Learning Programming in Squeak

Draft

Stéphane Ducasse
ducasse@iam.unibe.ch

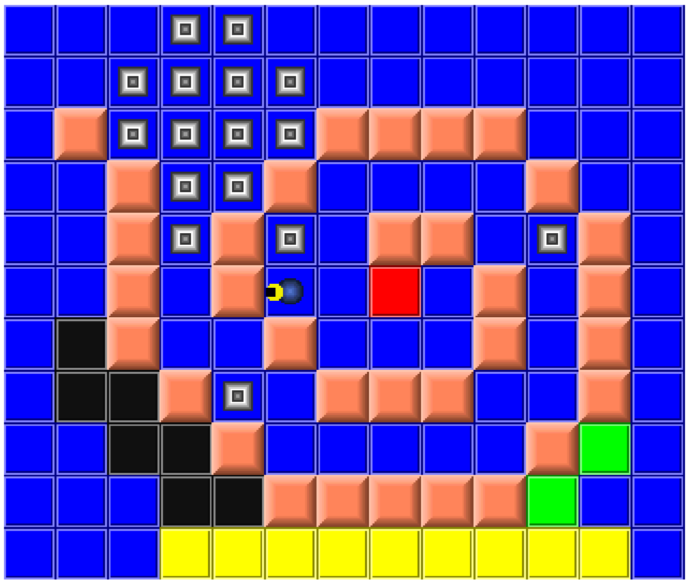
©-2002
To be published by Morgan Kaufman Publishers in 2003

May 23, 2003

Contents

1

Bots, Inc.



In this chapter we describe the environment of the robots, their behavior and capacities. We show how simple scripts can be defined using the environment and how they can be turned into methods as shown in Chapter **??**.

1 The Robot's World

The world of our robots is an area composed of different kind of tiles as shown by the first figure of this chapter.

- Blue tiles represent the ground. Robots can freely walk on them.
- Red tiles represent the *starting place*. This is on this tile that new robots appear. There is one and only one red tile per area. When you will be defining new areas you should always specify a starting place.
- Black, yellow, magenta, and green tiles are just painted ground tiles. However the robots have sensors to know if they are on of such tiles.
- Painted tiles (blue, red, black, yellow, and green) can contain diamonds and robots can drop diamonds on them.
- Light brown tiles represent bricks. Robots are blocked by bricks and therefore cannot walk on them. A robot has sensors that indicates it whether it is facing a wall.

A robot can walk and drop diamonds on all the tiles except the bricks. It cannot walk outside the world limits.

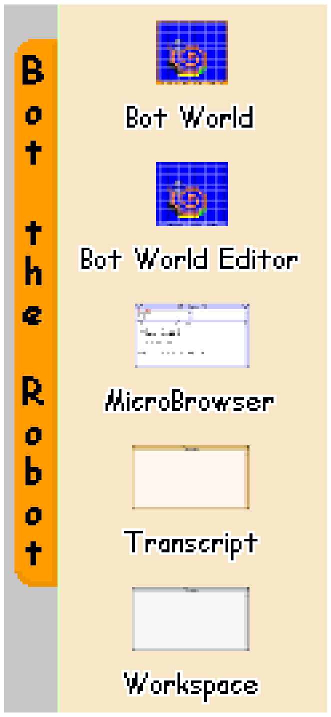


Figure 1.1: The flap containing the environment and the tools to program the robots.

2 Getting Started

To create a robot world grab and place on the Squeak desktop the first thumbnail of the orange flap named Bot World as shown by the Figure 1.1) or execute the script 1.1 in a workspace.

Script 1.1 (*Opening the World of Bot)*

```
BotWorldBoard newStandAlone openInWorld
```

You should obtain an environment similar to the one displayed in the first figure of the chapter. The bottom button bar allows you to access the functionality of this environment as shown by the Figure 1.2.

Figure 1.2: The functionality of the bot environment.

The Bot World window has the following buttons from left to right:

- **Add Bot.** Add a new robot in the current area. Once this button pressed the user is asked to give a name for the newly created bot. This name is also used as a variable to send messages to the robot using a robot Controller (see Figure 1.8). A robot appears on the starting place (the red tile) of the board.
- **Controller.** Open a robot Controller, *i.e.*, a dedicated workspace in which we can send messages to the robots by using their names (see Figures 1.4 and 1.8).
- **Restart.** To reset the area but without changing the bots that are already created.
- **Pick Area.** To select a new area among the list of all the areas defined.

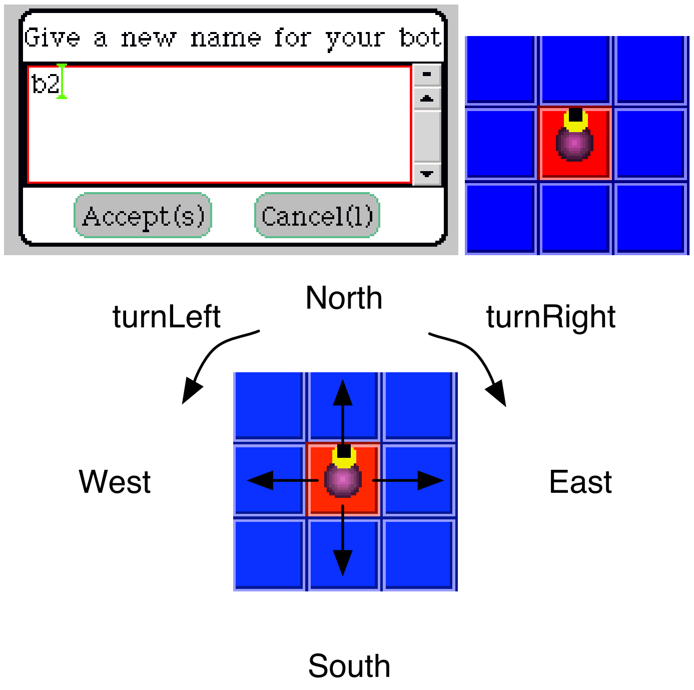


Figure 1.3: Giving a name to a new robot and getting a new robot on the starting place.

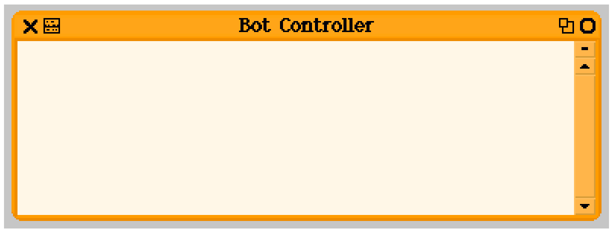


Figure 1.4: Opening a bot controller by pressing the button ’Controller’.

- **Quit.** To quit and close the playing board.

3 Robot Behavior

A robot is always on one tile and can only move from one tile to another one. Note that each tile has a location and that you can see the bot location by moving the mouse on the bot and waiting for a balloon to show up as shown in Figure 1.6. A bot can only move forward and turn in four directions as shown by the Figure 1.5 but we will show how to define more advanced operations such as moving back in the future.

A robot understand the following messages:

- `go`. In response to this message, the receiver moves by one tile in the direction in which it is pointing at. What is *really* important is that we do not want to damage our robot by making it crashing into a wall. Therefore you have check using the method `canMoveForward` if the move is possible as we will show later.

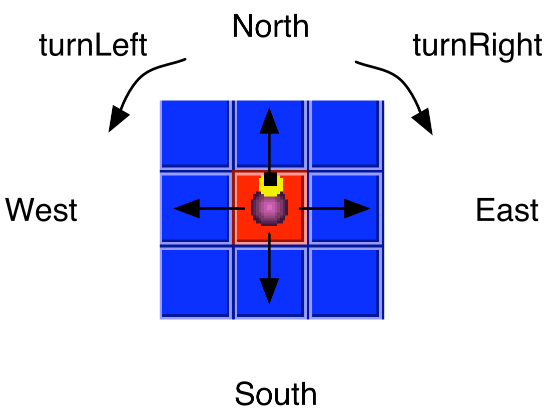


Figure 1.5: Possible robot movements.

- `turnLeft` and `turnRight`. In response the receiver changes its direction to the left or right relative of the current direction.
- `north`, `south`, `east`, and `west`. In respond the receiver changes its direction to point to the corresponding directions.

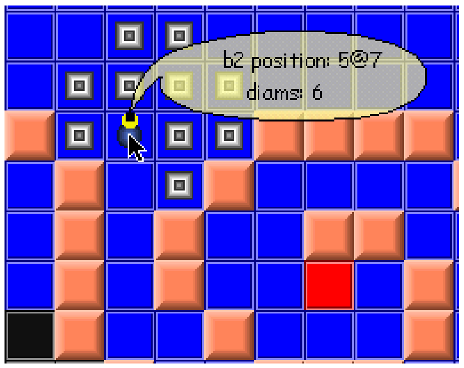


Figure 1.6: Displaying some information relative to a bot.

As we mentioned it, a robot should not bumped into a wall, else an error occusrs as shown by the Figure 1.7. A correct program should never such a kind of errors. Similarly, when a robot picks a diamond or drop one, it should check if it can do it. The following chapter explain how this can be done using conditional. When such an error occurs just close the window by clicking on the cross at the left top corner of the window.

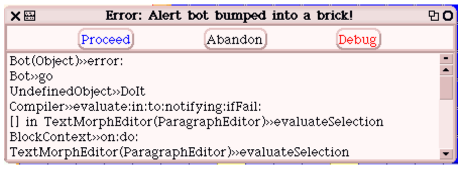
Teacher's Corner

It can be annoying to let the robots making noise while moving. You can indicate that all the robots have to move silently by executing the expression `Bot silent`. Read the Chapter 4 to have more information.

Teacher's Corner

A robot has several sensors. The following messages to check their status:

- `canMoveForward`, `canMoveLeft`, and `canMoveRight` return whether the bot can move forward, left, or right.



```
Bot(Object)»error:
Bot»go
UndefinedObject»DoIt
Compiler»evaluate:in:to:notifying:ifFail:
[] in TextMorphEditor(ParagraphEditor)»evaluateSelection
BlockContext»on:do:
TextMorphEditor(ParagraphEditor)»evaluateSelection
```

Figure 1.7: When a robot bumped into a wall.

- `isOnBlack`, `isOnMagenta`, `isOnGreen`, `isOnYellow`, and `isOnRed` return whether the bot is on a colored tile of the given color.
- `canPick` returns whether the robot is above a diamond and `canDrop` returns whether the the robot can drop a diamond.
- `isAtHome` returns whether the bot at the starting place.

