1
Conditions

Up until now the programs you defined executed *all* the expressions they contained, them one after the other. You had no way to say that certain messages should only be executed when certain *conditions were met*. This chapter and the next one introduce an important programming concept: the notion of *conditional* execution, that is the fact that a certain piece of code executes only when a specified condition holds. A condition is an expression that can be true or false.

This chapter starts with defining a simple problem that shows the need for conditional execution. Then it presents that a conditional expression is composed of a *condition* and a *conditional block* — that is a sequence of expressions that are conditionally executed.

1 A Simple Problem

Suppose you want to change the color of a robot depending on its distance from the center of the screen. If a robot is less than 200 pixels from the center, it should be red otherwise it should be green. This problem requires a *conditional* execution. Depending on a condition — the robot's location — its color should change.

The method distanceDetector shows a possible solution, and script 1.1 shows how the method distanceDetector is used.

Script 1.1 (*A Simple Detector)*

```
pica := Bot new.
pica jump: 20.
pica distanceDetector.
pica jump: 200.
pica distanceDetector.
```

Method 1.1

distanceDetector

```
  | distanceFromCenter |
  distanceFromCenter := self distanceFrom: World center.
  distanceFromCenter < 200
    ifTrue: [ self color: Color red ]
    ifFalse: [ self color: Color green ]
```

Let's analyze what happens when the expression pica distanceDetector is executed.

1. The expression self distanceFrom: World center computes the distance from the receiver to the center of the screen. This distance is stored into the variable distanceFromCenter.

2. Then the expression distanceFromCenter < 200 ifTrue: [self color: Color red] ifFalse: [self color: Color green] is executed as follows: *if* the distance is smaller than 200 the color of the receiver is changed to re; otherwise it is changed to green. This expression is a *conditional* expression. It is spread over three lines in the method definition but it is all one expression, because there is no period to terminate it earlier.

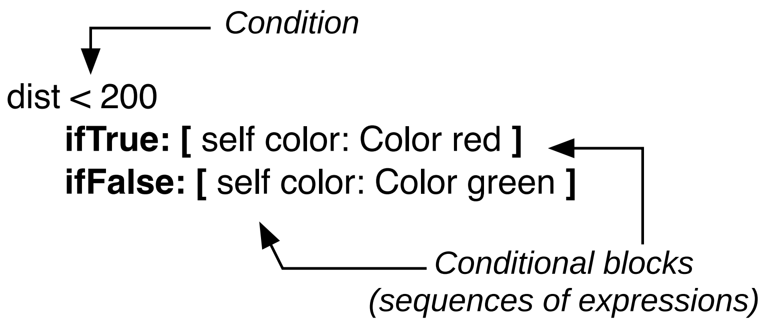
Condition
dist < 200
ifTrue: [self color: Color red]
ifFalse: [self color: Color green]
Conditional blocks
(sequences of expressions)

Figure 1.1: A conditional expression is composed of a condition and conditional block (a conditional sequence of expressions).

A conditional expression is composed of two parts: a *condition* and *conditional block* that is a sequence of expressions. The expression distanceFromCenter < 200 is a condition and the expressions [self color: Color red] and [self color: Color green] are conditional blocks as shown by Figure 1.1.

The method ifTrue:ifFalse: executes one condition, here distanceFromCenter < 200, and depending of its value, executes one of the conditional blocks and skip the other. The keyword ifTrue: indicates that the conditional block [self color: Color red] is only executed when the condition is true. Similarly the keyword ifFalse: indicates that the conditional block [self color: Color green] is only executed when the condition is false. The conditional blocks are also called the condition *branches*. Imagine that you would follow the trunk of a tree with your finger, each time you encounter a branch you would have to choose which path you want to follow, and you would follow only one at a time. The term branch refers to this situation. It represents the fact that the execution will have to choose between branches and only execute one at a time.

So there are different kinds of expressions. Some such as self distanceFrom: World center are always executed while others are executed only when their associated condition holds.

