Chapter

8

Control Structures

Modern programming languages provide ways to execute code conditionally and to repeat blocks of code. In C, we have `if () {}`, `if () {} else {}`, `do {} while ()`, `while () {}`, and `for (;;) {}`. These types of construct are usually part of the language.

In Smalltalk however, the language itself does not have any of these constructs. Instead, they are created by sending messages to objects. The result of this is that if you don't like the way a control message works, you can change it (not recommended!). If you want a new type of control structure, you can write it. For example, Smalltalk does not come with a switch/case message, so if you feel the need for one, you can write it (again, not recommended for reasons we'll see in Chapter 28, Eliminating Procedural Code). In fact, you probably won't need any control structures other than those that come standard with the Smalltalk library. Let's go through these.

Conditional Execution

The simplest type of conditional execution is to only execute a particular block of code if some condition is true (or false). In Smalltalk, there are two messages, one for when the condition is true and one for when it's false. Both messages are sent to an instance of *Boolean* (ie, to *true* or *false*). Here is the general approach, with an example.

```
booleanValue ifTrue: [some code].
booleanValue ifFalse: [some code].

3 < 4 ifTrue: [Transcript cr; show: 'True']
```

We can extend this to executing some code if the condition is true and other code if the condition is false. In Smalltalk we write one of the following. The `ifTrue:` and `ifFalse:` blocks can be in either order. For example, if the `ifTrue:` block is long enough to obscure a later `ifFalse:` block, put the `ifFalse:` block first.

```
booleanValue
   ifTrue: [some code]
   ifFalse: [some code].
```

```
booleanValue
   ifFalse: [some code]
   ifTrue: [some code].
```

For example,

```
3 < 4
   ifTrue: [Transcript cr; show: 'True']
   ifFalse: [Transcript cr; show: 'False'].
```

Smalltalk doesn't provide an easy way to do C's `if () {} else if () {} else {}` type of construct. You can do it with nesting such as:

```
booleanValue
   ifTrue: [some code]
   ifFalse: [booleanValue2
      ifTrue: [some code]
      ifFalse: [booleanValue3
         ifTrue: [some code]
         ifFalse: [some code]]]
```

However, this type of code is very procedural rather than Object-Oriented. If you find that you are writing code like this, it may be time to rethink how you are doing things. See Chapter 28, Eliminating Procedural Code for more information.

Looping

Smalltalk has no looping constructs in the language. Instead, it provides looping functionality by sending messages to BlockClosures. The most basic type of loop is one that continues to loop while some condition is true. As long as the block (ie, the last statement in the block) evaluates to *true*, the loop will continue. This type of mechanism is similar to C's `do {} while ( );` construct. Here are the syntax and an example.

```
[some code] whileTrue.

count := 0.
[Transcript cr; show: count printString.
count := count + 1.
count < 10] whileTrue.
```

The condition can be reversed so that the loop continues as long as the block evaluates to *false*.

```
[some code] whileFalse.
```

The next type of looping mechanism is similar to the above, but if the first block evaluates to *true*, a second block of code is executed. This type of mechanism is somewhat similar to C's `while () {}` but is more powerful because it allows multiple statements in the first block. Again, here are the syntax and an example.

```
 [some code]
   whileTrue:
      [more code]

[Dialog confirm: 'Continue']
   whileTrue:
      [Transcript cr; show: 'Continuing'].
```

As before, you can reverse the conditions so that the loop continues if the first block evaluates to *false*.

```
[some code]
   whileFalse:
      [more code]
```

The final straight looping mechanism is one that simply repeats. This means that the code in the block needs a way to return from the method, or to somehow stop execution or you will find yourself in an infinite loop.

```
[some code] repeat

[(Dialog confirm: 'Continue') ifFalse: [^self].
Transcript cr; show: 'Continuing'] repeat.
```

Repetition

In Smalltalk, there are three ways to repeat a block of code a fixed number of times. Two of them pass in the index, while the other one assumes that you don't care which iteration you are on. If the loop doesn't care about the index number, the simplest mechanism is to send `timesRepeat:` passing the code block as the parameter. For example:

```
5 timesRepeat: [Transcript cr; show: 'hello']
```

If the loop needs to know what iteration it is on, you'll need to send one of the following two messages. If you want to loop from one number to another, incrementing by one each time, send `to:do:` passing the code block as a parameter. The block expects to receive the index number as a parameter. For example,

```
1 to: 5 do: [ :index | Transcript cr; show: index printString]
```

If you need a loop like this but don't want to increment by one, you can specify the step value by sending `to:by:do:`. Again, this message expects the block of code as a parameter, and the block should expect to receive the index number as a block parameter. For example,

```
15 to: 1 by: -2 do: [ :index | Transcript cr; show: index
printString].
```

Despite the existence of these messages, they are less useful than you might expect. Most repetitive looping is done over collections of objects, and collections have some powerful mechanisms for looping. We'll talk a lot more about looping over collections in Chapter 11, Collections.

Optimized Messages

BlockClosures are used in the above control mechanisms usually as parameters, but sometimes as the receiver of the message. The BlockClosures can be stored in variables or can be written as literal blocks — ie, as code with square brackets around it.

If all the BlockClosures involved in the control message are literal blocks, the message is compiled in-line and so no message send occurs. On the other hand, if any of the BlockClosures involved in the message are

variables rather than literal blocks, the code will not be in-lined and the message will be sent. Usually these messages are compiled in-line since most blocks are literal blocks.