The Figure 1.6 shows that you can get some information on a bot by letting the mouse over it to get a balloon.
A robot can pick or drop diamonds executing the following methods:

- `pick` picks up a diamond. Again you will have to pay attention using the method `canPick` that before picking a diamond that there is effectively one. The method `pick` raises an error when there is no diamond.
- `drop` drops a diamond on the current location if the robot has still one diamond, else it raises an error. The method `canDrop` indicates wether you can drop a diamond.
- `diamNumber` returns the number of diamonds that a robot is carrying. Note that you can also load a robot with a number of diamonds using the method `loadWith: anInteger`.

Note that a robot can walk on tiles containing diamonds without problems.

Finally a robot can paint tiles. Ask a robot to paint a tile of a given color using the messages `paintGreen`, `paintBlue`, `paintYellow`, `paintMagenta`, and `paintBlack`. Note that the starting place and bricks cannot be painted.

Teacher's Corner

The design of the robot behavior forces robot programmers to make tests for picking and dropping a diamond or before moving forward. We designed it especially to have this behavior. However, if you do not want to have the students testing for example before dropping a diamond. As explained in the chapter 4 you just have to define the method `safeDrop` defined in the class `Bot` as follows:

Method 1.1

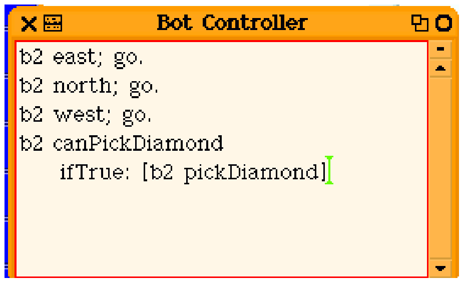
```
Bot>>safeDrop

   self canDrop
      ifTrue: [self drop]
```

Teacher's Corner

4 First Scripts and Methods

Using a Bot Controller we can send messages to a bot using its name. There is no need to declare a variable to refer to the bot and to initialize it as we were used to do with the turtle. This is because a Bot Controller is a special tool that automatically declares as variables all the robots defined in a given playing area.



```
b2 east; go.
b2 north; go.
b2 west; go.
b2 canPickDiamond
    ifTrue: [b2 pickDiamond]
```

Figure 1.8: Steering a robot by sending it messages.

Select the area named `area02` to have more space, create a robot, name it `b2`, and try the following script (script 1.2):

Script 1.2 (*A Simple Script)*

```
b2 west.
b2 go
```

Note that the script 1.2 can be expressed using cascade `;` to avoid to repeat unnecessary the receiver as shown by the script 1.3.

Script 1.3 (*A Simple Script using a Cascade)*

```
b2 west ; go
```

Hints... A Bot Controller allows you to execute only one single line if you position the cursor on it and select the menu item **do it** or command d. If you want to execute a script composed of multiple lines select them first.

For example the Figure 1.8 defines some scripts to steer the bot `b2`. Note that the variable `b2` in the script refers to the robot we created and named `b2`.

Some Steering Exercises

Here is a list of exercises you can try to get used to bot programming. Pick the area named `area2` to have more place to play and define the following scripts.

- Define a script that makes a robot walk 5 tiles in its current direction without taking care of bricks.
- Define a script that makes a robot turn east and walk 5 tiles in its current direction.
- Define a script that makes a robot walk a square of 6 tiles.

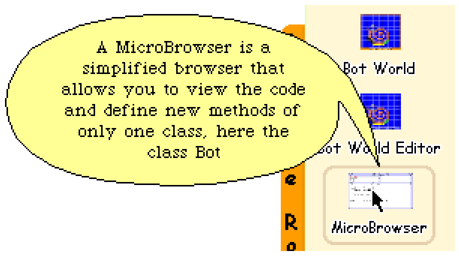
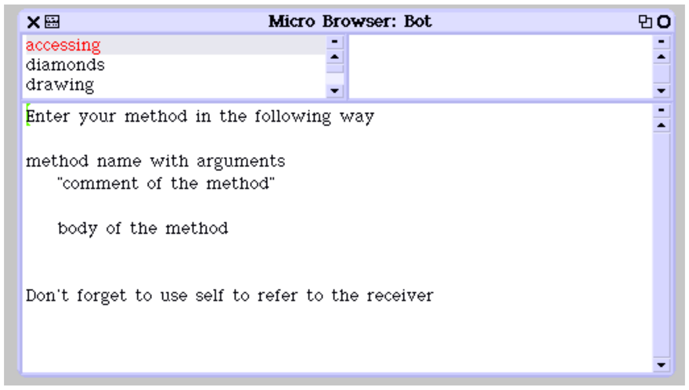

A MicroBrowser is a simplified browser that allows you to view the code and define new methods of only one class, here the class Bot
Bot World
Bot World Editor
MicroBrowser

Figure 1.9: Drag and drop the thumbnail of the micro browser to open it.

Defining Methods

In a similar fashion that you defined methods for the turtles (see Chapter **??**, you can define new methods for the robots using a dedicated code browser. To open such a code browser, drag its thumbnail from the orange flaps as shown in Figure 1.9 or execute the following expression `MicroBrowser browseBot`. You should obtain a micro browser as shown by the Figure 1.10. Note that the window title by containing the word 'Bot' indicates that you will be defining methods for the robots.



```
Enter your method in the following way

method name with arguments
    "comment of the method"

    body of the method


Don't forget to use self to refer to the receiver
```

Figure 1.10: A micro browser to define bot behavior.

You can create a category named for example `simple methods` for your methods. Define the method `fiveSteps` that is defined as follows:

Method 1.2

```
In category simple methods
fiveSteps

   5 timesRepeat: [self go]
```

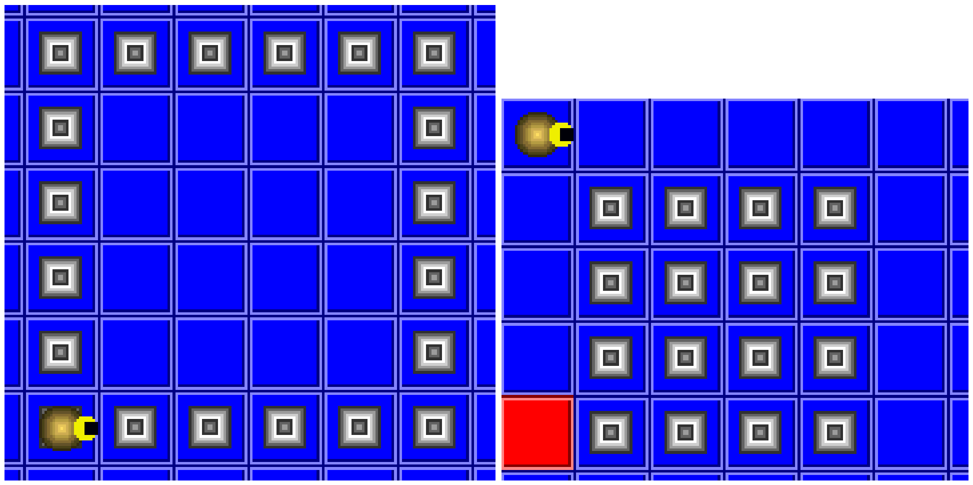


Figure 1.11: Two shapes.

With the assumptions that there is no brick, that the robot is in the middle of the world, and that it has enough diamonds to drop on the floor (Use the method `loadDiams:` to charge it with enough diamonds), define the following:

- Define a script that makes the bot draw the left square of diamonds shown in Figure 1.11.
- Define a script that makes the bot draw the right square of diamonds shown in Figure 1.11.
- Define a script that makes a triangle of diamonds as shown by the Figure 1.12.

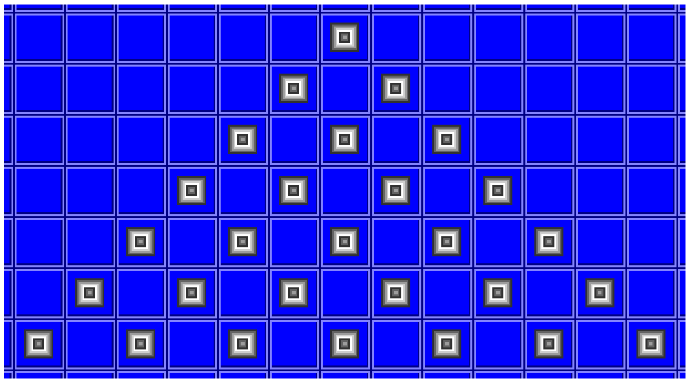


Figure 1.12: A triangle of diamonds.

5 Creating and Editing Bot's Areas

The bot environment allows you to define your own areas using a dedicated editor. To get the editor drag and drop the thumbnail named **Bot World Editor** from the orange flap or execute the following expression `BotWorldBoardEditor newStandAlone openInWorld`.

Once executed this script opens a blue window having the bar of buttons shown in Figure 1.13.

The Bot World Editor has from the left to right the buttons.

- **Open Tile Pane.** To open a pane with all the tiles that can be placed on an area.
- **Empty.** To empty the current area.
- **Pick Area.** To select one of the areas already defined.
- **Area Named.** To select one of the areas using its name.
- **Save Area.** To save the current area.

To add a tile, click on the wished tile in the pane containing the tiles or on a tile already present in the area and drop the tile at the wished location. If you want to delete a tile just pressed shift while clicking on the tile.

Now we get ready to try further topics such as how to program our robots.

6 Summary

You should now be able to open an robot environment, create some robots, steer them using the Bot Controller, and define new methods. The basic set of operations that a robot can execute is: `go`,`turnLeft`, `turnRight`, `north`, `south`, `east`, `west`, `pick`, `drop`, `loadDiams: anInteger`, `diamNumber`, `paintBlue`, `paintYellow`, `paintGreen`, `paintBlack`, and `paintMagenta`.

In addition to this a robot has several sensors that we will use in the following chapter: `canMoveForward`, `canMoveLeft`, `canMoveRight` `isOnBlack`, `isOnGreen`, `isOnYellow`, `isAtHome`, `isOnMagenta`, `canPick`, and `isAtHome`.