A condition is not limited to one single expression but can contain a composition of expression as I will present in Chapter **??**. The method ifTrue:ifFalse: defines two possible blocks, which can each contain a sequence of expressions. Note that the method ifTrue:ifFalse: is a *single* method with two arguments, one for the true case and one for the false case. Therefore you should not put a period after the] following the ifTrue: as it will break the conditional statement by ending it too soon, causing an error.

Adding a Trace to Understand. To understand how conditional expressions are executed, I suggest you use the Transcript and send it messages to generate trace as presented in Chapter **??**. You can also use the debugger by inserting the expression self halt as shown in the Chapter **??**. If you want to know whether a particular branch is executed, introduce expressions such as Transcript show: ’here’ ; cr in the branch. Method 1.2 presents one way to generate such a trace in the context of the distanceDetector method. Depending on the position of the robot different traces will be produced. Try to guess them before executing script 1.1. Do not hesitate to add and modify such a trace in all the scripts you have problem to understand.

Method 1.2

```
distanceDetector

  | distanceFromCenter |
  distanceFromCenter := self distanceFrom: World center.
  Transcript show: ’always’; cr.
  distanceFromCenter < 200
    ifTrue: [ self color: Color red.
          Transcript show: ’red’ ; cr ]
    ifFalse: [ self color: Color green
           Transcript show: ’green’ ; cr ]
```

About the Value Returned by a Method. When I presented how to define methods in Chapter **??** I explained that executing a method not only evaluates the messages it contains but also *returns* a value. Up until now you did not really used the result returned by methods. Now for the condition expression we are essentially interested by the result returned by the method. For example in method 1.1, the expression self distanceFrom: World center not only computes the distance of the receiver from the center of the screen but *returns* it. The condition expression, distanceFromCenter < 200, uses this value to decide wich branch should be executed. Note that the expression distanceFromCenter < 200 also returns a value telling whether the distance is smaller or not than 200 (see Chapter **??**).

@@— production the following should be in a frame—@@ A conditional expression is composed of two parts: a *condition* and *conditional messages*.

aCondition
  ifTrue: [*expressionsIfConditionIsTrue* **]**
  ifFalse: [*expressionsIfConditionIsFalse* **]**

@@— until here—@@

2 Two Other Conditional Messages: ifTrue: and ifFalse:

Sometimes you only need to perform one action when a specific condition is true but do nothing when the condition is false (or vice versa). For example the method redWhenCloseToCenter (1.3) only changes the color of the receiver to red when it is at a distance smaller than 200 pixels from the screen center.

Method 1.3

redWhenCloseToCenter

```
| distanceFromCenter |
distanceFromCenter := self distanceFrom: World center.
distanceFromCenter < 200
  ifTrue: [ self color: Color red ]
  ifFalse: [ ]
```

Using the method ifTrue:ifFalse: you just have to leave the second branch empty that is [ ]. However, Smalltalk provides two other methods ifTrue: and ifFalse: to express these kinds of conditional expressions. The method ifTrue: executes its conditional block when its condition is true. Using the method ifTrue: I rewrite the method redWhenCloseToCenter shown in 1.3 as shown in method 1.4.

Method 1.4

redWhenCloseToCenter

```
| distanceFromCenter |
distanceFromCenter := self distanceFrom: World  center.
distanceFromCenter < 200
  ifTrue: [ self color: Color red ]
```

Contrary to ifTrue:, the method ifFalse: executes its conditional block when its condition is *false*. The methods ifFalse: and ifFalse: both execute a condition and, depending on the value returned by the condition, either execute or skip the conditional block.

@@— production the following should be in a frame—@@ The method ifTrue: executes its conditional block when its condition is true. The method ifFalse: executes its conditional block when its condition is false.

aCondition
  ifTrue: [*expressionsIfConditionIsTrue* **]**

aCondition
  ifFalse: [*expressionsIfConditionIsFalse* **]**

@@— until here—@@

A Subtle Point. The difference between using ifTrue:ifFalse:, and ifTrue: followed by ifFalse: is that the *condition* using ifTrue:ifFalse: is only executed once, while with ifTrue: followed by ifFalse: the conditions of the *two* conditional expressions are executed. This can be a problem when the conditional block of the first conditional expression (ifTrue:) modify what is tested by the condition of the second conditional expression (ifFalse:). In such a case using ifTrue: followed by ifFalse: is not equivalent to using ifTrue:ifFalse:.

