1

Tools to Understand Programs

1 Other Loops with Variables

In this section we would like to show you two new methods to:do: and to:do:by: that are handy when working with loops as you do not have to declare a loop variable, initialize it and increase it explicitly.

In the script **??**, we defined and initialized the variable length, then we increase ten times its contents by 10 that we show again in the script **??**.

Script 1.1 (*Creating a flat growing stair.)*

```
| caro length |
caro := Turtle new.
length := 10.
10 timesRepeat: [caro go: length.
                 caro turnLeft: 90.
                 caro go: 5.
                 caro turnRight: 90.
                 length := length + 10 ]
```

We can rewrite this script using the loop method to:do: as shown in the script **??**.

Script 1.2 (*Using to:do:)*

```
| caro |
caro := Turtle new.
1 to: 10 do: [:length |
                caro go: length * 10.
                caro turnLeft: 90.
                caro go: 5.
                caro turnRight: 90]
```

If we look at the difference between the two scripts, we see that we do not need to declare the variable length in the script. However, we have to declare the variable inside the block, *i.e.*, the sequence of messages we want to repeat. To declare a variable inside a block we prefix it with the character : and the variable declaration is terminated by the character |. Therefore the expression [:length | declares a variable named length in the block.

> The block passed to the message to:do: and to:by:do: require an argument.

The other difference that we can see is that we do not have to initialize the variable nor increase explicitly its value using an expression such as length := length + 10. Indeed what the loop method does for us is that it will assign values from 1 to 10 to the variable of the block, here length each time the block will be repeated. Hence the variable length will get the values 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 and the block will be repeated 10 times with one of these values assigned to length. Note that a loop does not have to start at 1 but can start at any number. The section **??** will present in detail these aspects.

With the loop method to:do: we were forced to multiply the length by 10 to get the distance that the turtle should move to draw the same stair. With the method to:by:do: we can avoid to that as this message allows one to specify the increment from one value to the other. In fact to:do: is equivalent to to:by:1:do:. The script **??** shows how the same method can be expressed. Note that here the values taken by the variable length are 10, 20, 30,... until 100 therefore we do not have to multiple length as in the other script. Notice also that we start at 10 and not 1 because we want that the first value taken by the variable is 10 and not 1.

Script 1.3 (*Using to:do:*)

```
| caro |
caro := Turtle new.
10 to: 100 by: 10 do: [:length |
                        caro go: length.
                        caro turnLeft: 90.
                        caro go: 5.
                        caro turnRight: 90]
```

> Block arguments are similar to method argument. [:length | ...] is a block defining an argument named length. :length is the argument name. As blocks may have several arguments, | ends the argument list.

Some technical details. The last argument of to:do: and to:by:do: is a block *i.e.*, sequence of messages like for the `timesRepeat:` method. However, for to:do: and to:by:do: the block requires one argument. Block arguments are similar to method argument. [:length | ...] is a block defining an argument named length. :length is the argument name. As blocks may have several arguments, | ends the argument list. Do not give name to a block argument the name of a variable already existing in the script or method containing the block.

2 Tools for Understanding

Now we present some practices to understand loops. Such pratices can be used in general to understand how your program work and they are useful to find errors. For this purpose we will present you the Transcript, a small window to which messages can be written, and show how we can write message into it. The subsequent section will then use this functionality with loops.

2.1 Transcript

The Transcript, with an uppercase letter, is a global variable whose value is a small window in which output traces can be written as shown in Figure **??**. A global variable is a variable that always exists,

it had been defined once for all and that can be accessed from anywhere in the system.

To open the Transcript bring the World menu and select the menu item 'open', then select the menu item 'transcript', you should get the window shown in Figure **??**. You also can pick the transcript thumbnail like you made for the script editor or browser in the tools flap on the right and drop it into your desktop.