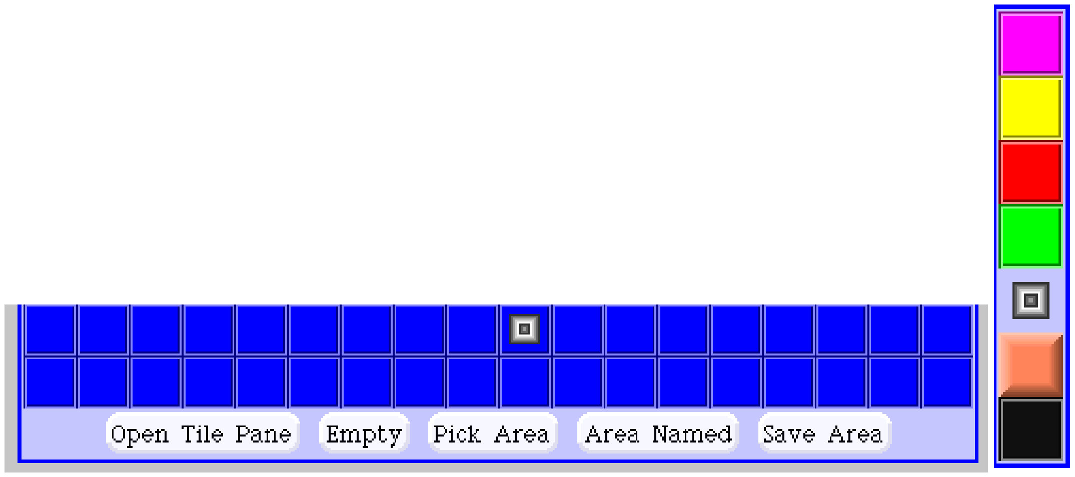


Figure 1.13: The actions proposed by the Bot World Editor and the tile pane.

2

Learning Conditions with Bot

Up until now all the programs we defined were executing *all* the messages they contained one after the other. There was no way to describe that certain messages have to only be executed when certain conditions were true. In this chapter and the following one we will introduce an important programming concept: the notion of conditional execution, *i.e.*, the fact a certain piece of code is executed under a given condition.

We start by defining a simple example that shows the need of conditional execution, in short conditional, then we present in detail the conditional expression offered by Squeak.

1 A Simple Example

For this exercise we suggest you to pick the second area as shown in the Figure 2.1. This will help you to understand and compare the results of your script with the one we describe.

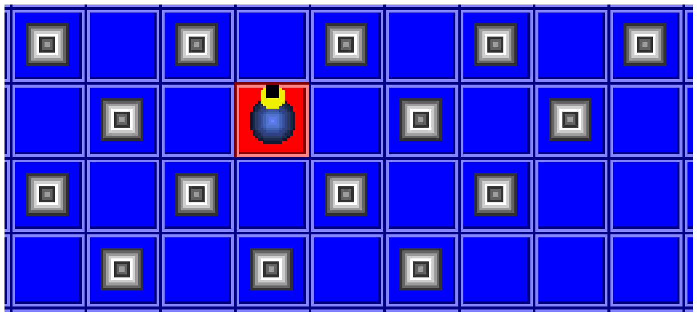


Figure 2.1: Part of the area named `area02`

A Small Problem. We would like that while a robot is moving it picks a diamond *if* possible, *i.e.*, if there is a diamond on the tile where the robot stands. This problem requires a *conditional* execution, *i.e.*,when there is a diamond the robot should pick it, when there is no diamond it should continue its way and do not try to pick the diamond.

The script 2.1 shows a solution and the Figure 2.2 shows the effect of executing this script 3 times in a row.

Script 2.1 (*Picking diamonds)*

```
b2 east.
b2 canPick
   ifTrue: [b2 pick].
b2 go
```

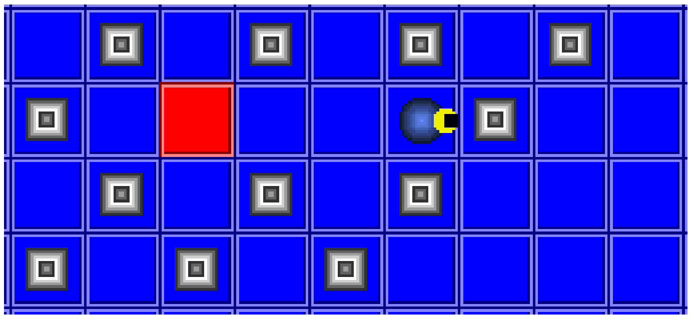


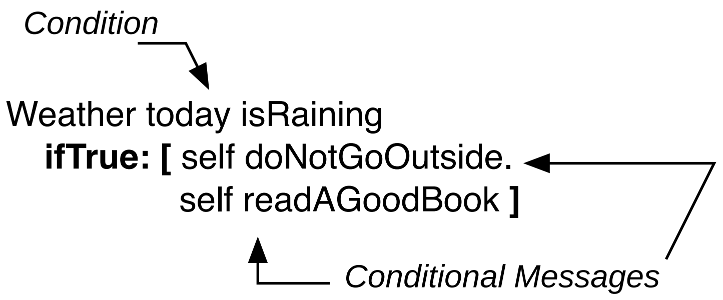
Figure 2.2: Result of applying 3 times the script 2.1

Let us analyze now what happened.

1. We asked the robot to face the east.
2. Then with the expression `b2 canPick ifTrue: [p2 pick]` we asked the robot to check whether it could pick a diamond and *if* this was the case to pick the diamond. This expression is a conditional expression.

   A conditional expression is composed of two parts: a *condition* and *conditional messages* as shown by the Figure 2.3. The expression `b2 canPick` is a condition and the expression `[p2 pick]` is a *conditional message* which gets only executed when the condition is true. The message `ifTrue:` defines the meaning of the condition, it says that the conditional messages are only executed when the condition is true.
3. Finally the expression `b2 go` is executed.

During the first two executions of this script, there was no diamond on the starting place (as shown by the Figure 2.1) so the expression `b2 canPick` was false and therefore the conditional expression `b2 pick` was *not* executed (because it is only executed when the condition is true as the `ifTrue:` message indicates it). During the third execution of the script, the bot was on a tile with a diamond, so the condition `b2 canPick` was true and therefore the conditional expression `b2 pick` was executed.



```
Weather today isRaining
   ifTrue: [ self doNotGoOutside.
             self readAGoodBook ]
```

Figure 2.3: A conditional expression composed of a condition and conditional messages.

What you see is that there are different kinds of expressions: some that are always executed while others are executed only when their associate condition hold. Note that a conditional expression is not limited to one single message but can be an extremely complex sequence of messages. Similarly the condition can be a complex expression as we will present it in the Chapter 7.

If you want to know how many diamonds a robot is carrying just ask it using the expression `b2 diamNumber` and print the result. After executing four times the previous script starting from the robot home it should carry one diamond. Modify the script 2.1 as shown in script 2.2 and open a transcript (first thumbnail of the yellow flap named Advanced) to follow its execution (note that the message `,` concatenate two strings into one).

Script 2.2 (*Picking diamonds)*

```
b2 east.
b2 canPick
   ifTrue: [b2 pick.
            Transcript show: 'Got a new diamond, now I''m carrying ',
                b2 diamNumber printString, ' diamonds'; cr].
b2 go
```

1.1 `ifTrue:` and `ifFalse:`

Squeak offers the methods `ifTrue:` and `ifFalse:` to express conditional expressions. Contrary to `ifTrue:`, the method `ifFalse:` executes its conditional messages when its condition is false. We can always use an `ifFalse:` method instead of a `ifTrue:` method by *negating* the condition. The script 2.3 is equivalent to the script 2.1 because we negated using the method `not` the condition.

Script 2.3 (*Picking diamonds using `ifFalse:`)*

```
b2 east.
b2 canPick not
   ifFalse: [b2 pick].
b2 go
```

The methods `ifFalse:` and `ifFalse:` follow the template shown below. Both execute a condition and depending on the value returned by the condition execute or not the conditional messages.

Important Messages 2.1

```
aCondition
   ifTrue: [ messagesIfConditionIsTrue ]

aCondition
   ifFalse: [ messagesIfConditionIsFalse ]
```

Teacher's Corner

When a message is executed the receiver of the message and the arguments are *always* evaluated. There is no exception to this rule. However, for conditional statements or loops, it is necessary to be able to control when a sequence of messages will be executed, if any. This is for this purpose that blocks are used delimited by `[` and `]`. A block is evaluated, but its semantics is to delay the execution of the messages it encloses. This is the reason why the methods such as `timesRepeat:` and `ifTrue:` require blocks as arguments.

Teacher's Corner

2 The Need for `ifTrue:ifFalse:`

Now imagine that we want a robot to turn north when it cannot pick a diamond. The idea is that the robot should walk in zigzag going to the north each time it cannot pick a diamond. The result of such a script is illustrated by the Figure 2.4.

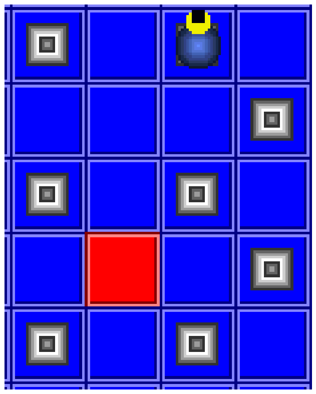


Figure 2.4: Result of applying 3 times the script 2.5 or the script 2.6.

We could write a script as the one shown in script 2.4, however this script does not do what we want as shown by the Figure 2.5. Read the script, execute it step by step slowly, and try to understand why the robot is only going straight to the north and not in zigzag as it should do.

Script 2.4 (*Picking diamonds or walking (Wrong Solution))*

```
b2 east.
b2 canPick
   ifTrue: [b2 pick].
b2 canPick
   ifFalse: [b2 north].
b2 go
```

The idea was to use the `ifTrue:` and `ifFalse:` methods on the same condition thinking that when one will be executed the other won't. But in fact when the first conditional expression is picking a diamond, it is modifying the context of the execution and when the second test is executed it is not the same context as when the first one was executed. Indeed the diamond is not there anymore! In fact when the robot can pick a diamond it does it then when the second condition `b2 canPick` is executed the condition is false. And it will *always* be false, so the robot will always walk in direction of the north.

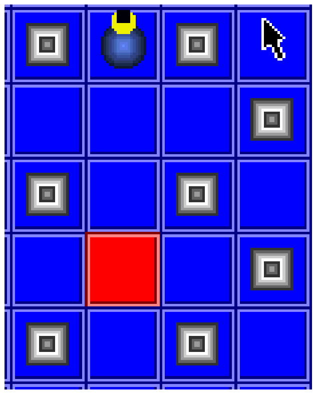


Figure 2.5: Result of applying 3 times the script 2.4.

What we need is to avoid to execute twice the method `canPick` and keep the result of the method for the second conditional expression. We should use the value of the condition when it was *first* executed. This is easy using a variable (see Chapter **??**. For example, the script 2.5 introduces a variable named `resOfCanPick` and the conditional expressions use this variables instead of invoking the message `canPick` two times.

