotes (`'This is a string'`), and as you can see, a string can contain white-space characters. For example, the string `'squeak is cool'` represents a sequence of fourteen characters: s q u e a…. Note that a space is also a character. A string can contain any number of characters, including zero. Thus `''` is an empty string, `'a'` is a string with only the character a, and `' '` is a string whose only character is a space.

■**Important!** A string is a sequence of characters delimited by single quotes. A string represents textual information such as words or sentences. Strings can be used to display information on the screen.

Selecting a string and printing it (menu **print it**) prints that string. Several methods are defined on strings, but the most important one in the context of this book is the method `,` whose name is the comma character. When the message `,` is sent to a string as receiver with one string as argument, the method `,` returns the concatenation of the two strings. That is, it returns a single string whose characters are those of the first string followed by those of the second.

```
'squeak'                          "the value of a string is itself"
—Printing the returned value: 'squeak'

'a'                               "a string can be composed of only one character"

''                                "an empty string"

'squeak' , 'is cool'              "concatenating two strings"
—Printing the returned value: 'squeakis cool'

'', 'squeak', ' ', 'is cool'       "concatenating multiple strings"
—Printing the returned value: 'squeak is cool'
```

The method `copyReplaceAll:` allows you to modify a string by replacing all occurrences of a particular substring with a different string. In the example below, `'not'` is replaced with `'really'`:

```
'Squeak is not cool' copyReplaceAll: 'not' with: 'really'
—Printing the returned value: 'Squeak is really cool'
```

Communicating with the User

Squeak offers some tools to display information on the screen and to request information from the user. The class `PopUpMenu` allows you to bring up a menu and display some information for the user using strings. For example, the expression `PopUpMenu inform: 'squeak is cool'` causes a small window to pop up that displays the string `'squeak is cool'` and then waits until the user presses the **ok** button. The class `FillInTheBlank` allows you to request input from the user. For example, the expression `FillInTheBlank request: 'is squeak cool?'` brings up a dialog box with an input field and waits for the user to fill in the input field and press the **accept** button or the **cancel** button. The result of this expression is a string that represents what was typed by the user. You can also specify a default user input using the message `request:initialAnswer:` as shown in the following script, and in Figure 17-1.

```
(1) PopUpMenu inform: 'squeak is cool'

(2) FillInTheBlank request: 'is squeak cool?'
    —Printing the returned value: 'yes'

(3) FillInTheBlank request: 'is squeak cool?' initialAnswer: 'yes of course'
    —Printing the returned value: 'yes of course'
```

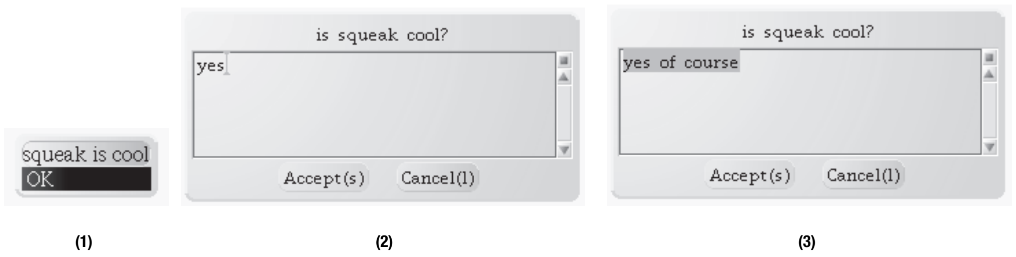


Figure 17-1. *Pop-up menu and fill in the blanks*

Strings and Characters

A string is composed of characters. While a string is enclosed in single quotes to show that it is a string, individual characters are prefixed by a dollar sign `$` to show that they are characters. For example, `$a` represents the character representing the letter “a”. Note that while individual characters are prefixed by the dollar sign `$`, when you edit a string you simply type the characters without the dollar sign.