3 Nesting Conditional Expressions

A conditional expression can contain any other expressions and in particular other conditional expressions. This is what I demonstrate next. There is nothing spectacular about it, but it is common — and that is why I want to show it to you. Conditions can be nested inside conditions.

Another Simple Problem. Let's modify the previous problem. Now if a robot is less than 200 pixels from the center it should be red; when it is between 200 and 300 pixels away, it should be yellow; and at a distance greater than 300 it should be green.

In this problem, different parts of the method should be executed under different circumstances. The color should change to yellow under different conditions that are different than changing the color to green. A possible solution to our problem is shown by method 1.5.

Method 1.5

```
setThreeColor

  | distance |
  distance := self distanceFrom: World  center.
  distance > 300
    ifTrue: [ self color: Color green ]
    ifFalse: [ distance < 200
      ifTrue: [ self color: Color red ]
      ifFalse: [ self color: Color yellow ] ]
```

Method 1.6 contains the exact same code with highlighting added to show the conditional expressions. There are two different conditional expressions. The first one is shown in italics and the second in bold as shown in the method 1.6. The second conditional expression (in bold) is only executed when the condition of the first conditional expression is false. When the distance is smaller than 300 the conditional expression 2 is executed. That means that its condition is executed and, depending on its values the correct branch is executed.

Method 1.6

```
setThreeColor

  | distance |
  distance := self distanceFrom: World  center.
  distance > 300
    ifTrue: [ self color: Color green ]
    ifFalse: [ distance < 200
      ifTrue: [ self color: Color red ]
      ifFalse: [ self color: Color yellow ]]
```

Condition 1:

```
distance > 300
    ifTrue: [ self color: Color green ]
    ifFalse: [ ... ]
```

Condition 2:

```
distance < 200
      ifTrue: [ self color: Color red ]
      ifFalse: [ self color: Color yellow ]
```

If you have trouble to identify this two conditional expressions, pick some particular values for the distance (like 150, 250 or 350). Take a colored pen and underline the part of each method that will be executed. Following each step carefully will show that only certain branches are executed. I suggest you introduce different traces to understand how the conditions are executed or using the debugger stepping slowly into the method.

4 Learning from Errors

As people are always making mistakes, looking at errors is an excellent way to learn and understand a concept from another perspective. I defined the method coloredTurn: anAngle that changes the color of a robot according to the direction it is heading. When the robot points to the north it should turn blue to represent cold. it should turn red when it points to the south, and otherwise it should be green. Our first try at defining of this method is shown in method 1.7.

Method 1.7

```
coloredTurn: anAngle
   "change the color of the robot so that it is blue aiming
   at the north and red to the south"

   self turn: anAngle.
   self direction = 90
      ifTrue: [ self color: Color blue ].
   self direction = -90
      ifTrue: [ self color: Color red ]
      ifFalse: [ self color: Color green ]
```

The above definition is not correct. Try to understand why before reading any farther. The above method is wrong because when the robot is pointing to the north, and its color is green. But it should be blue, as shown in the script 1.2.

Script 1.2 (*Illustrating the bug)*

```
| pica |
pica := Bot new.
pica coloredTurn: -90.
pica color —Printing the returned value:    Color red          "ok"
pica coloredTurn: 90.
pica color —Printing the returned value:   Color green.      "ok"
pica coloredTurn: 90.
pica color —Printing the returned value:   Color green      "wrong"
```

Why... Execute the method 1.7 mentally to identify why this method is wrong. The problem is that even if the condition self direction = 90 is true and its associated block is executed, the method continues and evaluates the false conditional block of the last conditional expression, changing the color of the robot to green. The following commented version of the code illustrates this.

```
pica coloredTurn: 90.

  self direction = 90                      "is true"
     ifTrue: [ self color: Color blue ]    "so the true conditional message is executed"
                                           "the robot becomes blue and evaluate the following"
  self direction = -90                     "is false"
     ifFalse: [ self color: Color green ]  "so the false conditional message is executed"
```

The solution... To solve the problem you have to be sure that all the code follows the right conditions and in particular that certain code is not executed. Hence, you have to nest code under the correct condition as shown in the method 1.8.