Script 2.5 (*Picking diamonds or walking (First Correct Solution)*)

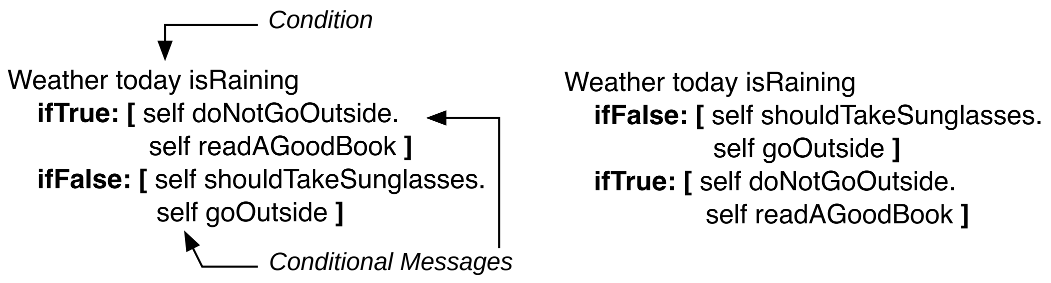
```
| resOfCanPick |
b2 east.
resOfCanPick := b2 canPick.
resOfCanPick
   ifTrue: [b2 pick].
resOfCanPick
   ifFalse: [b2 north].
b2 go
```

The solution is working but it is not really elegant to have to add an extra variable. Imagine if we would have a lot of conditional expression, we would end up having a lot of variables for one single conditional. Squeak offers the method `ifTrue:ifFalse:` to solve this problem. The script 2.6 presents the final solution. What we see is that we do not need to use an extra variable and that we do not need to have two conditions. Note that the method `ifTrue:ifFalse:` is a *single* method with two arguments, one for the true case and one for the false case, in a similar way that `color:withSize:`. Therefore you should not put a period after the `]` following the `ifTrue:`.

Script 2.6 (*Picking diamonds or walking (Correct Solution using `ifTrue:ifFalse:`)*)

```
b2 east.
b2 canPick
   ifTrue: [b2 pick]
   ifFalse: [b2 north].
b2 go
```

The method `ifTrue:ifFalse:` executes one condition, here `b2 canPick`, and depending of its value executes the messages corresponding to the true case or to the false case as explained by the Figure 2.6 and the following template. We say that the method `ifTrue:ifFalse:` has two branches corresponding to two different possible executions. Here the branches are limited to a message send (`b2 pick` for the true branch and `b2 north` for the false branch) but a branch can contain a complex sequence of messages as we will see later.



```
Weather today isRaining
   ifTrue: [ self doNotGoOutside.
             self readAGoodBook ]
   ifFalse: [ self shouldTakeSunglasses.
              self goOutside ]
```

```
Weather today isRaining
   ifFalse: [ self shouldTakeSunglasses.
              self goOutside ]
   ifTrue: [ self doNotGoOutside.
             self readAGoodBook ]
```

Figure 2.6: Conditional with two branches: one for true and one for false

Note that the method `ifFalse:ifTrue:` also exists and that it works the same way the method `ifTrue:ifFalse:`, *i.e.*, it will execute the false case when the condition is false and the true one when the condition is true, exactly as the method `ifTrue:ifFalse:`. This method is just there to help you writing more readable code if you want to start reading the messages executed when the condition is false as shown by the Figure 2.6.

Important Messages 2.2

```
aCondition
   ifTrue: [ messagesIfConditionIsTrue]
   ifFalse: [ messagesIfConditionIsFalse ]


aCondition
   ifFalse: [ messagesIfConditionIsFalse ]
   ifTrue: [ messagesIfConditionIsTrue ]
```

Now you are ready to solve a lot of problems based on conditional. The final variation of the small problem with work on will show you that conditions can be nested.

3 Nested Conditions

A conditional expression can contain any other messages and in particular other conditional expressions. This is what we will present now. There is nothing spectacular but it is common that's why we want to show it to you.

A Small Problem. We would like that a robot picks a diamond *if there is one or* drops a diamond *if there is no diamond and if it has enough diamond to drop*.

We load the robot using the script 2.7 because we can be in a situation where the robot has to drop diamonds without been able to pick some and it should still be able to do it.

Script 2.7 (*Initializing the robot)*

```
b2 east.
b2 loadWith: 10.
```

Then to solve our problem we write the script 2.8. If you execute four times in a row this script you should obtain the situation shown by the Figure 2.7.

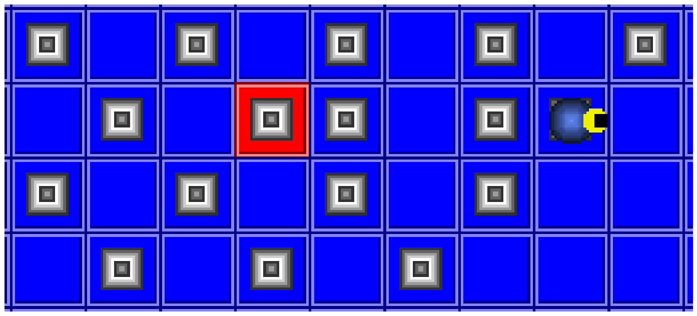


Figure 2.7: Situation once the script pick or drop has been executed four times.

Script 2.8 (*Pick or drop)*

```
b2 canPick
   ifTrue: [b2 pick]
   ifFalse: [b2 canDrop
                ifTrue: [b2 drop].
             Smalltalk beep]
b2 go.
```

As you see this script has the same general structure than the script 2.6. Let us analyze what can happen and how the script works by evaluating different scenario.

1. First we start to test whether the robot can pick a diamond.

2. If this is possible, the robot does it and the conditional expression is over, it then walks on the next tile.

3. If it was not able to pick a diamond then the messages composing the false case are executed. These messages contains a sequence of message: the first one `b2 canDrop ifTrue:  [b2 drop]` is again a conditional expression and the second one a message to make a little noise (We added the expression `Smalltalk beep` to show you that you can have multiple condition messages).

   The expression `b2 canDrop` is executed

   - When it is true the conditional message `[b2 drop]` is executed.
   - When the robot can not drop diamond, the conditional expression is over.

   The second message `Smalltalk beep` is executed independently that the robot can or not drop a diamond.

This example shows that we can nest conditional expressions inside other conditional expressions. It allows one to express more complex situations. Note that when a conditional branch is executed all the expressions that compose it are executed. For example the execution of the message `Smalltalk beep` does not depend on the fact that the robot can or not drop a diamond. The beep will be always produced when the robot cannot pick a diamond because it is contained in a sequence of messages that depend on the `canPick` condition.

4 About Method Returned Values

An important point that we want to stress now is that when we have a conditional expression such as the one shown in script 2.1, the conditional expression `b2 canPick` should *return* a boolean *i.e.*,`true` or `false`. This means that we are not only interested by the execution of the message *but also* by the result it computes and returns. In Squeak per default a method always returns the *receiver* of the message, `self`. If we want that the method returns another object, we have to explicitly mention it using the caret `^` construct followed by the value to return. The method `example1` shown in 2.1 returns the number 1, the method `example2` returns the value of the expression `1 + 2` after the `^` , so it returns 3.

Method 2.1

```
example1
   "returns 1"

    ^ 1

example2
   "returns the result of the expression 1+2"

    ^ 1 + 2
```

In general methods are not limited to return only booleans. They can return all kind of objects. Look at what we already saw without paying attention. A method can return

- *numbers*. The method + in `10 + 2` returns the sum of the two numbers, `10 max: 20` returns the maximum, `20 atRandom` returns a number between 1 and 20, `caro direction` is returning the current direction to which a turtle is pointing at.
- *booleans* as we just saw.
- *colors* as we were used to do with the turtle. For example the expression `Color blue` is asking the class `Color` to create a new color blue.
- *points*. For example, `b2 botWorldPosition` returns the position of a robot in its environment.
- *string of characters*. For example `b2 name` returns the name of a robot.
- *turtles*. In the expression `Turtle new` the class `Turtle` returns a newly created instance.

The chapter 7 explains in detail the concept of boolean and boolean expressions. Here we want to stress the point that the *condition* of a conditional expression should return a boolean value *i.e.*,`true` or `false`. This boolean value can be computed based on other result. For the example, in the following script 2.9 the expression `b2 diamNumber` returns a number, but the expression `b2 diamNumber = 3` returns a boolean, so this is correct. The conditional message, `Smalltalk beep` will only be executed when the robot named `b2` carries 3 diamonds.

Script 2.9 (*A conditional expression comparing numbers.)*

```
b2 diamNumber = 3
  ifTrue: [Smalltalk beep].
```

5 Final Experiments

- Pay attention to the slightly different problem. Imagine that we want the bot to pick a diamond if there is one *and* dropping one if it can.
- It is quite annoying to get an error when a bot bumps into a wall or in the limits of its world. To fix this problem, we can define the method `sureGo` that checks first whether the move is possible. Define such a method (note that the problem is that we will never know if the robot is indeed bumping into a wall when building more complex behavior.
- In a similar fashion, define the method `safePick` that checks that there is a diamond before picking one.
- Define a method `goBackIfBrick` that makes the robot turning in the opposite direction when it cannot walk anymore.

6 Summary

Method	Description

`aCondition`
`   ifTrue: [messagesIfConditionIsTrue ]`
	Execute messagesIfConditionIsTrue only if aCondition is true. The robot will only pick the diamond if he can.
	
	`b2 canPick`
	`   ifTrue: [ b2 pick]`

`aCondition`
`   ifFalse: [messagesIfConditionIsFalse ]`
	Execute messagesIfConditionIsFalse only if aCondition is false. The system beeps only when the robot cannot pick a diamond
	
	`b2 canPick`
	`   ifFalse: [Smalltalk beep]`

`aCondition`
`   ifTrue: [messagesIfConditionIsTrue ]`
`   ifFalse: [messagesIfConditionIsFalse ]`
	Execute messagesIfConditionIsTrue whether aCondition is true otherwise execute messagesIfConditionIsFalse. The robot pick a diamond when it can otherwise the system beeps.
	
	`b2 canPick`
	`   ifTrue: [b2 pick]`
	`   ifFalse: [Smalltalk beep]`

`aCondition`
`   ifFalse: [messagesIfConditionIsFalse ]`
`   ifTrue: [messagesIfConditionIsTrue ]`
	Execute messagesIfConditionIsTrue whether aCondition is true otherwise execute messagesIfConditionIsTrue. The robot pick a diamond when it can otherwise the system beeps.
	