There are several methods for accessing the individual characters of a string. For example, the methods `first`, `second`, and `third` return the first, second, and third characters of a string. The method `size` returns the number of characters in a string, while the method `at: aNumber` returns the character located at the specified position in the string. You can replace the character at a specified position with another character using `at: aNumber put: aCharacter`. The method `copyUpTo: aCharacter` returns the beginning of a string up to the first character that matches `aCharacter`. Here are some examples:

```
'squeak is cool' first
—Printing the returned value: $s

'squeak is cool' size
 —Printing the returned value: 14
```

```
'squeak' at: 5
—Printing the returned value: $a

'squeak is cool' at: 11 put: $f
—Printing the returned value: 'squeak is fool'

'squeakiscool' copyUpTo: $i
—Printing the returned value: 'squeak'

'squeak is cool' copyUpTo: Character space
—Printing the returned value: 'squeak'
```

To create a character that does not have a graphical representation, such as the space, tab, or carriage return character, you can send a message to the class `Character`. The messages `Character space`, `Character tab`, and `Character cr` return respectively the space, tab, and carriage return characters.

Script 17-1 shows how to insert a carriage return into a string. Note that the method `at:put:` does not return the modified string, but the character that was inserted. This is an example where the *effect* of the message and its *result* are clearly different. Printing the result of the message `'squeak is cool' at: 7 put: Character cr` does not illustrate the effect of the method. Therefore, we print instead the modified string. To review how to print results of a message send on the screen, see Chapter 5, "Pica's Environment."

Script 17-1. *Inserting a carriage return into a string*

```
|myString|
myString:= 'squeak is cool'.
myString at: 7 put: Character cr.
myString
—Printing the returned value: 'squeak
is cool'
```

A character can also be converted into a string by sending it the message `asString`. In Script 17-2, three strings are concatenated together. The middle one is the string `'a'` created by the message send `$a asString`.

Script 17-2. *A character is converted into a string, which is then concatenated with other strings.*

```
'sque', $a asString, 'k'
—Printing the returned value: 'squeak'
```

Strings and Numbers

A string can represent a number. For example, the *string* `'10'` is a textual representation of the *number* 10. However, a string is not a number. A string does not know how to perform any mathematical operation, and a number does not know how to behave as a string. For example, we cannot concatenate two numbers or add two strings. However, a number knows how to produce a string that *represents* it using the method `asString`. In addition, a string knows how to convert a representation of a number into a number using the method `asNumber`.

Thus there is a difference between the number `10` and the string `'10'`. The number `10` represents the mathematical quantity 10, while the string `'10'` represents the textual representation of the number 10 that consists of the two characters 1 and 0. The string `'10'` is composed of the two characters: `$1` and `$0`, and the string `'12'` is composed of the two characters `$1` and `$2`. Here are some illustrations of operations with strings and numbers:

```
10 , 12
-> error! a number does not know the message ,

'10', '12'
—Printing the returned value: '1012'

10 asString
—Printing the returned value: '10'

10 asString , 12 asString
—Printing the returned value: '1012'

'10' asNumber
—Printing the returned value: 10
```

■**Note** A string can represent a number, but such a string is not a number. For example, the string `'79'` is composed of the two characters: `$7` and `$9`. To obtain the string representing a number, send the message `asString` to it.

Using the Transcript

Squeak offers several powerful tools for understanding program execution, such as the debugger (see Chapter 15). Another tool is called `Transcript`, with which you can display information in the form of strings. To open a transcript window, drag the thumbnail that is available in one of the flaps onto the desktop, or you can choose the **transcript** item of the **open...** menu. This opens a window, as shown in Figure 17-2.