Method 1.8

```
coloredTurn: anAngle
  "change the color of the robot so that it is blue aiming at the
  north and red to the south, green else"

  self turn: anAngle.
  self direction = 90
    ifTrue: [ self color: Color blue ]
    ifFalse: [ self direction = -90
      ifTrue: [ self color: Color red ]
      ifFalse: [ self color: Color green ] ]
```

5 Interpreting a Mini Language

In theoretical biology, researchers have developed systems called Lindermeyer systems for studying the growth of plants. Lindermeyer systems are based on robot graphics similar to the robot you use. In addition the robot understands a mini-language composed of characters such as $g and $t. A robot action is associated with each of these characters. For example the character $g is associated with moving forward, and $t with turning of 45 degrees. A Lindermeyer system generates a sequence of characters. These characters are then interpreted by a robot and the actions it takes produce pictures.

Define the method interpret: aCharacter that makes the robot either move forward when the character is $g, or turn 45 degrees when the character is $t as illustrated in the script 1.3. The script 1.3 illustrates how this method is used.

Script 1.3 (*Using interpret: aCharacter)*

```
| pica |
pica := Bot new.
4 timesRepeat:
  [ pica
     interpret: $g;
     interpret: $t;
     interpret: $g;
     interpret: $g;
     interpret: $t;
     interpret: $g ]
```

The method interpret: can be defined this way:

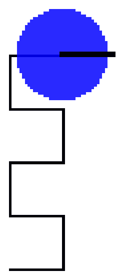

Figure 1.2: A picture generated using the method interpret: with the sequence of characters ’gttgttgtttttgttttttgttgttgttttttgttttttg’.

Method 1.9

```
interpret: aCharacter

  aCharacter = $g
    ifTrue: [ self go: 20 ]
    ifFalse:
      [ aCharacter = $t
          ifTrue: [ self turn: 45 ] ]
```

Try to define reproduce Figure 1.2. You will build a complete Lindermeyer in the following book. As a string is a sequence of characters I can represent a picture as a string. For now use the following script 1.4 which repeatly sends the message interpret: to the robot with each character and change the string ’gttgttgtttttgttttttgttgttgttttttgttttttg’ to your own idea to create pictures.

Script 1.4 (*Using interpret: in a loops)*

```
| pica |
pica := Bot new.
’gttgttgtttttgttttttgttgttgttttttgttttttg’
   do: [ :aChar | pica interpret: aChar ]
```

Further Experiments. Enhance the method interpret: aCharacter so that either $g or $G will make the robot goes forward, and either $t or $T will make it turn 45 degrees. Also add that the character $+ make the robot turn left, and $- to turn right.

6 Summary

- A conditional expression is composed of two parts: a *condition* and *conditional blocks*. The conditional blocks are executed depending on the value of the condition.

Method	Description
aCondition ifTrue: [*expressionsIfConditionIsFalse*]	Execute expressionsIfConditionIsTrue only if aCondition is true. If a robot is pointing to the north, it turns green. self direction = 90 ifTrue: [self color: Color green]
aCondition ifFalse: [*expressionsIfConditionIsFalse*]	Execute expressionsIfConditionIsFalse only if aCondition is false. The system beeps only when the robot is not pointing to the north. self direction = 90 ifFalse: [Beeper beep]
aCondition ifTrue: [*expressionsIfConditionIsTrue*] ifFalse: [*expressionsIfConditionIsFalse*]	Execute expressionsIfConditionIsTrue whether aCondition is true otherwise execute expressionsIfConditionIsFalse. self direction = 90 ifTrue: [self color: Color green] ifFalse: [Beeper beep]