	`b2 canPick`
	`   ifFalse: [Smalltalk beep]`
	`   ifTrue: [b2 pick]`

3

Conditional Loops

Having conditions is a crucial tool for expressing complex programs. However, conditions are not enough. Sometimes we would like to combine loops and conditions[1]. In fact we would like *conditional loops*, *i.e.*,loops that repeat sequence of messages until a certain condition holds.

We will use heavily conditional loops to simulate animal behavior and other strategies such as escaping mazes or following paths.

1 Conditional Loops

The idea behind conditional loops is that a sequence of messages is repeated while a certain condition holds. Squeak defines two messages `whileTrue:` and `whileFalse:` that allows one to define conditional loops as shown by the templates below. The Figure 6.1 shows that a conditional loops is composed of a *condition* and *conditional messages*.

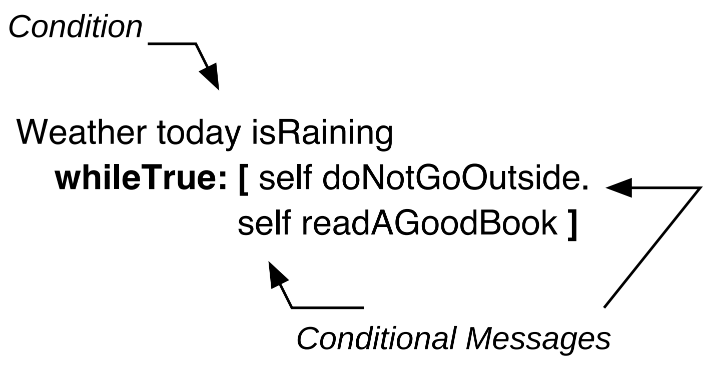


Figure 3.1: The `whileTrue:` conditional loops is composed of a condition and a sequence of conditional messages.

Important Messages 3.1

```
[ condition ] whileFalse:
  [ conditional messages ]

[ condition ] whileTrue:
  [ conditional messages ]
```

An example. Let us take a simple example to illustrate their use. Imagine that we want that a robot takes all the diamonds available on a line. Let us call this method `pickCompleteLine`.

[1]In this book we only superficially present recursion, *i.e.*,the fact that we can define method in terms of themselves.

To simplify the example, we first define the method `safePick` method 3.1 that makes a robot pick a diamond if there is one and do not raise error otherwise.

Method 3.1

```
safePick
   "pick a diamond if there is one"

   self canPick
      ifTrue: [self pick]
```

Now we can define the method `pickCompleteLine` as follows:

Method 3.2

```
pickCompleteLine

   [self canMoveForward]
      whileTrue: [self safePick.
                  self go]
```

How does it work?

1. First a condition is evaluated. Here the expression `self canMoveForward` is executed.
2. ◦ If the condition is false then the conditional loop stops and nothing more is executed in this method.
   ◦ If the condition is true then the conditional messages are executed. Here, `self safePick. self go` are executed. Then the process restarts as described in the point 1. This is why we have a conditional loop.

Note that in a similar way that with conditional expression, the condition in a conditional loops should return a boolean (See **??**).

As for `ifTrue:` and `ifFalse:`, `whileTrue:` can be substitued by `whileFalse:` by negating the condition. For example, the method `pickCompleteLine` can easily be defined using the method `whileFalse:` by negating the condition as follow:

Method 3.3

```
pickCompleteLine

   [self canMoveForward not]
      whileFalse: [self safePick.
                   self go]
```

As you see the new version is less understandable. The advice is that you should use the method that helps you to understand the algorithm you are defining.

2 About the use of `[]`

You may have difficulties to remember when to put `[]` and not. There are basically two rules in Squeak. You surround an expression with `[` and `]` when:

- you need to execute several times the same expression. For example,
  - `4 timesRepeat: [caro go: 10; turnLeft:90]` repeats 4 times the messages `caro go: 10; turnLeft:90`,

    - `1 to: 10 do: [:i | Transcript show: i printString ; cr]` repeats ten times the `[:i | Transcript show: i printString ; cr]` which prints the number to the transcript.

- the expression is not always executed. For example,

    - `b2 canPick ifTrue: [b2 pick]` only executes `b2 pick` under certain circumstances,
    - `[self canMoveForward] whileTrue: [self safePick; go]` repeats multiple times conditionally both `self canMoveForward` and `self safePick; go`, therefore the receiver and the argument are blocks.

3 Learning from Errors

You may note that the method `pickCompleteLine` method 3.3 was not really good. Can you understand what is the problem? What's happen for the diamonds that are just in front of a brick? A robot will not pick them.

Indeed using this method a robot does not pick a diamond that is just near a brick or near the limit of the world. First explain why by showing step by step why this situation occurs. Second propose a solution to this problem.

The solution we thought about is shown by the method 3.4. The idea is that a robot should pick a diamond then try to move forward.

Method 3.4

```
pickCompleteLine

   [self safePick.
   self canMoveForward]
      whileTrue: [self go]
```

This example shows that the condition can be composed by a sequence of expressions. As for all the conditional expressions there is one important constraint that the last expression should return a boolean. What we mentioned in the section 4 applies also for conditional loops too.

Stopping an endless loop. As you may guess it may happens that the loop does terminate.

In fact it is not exceptional to write endless loop, *i.e.*, the sequence of messages is continously executed. This happens because the boolean expression never returns true for `whileFalse:` and false for `whileTrue:`. You can stop a loop by pressing Apple-. on Mac or Alt or Control-C on other platform. Then to understand why the loop does terminate you can use the debugger that pops up and by clicking its Debug button.

What you should understand is that there should be something in the sequences of expressions that leads to change the value of the condition. For example, the message `go` in the method 3.3 linked to the fact that our environment is bounded makes sure that the robot will arrive at one point in time in a situation where the condition `self canMoveForward` will not be true anymore, hence the loop will stop. As you see this is not simple but you should always ask yourself the question whether your loops expressions contain at least expressions that would change the condition value.

To help to understand what is going wrong with your loops you can use simple debugging techniques such as introducing a `self halt` expression or the conditional messages in the condition which opens a debugger. Another approach is to generate a trace in the Transcript. For example in the following method we write in the transcript the position of the robot. Note that the position are points and that we transformed them into strings because the transcript only knows to print strings.

Method 3.5

```
pickCompleteLine

   [self halt.
   self safePick.
   self canMoveForward]
      whileTrue:
       [Transcript show: self botWorldPosition printString ; cr
       self go]
```

4 Practising

Now we are ready to write some other simple conditional loops. A robot can drop multiple diamond on the same tile, define the following methods:

- `fullyDrop` that make a robot drops all the diamonds it is carrying on the current tile.
- `fullyPick` that make the robot picks all the diamonds on the current tile.
- `dropLineOfFive` that make the robot drop all its diamond per packets of five in a line.
- `fullPickLine` that makes the robot picking all the diamonds available on each tiles for a complete line.
- `reverseLine` that makes the robot picks a diamond if there is one or dropping one if there is none and this on a complete line.
- `straightUntilYellow` that makes the robot moving straight and stop if it is passing over a yellow tile.

Note that you may have to nest conditional and conditional loops.

5 Experiments: Yellow Finder

Now we would like to build different strategies to let a robot walk until it finds a yellow tile. First define an environment that looks like the one shown by the Figure 3.3 to have a bit more fun.

We need some methods to manage the direction of our robot for example, we can define the method `pointBack` which makes a robot pointing in the opposite direction, `pointLeftOrRight` which makes it points randomly on the left or the right.

Imagine how such methods could be implemented to train yourself.

Method 3.6

```
pointBack

   self direction: self direction negated
```

Method 3.7

```
pointLeftOrRight

   2 atRandom = 2
      ifTrue: [self turnLeft]
      ifFalse: [self turnRight]
```

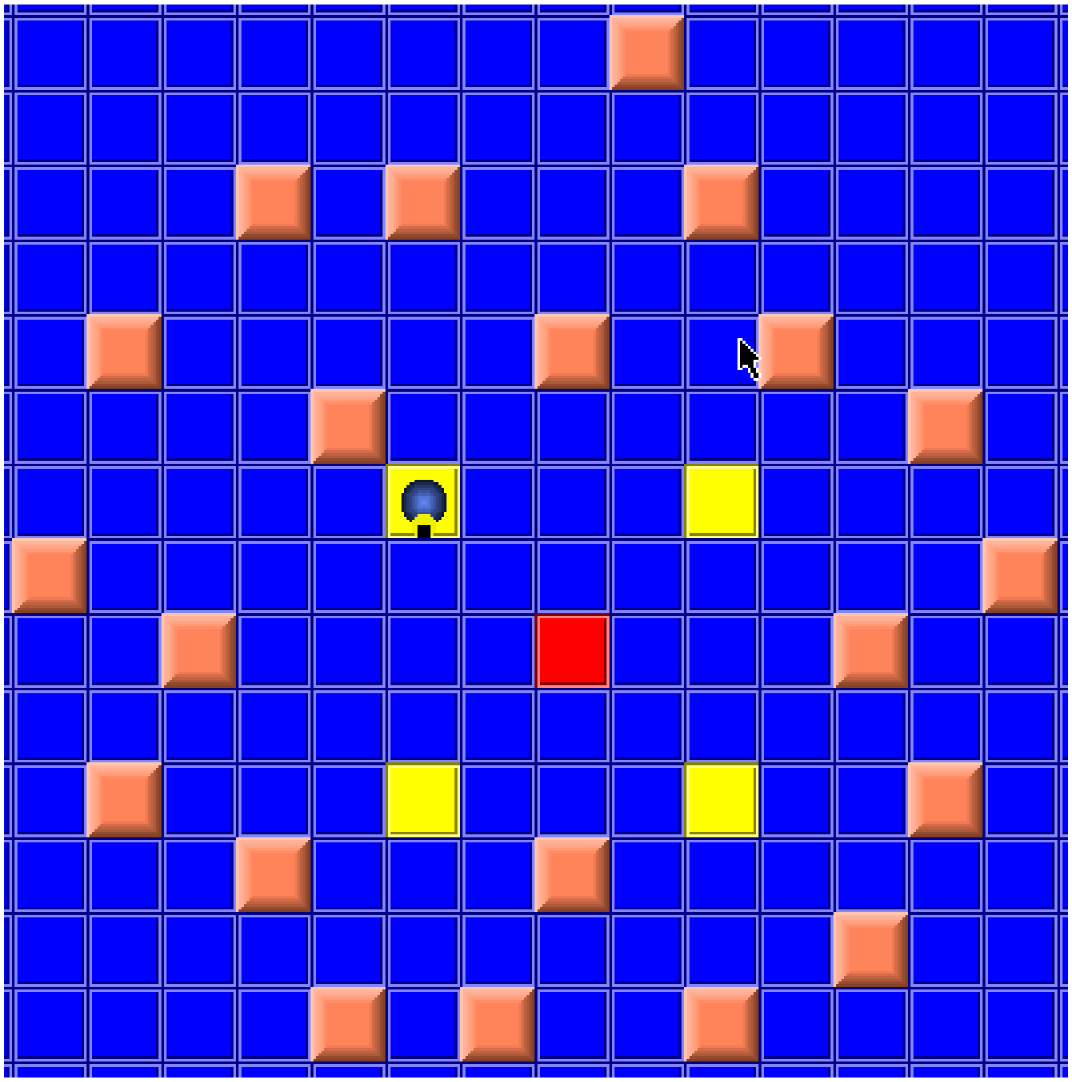


Figure 3.2: An environment for fooling around looking for yellow tiles.