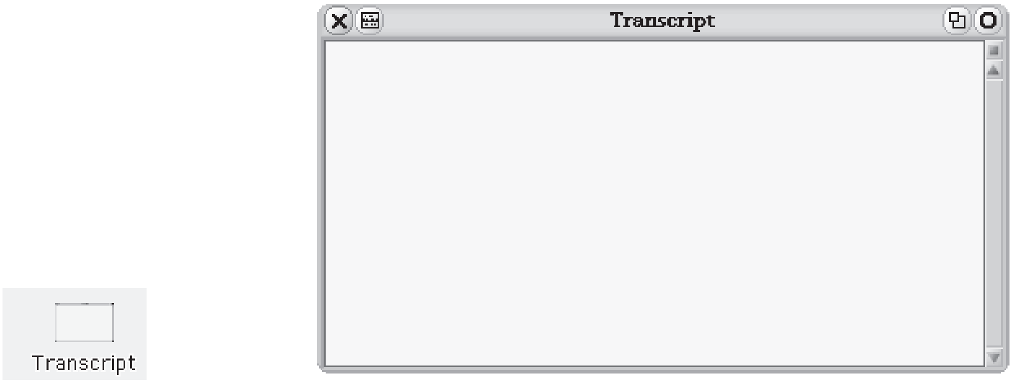
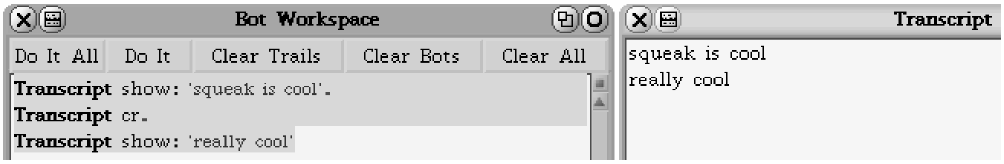


Figure 17-2. *To open a transcript window, drag and drop the thumbnail that you can find in a flap.*

There are two main messages for displaying information in a transcript window: `show:` and `cr`. The message `show: aString` displays the given string in the window, and the message `cr` inserts a new line. See Figure 17-3.



```
Transcript show: 'squeak is cool'.
Transcript cr.
Transcript show: 'really cool'
```

```
squeak is cool
really cool
```

Figure 17-3. *Writing to the transcript*

Script 17-3 gives some examples of displaying information in a transcript window.

Script 17-3. *Displaying information in a transcript window*

```
Transcript show: 'squeak is cool'.
Transcript cr.
Transcript show: 'really cool'
```

Note that a transcript window can display only strings. And so if you want to display a number, you have to obtain a string representing it using the method `asString`. This is illustrated in Script 17-4.

Script 17-4. *A number is converted into a string before it is displayed.*

```
Transcript show: '21 + 21 is: ', 42 asString ; cr
```

Generating and Understanding a Trace

Now I would like to show you how you can use `Transcript` to generate a trace of a program. A trace is a collection of indications of what is going on that is generated by a program. For example, you might want to track a robot's movements in a script by having the script print `'I am turning right'` every time the robot makes a right turn. To generate a trace, you simply introduce one or more expressions into your script that do not change the original execution of the program but, for example, display information in a transcript window. Let us begin with Script 17-5, which draws a staircase with treads of increasing length.

Script 17-5. *A staircase with treads of increasing length*

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

The first simple trace that we can generate tells us when the program is about to enter the `timesRepeat:` loop and when it has exited the loop. This is shown in Script 17-6.

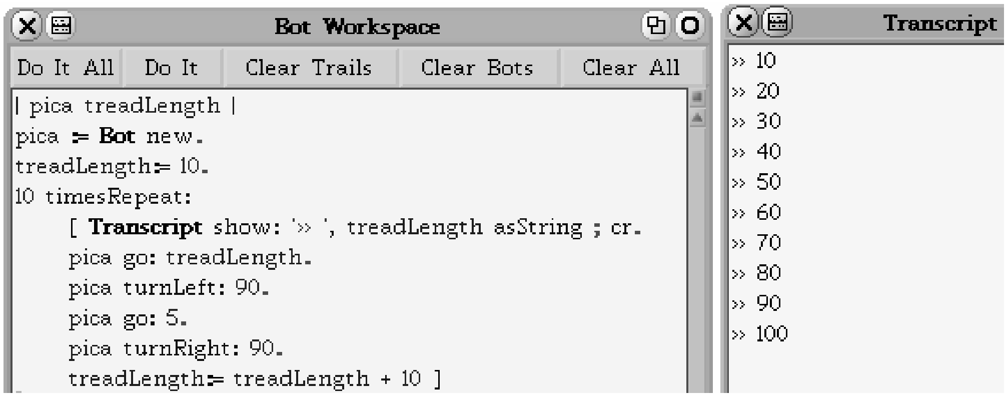
Script 17-6. *The staircase with a simple trace*

```
| pica treadLength |
pica := Bot new.
treadLength:= 10.
Transcript show: 'Before the loop' ; cr.
10 timesRepeat:
     [ pica go: treadLength.
     pica turnLeft: 90.
     pica go: 5.
     pica turnRight: 90.
     treadLength:= treadLength + 10 ].
Transcript show: 'After the loop' ; cr.
```