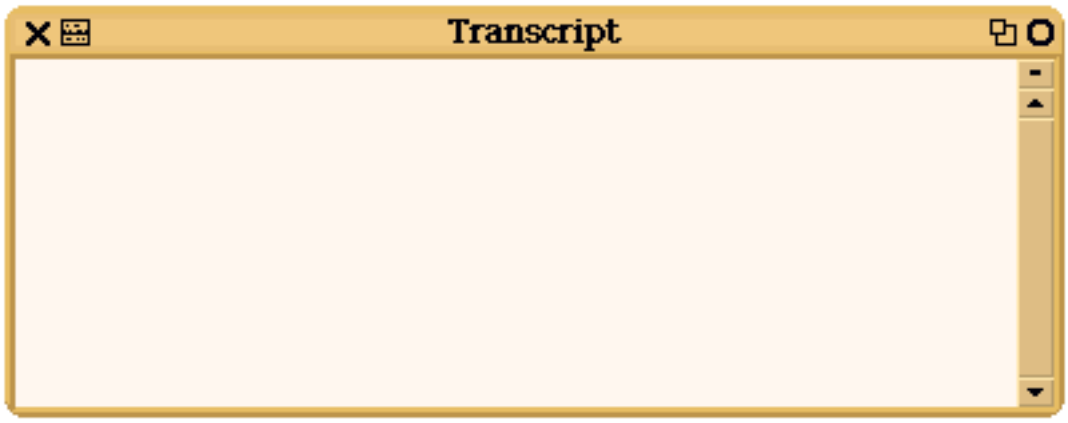

Figure 1.1: Here we have a Transcript

2.2 Using the Transcript

We can send messages to the Transcript so that it displays some texts. For example try the expression Transcript show: 'hello'. It asks the Transcript to print the text 'hello'. More precisely it asks the Transcript to display a string. 'hello' is a string composed by the characters $h, $e, $l, $l and $o. In Smalltalk a character is represented by the symbol $ and the character letter, but when we use " we get a string composed of characters without needing to use $.

The script **??** presents a sequence of messages that we suggest you to try line by line. You will learn that the Transcript simply displays the strings one after the other ones and that we have to put space to be sure that the strings are not touching each other. If you want the Transcript to display the new text to the next line, you should tell it using the message cr. Note that as we usually do not want to repeat Transcript in all the messages sent to it, we use a cascade to group messages as shown by the fourth line of the script **??**.

The message show: requires that you pass a string as argument, therefore if you want to print numbers you have to convert them into strings using the method printString. 12 printString returns '12' the string that has the character $1 as first element and character $2 as second element. The expression Transcript show: 12 printString displays the string '12' in the Transcript. Note that you can take several strings and turn them in a single one by sending the message , to a string and passing the other as argument. The expression 'Squeak',' is ', 'fun' appends the three strings and returns the string 'Squeak is fun'.

Script 1.4 (*Some expressions to write in the Transcript)*

```
Transcript show: 'hello'.
Transcript show: ' on the same line'.
Transcript cr.
Transcript show: 'on another line it is better'; cr.
Transcript show: 'Squeak' , ' is ', 'fun' ;cr
```

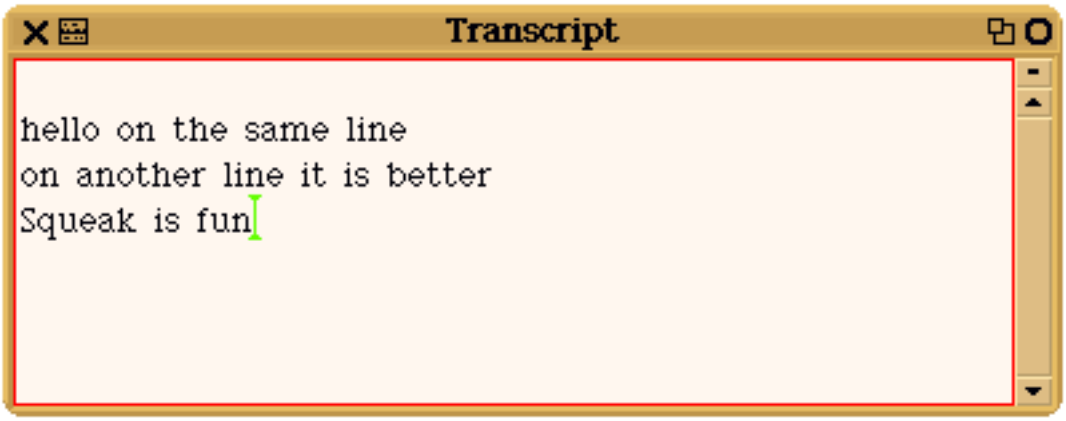

Figure 1.2: A trace in the Transcript showing the value of the variable length at the beginning and the end of the loop.