The method `pointRandomly` can be implemented in different ways. We propose you to define the method `randomDirection` that will return a random direction @@To move in the distribution@@

Method 3.8

```
pointRandomly

self direction: self randomDirection

randomDirection
    "a direction is a point composed by -1,0, or 1 with the  constraint
that x * y should always be = to 0"

| x y |
x := 3 atRandom - 2.
y := 3 atRandom - 2.
^ (x * y) isZero
ifTrue: [x @ y]
ifFalse: [self randomDirection]
```

@@To move in the distribution@@

Bumping and Turning. A simple strategy is to make a robot walking until it is on top of a yellow and when the robot bumps into a wall make it turns randomly on the right or the left.

Method 3.9

```
untilBumping

   [self isOnYellow]
      whileFalse: [self canMoveForward
                      ifFalse: [self leftOrRightDirection].
                   self go]
```

Note that there is problem with the method `untilBumping`. Can you see it? What's happen if there is just a brick close to another one and that the robot choose to go exactly in that direction. Propose a solution.

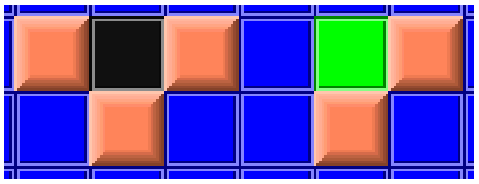


Figure 3.3: Brick configuration potentially blocking a robot using leftOrRightDirection strategy.

Fooling around. Another strategy is to let the robot randomly browse and check whether it is walking on a yellow tiles.

Method 3.10

```
browse

   [self isOnYellow]
      whileFalse:
          [self pointRandomly.
          self canMoveForward
            ifTrue: [self go]]
```

We suggest you to make the robot paint the tiles were it is. This way you will be able to see where it moved. You can imagine a solution where you use simply green (left in Figure 3.4). Then a fun variation is to use multiple colors that represent the number of times the bot passed on the same tile. For example in the right picture of the Figure 3.4 a robot painted first in green, then magenta when it was on a green tile and finally black when it was on a magenta tile.

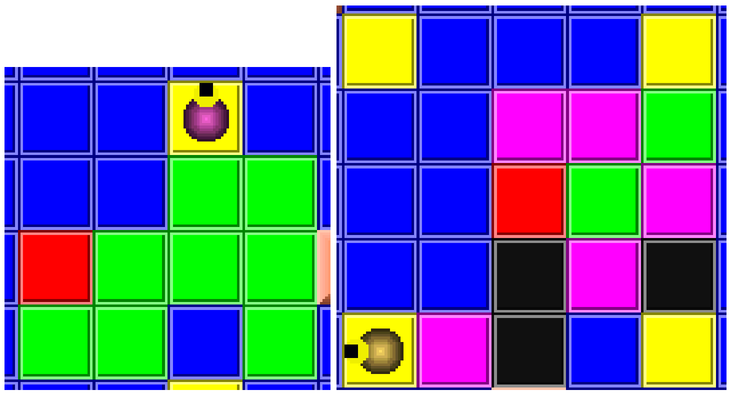


Figure 3.4: Left: A robot browsing and letting a simple trace. Left: A robot browsing and letting a complex trace.

6 Summary

Method	Description
`[aCondition] whileFalse: [SequenceOfMessages]`	Execute SequenceOfMessages only if *aCondition* is false.
`[ aCondition ] whileTrue: [SequenceOfMessages]`	Execute SequenceOfMessages only if *aCondition* is true.

4

Customizing the Bot

The implementation of the robot we propose can easily be changed to fit your particular needs as teacher or as robot designer. In this chapter we show how you can adapt the robot code to your own wishes.

1 Extending or Refining Robot Behavior

When we designed the robot interface we payed attention to have short and descriptive methods that we can type easily. However certain methods can still be shortened such as for example `canMoveForward` could become `canFwd`.

Method 4.1

```
In category extension
Bot>>canFwd
   "return true wether the receiver can move forward"

   ^ self canMoveForward
```

The method `safePick` method 3.1 that we presented in previous chapter is another one of the changes that can be useful. Indeed the design of the method `pick` forces the programmer to check whether there is a diamond available. This can be annoying after a while.

2 Controlling Robot Sounds

As a teacher you may want to control the sounds emitted by the robots. You can do that by sending messages directly to the `Bot` class itself as shown in the script 4.1.

Script 4.1 (*Controlling robot sounds)*

```
Bot silentMotor.
Bot noisyDrop.
Bot noisyPick.
Bot silentBump
```

The method `silent` disables all the sounds and the method `noisy` enables them all. If you want a finer granularity use the methods `noisyMotor`, `noisyDrop`, `noisyPick`, `noisyBump`, `silentMotor`,`silentDrop`, `silentPick`, and `silent`.

You can also redefine the method `Bot class»initialize` defined on the class side of the `Bot` class (you should use a plain Smalltalk browser and press the class Button to see its definition see Chapter**??**). The default definition of the method `Bot class»initialize` is shown in 4.2. You can change its last line.

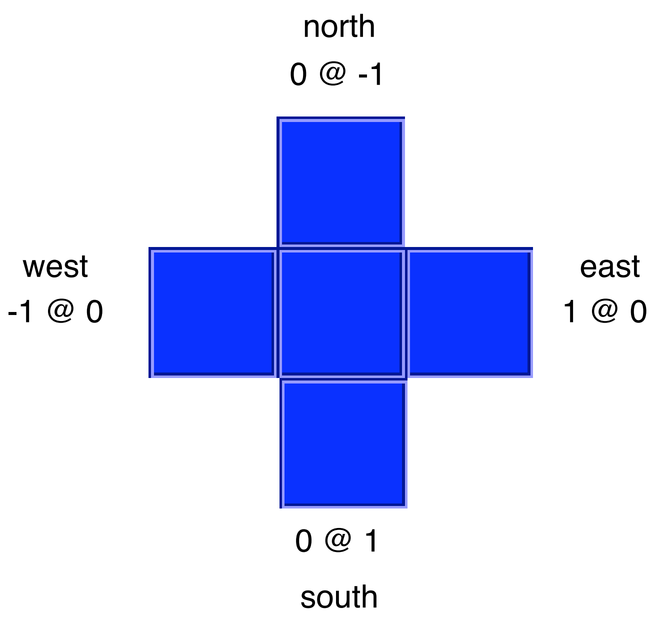


Figure 4.1: The encoding of directions: changing the direction to one of the direction encoding values changes the bot direction.

Method 4.2

```
In category initialization
initialize
"self initialize"

colorIndex := 0.
colorTable := Color
wheel: self maxColor
saturation: 0.6
brightness: 1.
self noisy
```

3 New Sensors

The robot sensors are quite limited as per default they only test the tile on which a robot is standing. Here are the definitions of new sensors that can check the color of a tile one tile ahead in one direction.

The direction is encoded by points having the following constraints: their values is -1, 0, or 1 and as a robot can only move in four directions the x and y cannot be both 1 or -1, said mathematically x * y should be zero.

For example here are the methods `westEncoding` and `northEncoding` that represent the absolute directions `west` and `north`.

Method 4.3

```
Bot>>westEncoding

   ^ -1 @ 0

Bot>>northEncoding

   ^ 0 @ -1
```

Then the method `west` just changes the direction of a robot using the method `direction:` with the correct direction ecoding as follow.

Method 4.4

```
Bot>>west

Delay forMilliseconds: 100.
self direction: self westEncoding
```

The method `leftDirectionEncoding` shows how the direction pointing to the left from the current direction can then be computed.

Method 4.5

```
Bot>>leftDirectionEncoding

   | dir |
   dir := self direction.
   ^ dir y @ dir x negated
```

This is easy to define new sensors because we just have to add direction to the bot position to have access to the tiles in those directions. The methods `isLeftTileYellow`, `isRightTileYellow`, and `isRightTileYellow` shown in 4.6 show how to define new sensors that we will use in the chapter**??**.

Method 4.6

```
In category advanced sensors
Bot>>isLeftTileYellow

   | leftTilePosition |
   leftTilePosition := self leftDirectionEncoding + self botWorldPosition.
   ^ (self isNextMoveInWorld: leftTilePosition)
      and: [(botWorld tileAt: leftTilePosition) isYellowTile]

Bot>>isRightTileYellow

   | rightTilePosition |
   rightTilePosition := self rightDirectionEncoding + self botWorldPosition.
   ^ (self isNextMoveInWorld: rightTilePosition)
      and: [(botWorld tileAt: rightTilePosition)
            isYellowTile]

Bot>>isRightTileYellow

   | tileInFront |
   tileInFront := self direction + self botWorldPosition.
   ^ (self isNextMoveInWorld: tileInFront)
      and: [(botWorld tileAt: tileInFront)
            isYellowTile]
```

4 Measuring Distance

Another interesting information you may need is the distance from which a robot is from its starting point. Let us call this method `homeDistance` and define it as follow. First we calculate the distance along the axis (here we use the fact that subtracting two points returns a point representing the difference for each coordinate) then based on that we compute the distance.

Method 4.7

```
Bot>>homeDistance
   "Return the distance from the starting place"

   | diff |
   diff := self botWorld startingPlace - self botWorldPosition.
   ^ (diff x squared + diff y squared) sqrt
```

5 Robot Variables

A robot can have its own set of variables. A robot can hold and remember all kinds of values using the methods `valueOf:` and `set:to:`. The method `valueOf: aVariableName` returns the value of the variable with the name `aVariableName`. The method `set: aVariableName to: aValue` changes the variable `aVariableName` to hold `aValue` as a normal variable. In a similar way we use variable to hold value, these methods allows one to give names to value stored in a robot and retrieve them using their names. In addition, the method `isVariableDefined: aVariableName` allows one to know whether a variable is defined.