Experiment 17-1 (Putting a Trace inside the Loop)

Modify Script 17-6 by introducing the expression `Transcript show: 'inside the loop' ; cr.` inside the loop. Your transcript should now print `'inside the loop'` ten times, one for each pass through the loop. You can also insert the expression `self halt` to allow you to open the debugger. But watch out, or you will get ten debuggers!

Now I would like to use the same technique to generate a more sophisticated trace. For example, it would be nice to see how the value of the variable `treadLength` evolves while the program is executed. Script 17-7 contains a new expression that prints the value of the variable `treadLength` at the beginning of the loop each time it is executed. The results are shown in Figure 17-4.



```
| pica treadLength |
pica := Bot new.
treadLength:= 10.
10 timesRepeat:
     [ Transcript show: '>> ', treadLength asString ; cr.
     pica go: treadLength.
     pica turnLeft: 90.
     pica go: 5.
     pica turnRight: 90.
     treadLength:= treadLength + 10 ]
```

```
>> 10
>> 20
>> 30
>> 40
>> 50
>> 60
>> 70
>> 80
>> 90
>> 100
```

Figure 17-4. *Adding a trace to a script*

Script 17-7. *The staircase with a more sophisticated trace*

```
| pica treadLength |
pica := Bot new.
treadLength:= 10.
10 timesRepeat:
     [ Transcript show: '>> ', treadLength asString ; cr.
     pica go: treadLength.
     pica turnLeft: 90.
     pica go: 5.
     pica turnRight: 90.
     treadLength:= treadLength + 10 ]
```

Adding a trace after an assignment is often useful, since it reveals some key behavior of a program. For example, in Script 17-8, the expression `Transcript show: 'After := ' , treadLength asString ;cr.` has been added after the assignment statement that is the last expression of the loop. The trace, which is shown following the script, prints the value of the variable `treadLength` at the beginning and the end of the loop. These two values should be the same, and indeed, they are, as the trace shows.

Script 17-8. *The staircase with a trace after an assignment*

```
| pica treadLength |
pica := Bot new.
treadLength:= 10.
10 timesRepeat:
     [ Transcript show: 'treadLength: ', treadLength asString ; cr.
     pica go: treadLength.
     pica turnLeft: 90.
     pica go: 5.
     pica turnRight: 90.
     treadLength := treadLength + 10.
     Transcript show:' treadLength after := ' , treadLength asString ; cr. ]
```

And here is the trace:

```
treadLength:  10
treadLength after :=  20
treadLength:  20
treadLength after :=  30
treadLength:  30
treadLength after :=  40
treadLength:  40
treadLength after :=  50
treadLength: 50
treadLength after := 60
treadLength:  60
treadLength after :=  60
treadLength:  70
treadLength after :=  70
treadLength:  80
treadLength after :=  90
treadLength:  90
treadLength after :=  100
treadLength:  100
treadLength after :=  110
```

Summary

- A string is a sequence of characters delimited by single quotes: `'This is a string'`. A string represents textual information such as words or sentences and can be used to display information on the screen. For example, `'squeak is cool'` is a string of 14 characters.
- A character is one letter prefixed by the dollar sign `$`. Thus `$a` represents the character `a`.
- A string can represent a number, but such a string is not a number. For example, the string `'79'` is composed of the two characters: `$7` and `$9`. To obtain the string representing a number, send the message `asString` to it.
- A `Transcript` window is a small window used to display messages. The message `show: aString` displays the value of the argument `aString`, which must be a string, in the transcript window. The message `cr` adds a new line in the transcript window.