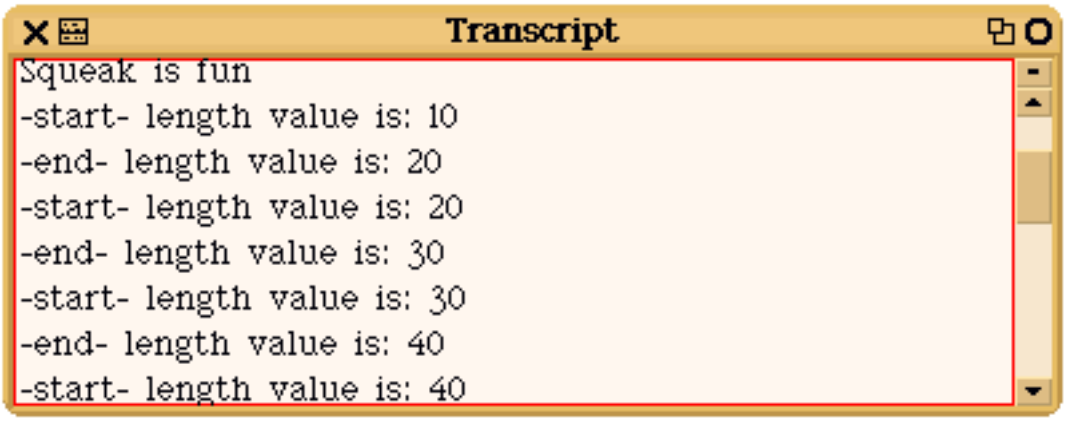

Figure 1.3: A trace in the Transcript showing the value of the variable length at the beginning and the end of the loop.

2.3 Trace in Loops

Using the Transcript functionalities we can now add trace generation to the Script **??** as shown in the Script **??**. Executing now the script produces in the Transcript the trace shown in Figure **??**.

Script 1.5 (*Adding a trace to a script)*

```
| caro length |
caro := Turtle new.
length := 10.
10 timesRepeat: [ Transcript show: '-start- length value is: ',
                                                  length printString ; cr.
                caro go: length.
                caro turnLeft: 90.
                caro go: 5.
                caro turnRight: 90.
                length := length + 10.
                Transcript show: '-end- length value is: ',
                                              length printString.
                Transcript cr]
```

We suggest you to do the same with the scripts **??** and **??**.

Script 1.6 (*Adding a trace to a script)*

```
| caro |
caro := Turtle new.
10 to: 100 by: 10  do:
             [:length |
                Transcript show: 'length: ', length printString; cr.
                caro go: length.
                caro turnLeft: 90.
                caro go: 5.
                caro turnRight: 90]
```

2.4 Experimenting with Loops

Now we want to show you that using the right message for loops can really simplify your programs and the way you solved them. Imagine that we want to generate lengths from 14 pixels to 100 pixels 2 by 2. You could write it using `timesRepeat:` the following way:

```
|len|
len := 14.
43 timesRepeat: [ Transcript show: len printString ; cr.
                len := len + 2]
```

You may wonder where the 43 comes from. It comes from the following calculation 43 = 100 - 14 / 2. As you see this is not as straightforward as the next solution to the problem using to:by:do:

```
14 to: 100 by: 2 do: [:len | Transcript show: len printString ; cr.]
```

The loop is executed until the specified upper limit is reached, the limit itself included. For example, the following script produces 0, 3, 6, 9 and 12, while the following one only produces 0, 3, 6, 9 as 12 the next value is larger than the limit, 10, we specified.

```
0 to: 12 by: 3 do: [:len| Transcript show: len printString;cr]

0 to: 10 by: 3 do: [:length| Transcript show: length printString;cr]
```

As we mentioned earlier, to:do: is equivalent to to:by:do: the following scripts both produce 1, 2, 3, 4 and 5.

```
1 to: 5 do: [:length | Transcript show: length printString ; cr.]
1 to: 5 by: 1 do: [:length | Transcript show: length printString ; cr.]
```

Note that with to:do:by: you can also give negative numbers. The following script produces 4, 2, 0, -2 and -4.

```
4 to: -4 by: -2 do: [:i | Transcript show: i printString ; cr.]
```

Summary

- The block passed to the message to:do: and to:by:do: require an argument.
- Block arguments are similar to method argument. [:length | ...] is a block defining an argument named length. :length is the argument name. As blocks may have several arguments, | ends the argument list.
- Do not give name to a block argument the name of a variable already existing in the script or method containing the block.