The script script 4.2 defines for the robot named `b2` the variable `count` with the value `0`, then access this value.

Script 4.2 (*Using robot variable in a bot controler)*

```
b2 set: #count to: 0.
"Define the variable count and set it to 0"

b2 valueOf: #count
"Access the value of the variable count and return it
here 0 is returned"
```

The script script 4.3 defines for the robot named `b2` the variable `count` with the value `0` then add `1` to this variable. Then a variable `newCount` is defined with the value `3`. Finally the sum of the variable `newCount` and `count` is set to the `newCount` variable.

Script 4.3 (*Defining, adding bot variable)*

```
b2 set: #count to: 0.
"Define the variable count and initializes it with 0"

b2 set: #count to: (b2 valueOf: #count) + 1.
"Add one to the value of the variable count"

b2 valueOf: #count
"return 1"

b2 set: #newCount to: 3.
b2 set: #count to: (b2 valueOf: #count) + (b2 valueOf: #newCount).
"Add in count the previous count value and the one of newCount"
b2 valueOf: #count
"return 4"

b2 isVariableDefined: #count
"return true as the variable #count is defined"
```

Note that accessing the value of a variable that does not have been defined previously is an error as shown by the script script 4.4.

Script 4.4 (*Error: accessing an undefined variable)*

```
b2 valueOf: #lulu
"report an error stating that the variable lulu is not defined"
```

Using a robot variable we can now introduce the possibility to count the number of steps that a robot is making. The solution is based on the definition of the new methods `initializeSteps`, `stepPlusOne`, `steps`, and `countingGo`.

Method 4.8

```
In category distance
Bot>>initializeSteps
  "initialize to 0 the number of steps a robot did"

  self set: #steps to: 0

Bot>>stepPlusOne
  "increment the number of steps by one"

  self set: #steps to: (self valueOf: #steps) + 1

Bot>>steps
  "return the number of steps"

  ^ self valueOf: #steps
```

Method 4.9

```
In category distance
Bot>>countingGo
  "move forward the receiver and count this move"

  (self isVariableDefined: #steps)
     ifFalse: [self initializeSteps].
  self go.
  self stepPlusOne
```

Note that this approach has the drawback of only initializing the `steps` variable when the `countingGo` is invoked. So using the `steps` method before a call to `initializeSteps` or `countingGo` will result in an error mentioning that the variable is not defined. Propose a solution using the method `isVariableDefined:`.

Another solution is to redefine the methods `otherInitialize` and `go` directly to introduce the initialization and the counting.

The method `otherInitialize` is a method dedicated for robot extension initialization, per default it does nothing so you can freely add your extension initialization code there.

Method 4.10

```
In category initialize
Bot>>otherInitialize

   self initializeSteps
```

The method `go` is a bit more complex but you should not really pay attention to that and only add at the end of the method a line to count that one extra step has been made.

Method 4.11

```
In category operations
Bot>>go
   "Move the bot one tile in the current direction without
   checking wether it is possible. Raise an error if there is a
   problem "

   | nextPosition |
   nextPosition := self nextPosition.
   (self isNextMoveInWorld: nextPosition)
      ifFalse: [self scratch.
         ^ self error: ’Alert bot out of limits!’].
   self isWallInFront
      ifTrue: [self scratch.
         self error: ’Alert bot bumped into a brick!’]
      ifFalse: [self goTo: nextPosition.
         World doOneCycle.
         self motor].
    self stepPlusOne
```

5 Conditions with Caro

Up until now the programs we defined were executing *all* the messages they contained. There was no way to describe that certain messages have to only be executed when certain conditions were true. In this chapter and the following one we will introduce an important programming concept: the notion of conditional execution, *i.e.*,the fact a certain piece of code is executed under a given condition.

We start by defining a simple problem that shows the need of conditional execution, then we present in detail the conditional expression offered by Squeak.

1 A Simple Problem

We would like to change the color of a turtle depending on its distance from the center of the screen. Let us say that if a turtle is located at a distance smaller than 200 pixels from the center it should be red else it should be green.

This problem requires a *conditional* execution, depending on the turtle location its color will be different[1]. The script 5.1 presents a possible scenario showing how the method `distanceDetector` is used.

Script 5.1 (*A Simple Detector)*

```
caro := Turtle new.
caro jump: 20.
caro distanceDetector.
caro jump: 20.
caro distanceDetector.
```

A possible definition of the method `distanceDetector` is shown in the method 5.1.

Method 5.1

```
distanceDetector

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

Let us analyze now what happened.

1. The distance from the receiver to the center of the screen is computed and stored into the variable `dist`.
2. Then with the expression `dist < 200 ifTrue: [self color: Color red] ifFalse: [self color: Color green]` *if* the distance is smaller than 200 the color of the receiver is changed to red else it is changed to green. This expression is a *conditional* expression.

[1]We could have defined the color as a function in terms of the distance but this is not our purpose here. Moreover, it may be rare to be able to define a continuous function so we may need also condition to define such a method.

A conditional expression is composed of two parts: a *condition* and *conditional messages* as shown by the Figure 5.1. The expression `dist < 200` is a condition and the expression `[self color: Color red]` is a *conditional message* which gets only executed when the condition is true. `ifTrue:` defines the meaning of the condition, it says that the conditional messages are only executed when the condition is true. Similarly the expression `[self color: Color green]` is a conditional message that is only executed when the condition is false.

What you see is that there are different kinds of expressions: some that are always executed while others are executed only when their associate condition hold. Note that a conditional expression is not limited to one single message but can be an extremely complex sequence of messages. Similarly the condition can be a complex expression as we will present it in the Chapter 7.

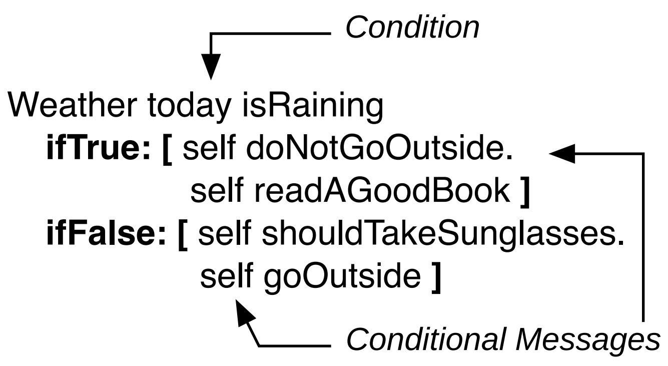


Figure 5.1: A conditional expression composed of a condition and conditional messages.

Teacher's Corner

When a message is executed the receiver of the message and the arguments are *always* evaluated. There is no exception to this rule. However, for conditional statements or loops, it is necessary to be able to control when a sequence of messages will be executed, if any. This is for this purpose that blocks are used. A block is evaluated delimited by `[` and `]`, but its semantics is to delay the execution of the messages it encloses. This is the reason why the methods such as `timesRepeat:` and `ifTrue:ifFalse:` require blocks as arguments.

Teacher's Corner

Experiments. You can use the Transcript (first thumbnail of the yellow flap named Advanced) and a trace to see how the expressions are executed as shown in the method 5.2.

Method 5.2

```
distanceDetector

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

The method `ifTrue:ifFalse:` executes one condition, here `dist < 200`, and depending of its value executes the messages corresponding to the true case or to the false case as explained by the Figure 5.2 and the following template. We say that the method `ifTrue:ifFalse:` has two branches corresponding to two different possible executions. A branch can contain a complex sequence of messages as we will see later. Note that the method `ifTrue:ifFalse:` is a *single* method with two arguments, one for the true case and one for the false case, in a similar way that `color:withSize:`. Therefore you should not put a period after the `]` following the `ifTrue:`.

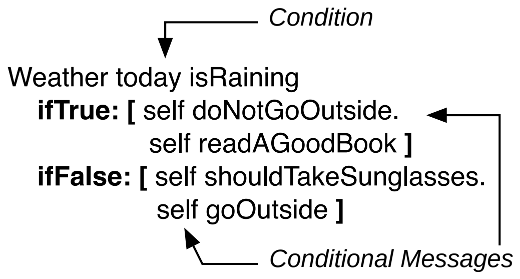


Figure 5.2: Conditional with two branches: one for true and one for false

Note that the method `ifFalse:ifTrue:` also exists and that it works the same way the method `ifTrue:ifFalse:`, *i.e.*,it will execute the false case when the condition is false and the true one when the condition is true, exactly as the method `ifTrue:ifFalse:`. This method is just there to help you writing more readable code if you want to start reading the messages executed when the condition is false as shown by the Figure 5.2.

Important Messages 5.1

```
aCondition
   ifTrue: [ messagesIfConditionIsTrue]
   ifFalse: [ messagesIfConditionIsFalse ]


aCondition
   ifFalse: [ messagesIfConditionIsFalse ]
   ifTrue: [ messagesIfConditionIsTrue ]
```

About Returned Value. We suggest you to read the section 4 that explains the important point about the returned value of a method that is central to the concept of condition. Indeed for a condition we are not only interested by the execution of a message *but also* by the result it computes and returns. For example in the method 5.1, the expression `self distanceFrom: World bounds center` not only computes it but *returns* an integer that represents the distance of the receiver from the center of the screen, the condition used this value to decide wich branch should be executed. The condition `dist < 200` returns a boolean, `true` or `false`, which is used by the conditional expression.

2 `ifTrue:` and `ifFalse:`

Sometimes we do not need two conditional branches. We only need to perform one action when a given condition is true but nothing when the condition is false or vice versa. For example the method `redWhenCloseToCenter` (5.3) only changes the color of the receiver to red when it is at a distance smaller than 200.

Method 5.3

```
redWhenCloseToCenter

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

Using the method `ifTrue:ifFalse:` method we just have to leave the second branch empty. However, Squeak offers the methods `ifTrue:` and `ifFalse:` to express this kind of conditional expressions. The method `ifTrue:` executes its conditional messages when its condition is true. Using the method `ifTrue:` the method redWhenCloseToCenter (5.3) is expressed as in method 5.4.

Method 5.4

```
redWhenCloseToCenter

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

Contrary to `ifTrue:`, the method `ifFalse:` executes its conditional messages when its condition is false. We can always use an `ifFalse:` method instead of a `ifTrue:` method by *negating* the condition as shown by the new version of the method redWhenCloseToCenter (5.5).

Method 5.5

```
redWhenCloseToCenter

   | dist|
   dist := self distanceFrom: World bounds center.
   (dist < 200) not
      ifFalse: [self color: Color red]
```

The methods `ifFalse:` and `ifFalse:` follow the template shown below. Both execute a condition and depending on the value returned by the condition execute or not the conditional messages.

Important Messages 5.2

aCondition
   ifTrue: [*messagesIfConditionIsTrue* **]**

aCondition
   ifFalse: [*messagesIfConditionIsFalse* **]**

A Subtle Difference. 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 condition of the two conditional expressions is executed twice. This can be a problem when conditional messages of the first conditional expression (`ifTrue:`) modifies what is tested by the condition of the second conditional expression (`ifFalse:`). This particular point is explained in detail in 2

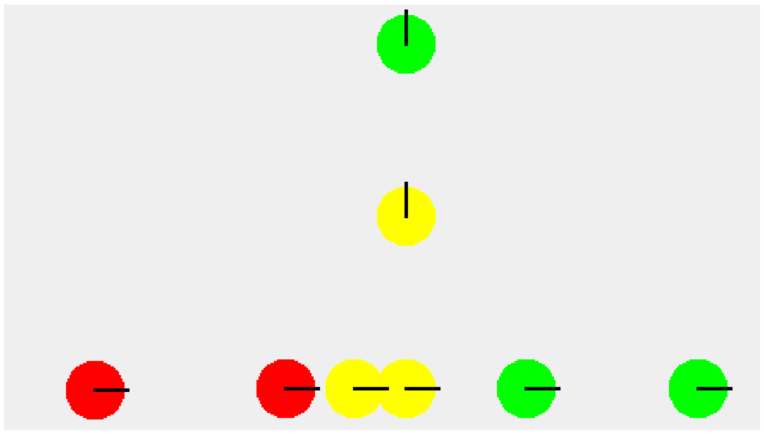


Figure 5.3: Various turtles at different distances from a point.

3 Nesting Conditional Expressions

A conditional expression can contain any other messages and in particular other conditional expressions. This is what we will present now. There is nothing spectacular but it is common that's why we want to show it to you. Conditions can be nested inside conditions.

Another Simple Problem. Let us modify our previous problem. Now we would like that if a turtle is located at a distance smaller than 200 pixels from a point it should be red, when it is between 200 and 300 pixels it should be yellow and at a distance greater than 300 it should be green. The Figure 5.3 shows various turtles that changed their color using the method `distanceDetector`.

What we see from our problem is that different parts of the method should be executed under different circumstances changing the color to yellow should be performed under conditions that are different than changing the color to green.

A possible solution to our problem is shown by the method 5.6.

Method 5.6

```
threeColorDetector

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

We have two different conditions that we hilighted in italics and in bold as shown explicitly in the script 5.2 and in the method 5.7. The second condition (in bold) is only executed whether the condition of the first one is false. Here when the distance is smaller than 300 the condition 2 is executed which means that its condition is executed and that depending on its values the right branch is executed. Take a colored pen and for one particular value of the distance, underline the part of the method that will be executed, you will see that only certain branches are executed.

Script 5.2 (*The two conditions of `threeColorDetector`*)

Condition 1:
```
dist > 300
      ifTrue: [self color: Color green]
      ifFalse: [ ... ]
```

Condition 2:
```
dist < 200
         ifTrue: [self color: Color red]
         ifFalse: [self color: Color yellow]
```

Method 5.7

```
threeColorDetector

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

4 Learning from Errors

We would like to show you how to learn from an error. We defined the method `coloredTurn: anAngle` that change the color of a turtle is relation with the direction to which it is heading at. We decided that when the turtle was pointing to the north it should become blue to represent cold, red when pointing to the the south else it should be green. Our first definition of this method is presented in the method 5.8.

Method 5.8

```
coloredTurn: anAngle
     "change the color of the turtle 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]
```

This definition, however, is not correct. Before reading the explanation shown in the script 5.3 and the script 5.4, try to guess why it is wrong by yourself. This method is wrong because when the turtle is pointing to the north its color is green while it should be blue.

Script 5.3 (*Illustrating the bug)*

```
| caro |
caro := Turtle new.
caro coloredTurn: -90.
caro color -> Color red           "ok"
caro coloredTurn: 90.
caro color -> Color green.      "ok"
caro coloredTurn: 90.
caro color -> Color green      "wrong"
```

Why... Execute mentally the method 5.8 and identify why this method does not specify the expected behavior. In fact the problem is that even if the condition `self direction = 90` is true and that its associated block is executed, the method continues and evaluate the false branch of the last conditional statement, changing then the color of the turtle to green. script 5.4 illustrates the execution flow that leads to the problem.

Script 5.4 (*Analyzing the problem)*

```
caro coloredTurn: 90.

   self direction = 90 is true
       so we go into the ifTrue: [self color: Color blue]
       we evaluate the following
   self direction = -90 is false
       so we do not go into the ifTrue:  [self color: Color red]
       but go into the ifFalse: [self color: Color green]
```

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

Method 5.9

```
coloredTurn: anAngle
   "change the color of the turtle 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 Other Examples and Further Experiments

Now we show some examples of conditional expressions. We first start by improving the robustness of the script **??**. The script **??** was not robust enough because we could enter a string that was not representing a number and it would broke. The script 5.5 improves its robustness by checking that the answer represents a number using the message `isAllDigits`. The stair is drawn only when the answer represents a number.

Script 5.5 (*Interactive stair)*

```
| caro  |
answer := (FillInTheBlank
              request: ’Number of steps’
              initialAnswer: ’10’).
answer isAllDigits
   ifTrue: [caro := Turtle new.
           answer asNumber timesRepeat:
                  [caro
                     go: 10 ;
                     north ;
                     go: 10 ;
                     east]
```

In a Box. The method 5.10 shows a method that allows a turtle to move forward only if it stays inside the specified rectangle as shown by the script 5.6. We will extend this script in the chapter **??**.

Method 5.10

```
go: anInteger ifStayInBox: aRectangle
   "Move foward a turtle only if the move will let the receiver into
   the rectangle specified by aRectangle"

   (aRectangle
      containsPoint: (self positionInDirectionForDistance: anInteger))
         ifTrue: [self go: anInteger.]
```

Script 5.6 (*Example of `Wider coloring`.)*

```
Turtle new
   go: 100
   ifStayInBox: (Rectangle center: World center extent: 400 )
```

Wider coloring. We would like to change the color of a turtle to blue when its is between 45 and 125, red when it is pointing between -45 and -125. Try to use the method finder (**??**) to find a method that would help you to simplify the expression of the condition.

the method 5.11 shows you one possible solution. The use of the method `between:and:` provides a readable way to express certain conjunction. `5 between: 0 and: 10` is equivalent to `(5<10) & (5>0)`.

Method 5.11

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

   self turn: anAngle.
   (self direction between: 45 and: 125)
      ifTrue: [^self color: Color blue].
   (self direction between: -45 and: -125)
      ifTrue: [^self color: Color red].
   self color: Color green
```

Interpreting. Imagine that we would like that a turtle understand a mini language constituted by characters such as $g to move forward and $t to turn of 45 degree as shown in script 5.7. Note that this method will be reused in the chapter **??** in which we will show how to use collections.

Script 5.7 (*Using* `interpret: aCharacter`)

```
| aTurtle |
aTurtle := Turtle new.
4 timesRepeat:
   [aTurtle
      interpret: $g;
      interpret: $t;
      interpret: $g;
      interpret: $g;
      interpret: $t;
      interpret: $g]
```

The method `interpret` can be defined as follows:

Method 5.12

```
interpret: aCharacter

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

Further Experiments. Enhance the method `interpret: aCharacter` so that gorG makes the turtle goes forward and that torT makes it turns. Add also that $+ makes the turtle turning left and $- right.

6 Summary

Method	Description
`aCondition ifTrue: [messagesIfConditionIsTrue]`	Execute messagesIfConditionIsTrue only if aCondition is true. If a turtle is pointing to the north, it turns green. `self direction = 90 ifTrue: [ self color: Color green]`
`aCondition ifFalse: [messagesIfConditionIsFalse]`	Execute messagesIfConditionIsFalse only if aCondition is false. The system beeps only when the turtle is not pointing to the north. `self direction = 90 ifFalse: [Smalltalk beep]`
`aCondition ifTrue: [messagesIfConditionIsTrue] ifFalse: [messagesIfConditionIsFalse]`	Execute messagesIfConditionIsTrue whether aCondition is true otherwise execute messagesIfConditionIsFalse. `self direction = 90 ifTrue: [self color: Color green] ifFalse: [Smalltalk beep]`
`aCondition ifFalse: [messagesIfConditionIsFalse] ifTrue: [messagesIfConditionIsTrue]`	Execute messagesIfConditionIsTrue whether aCondition is true otherwise execute messagesIfConditionIsTrue. The robot pick a diamond when it can otherwise the system beeps. `self direction = 90 ifFalse: [Smalltalk beep] ifTrue: [self color: Color green]`

6

Conditional Loops

Having conditions is a good tool for expressing complex programs. However, conditions are not enough. Sometimes we would like to combine loops and conditions. In fact we would like conditional loops, *i.e.*,loops that repeat sequence of messages until certain condition hold. In this chapter we present the conditional loops offered by Squeak using simple examples. We will use heavily conditional loops to simulate animal behavior and other strategies, such as escaping mazes or following paths.

1 Conditional Loops

The idea behind conditional loops is that a sequence of messages is repeated while a certain condition holds. Squeak defines two messages `whileTrue:` and `whileFalse:` that allow one to define conditional loops as shown by the templates below. The Figure 6.1 shows that a conditional loops is composed of a *condition* and *conditional messages*.

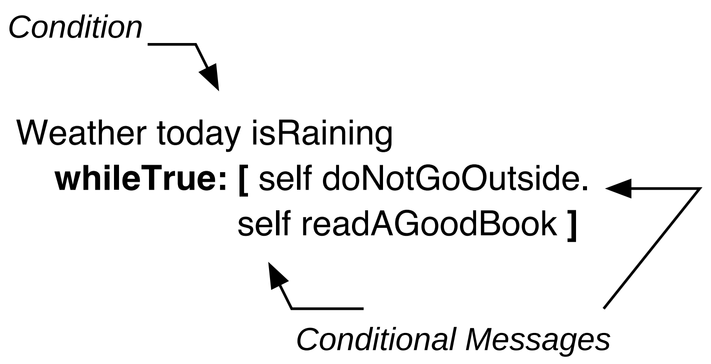


Figure 6.1: The `whileTrue:` conditional loops is composed of a condition and a sequence of conditional messages.

Important Messages 6.1

```
[ condition ] whileFalse:
  [ conditional messages ]

[ condition ] whileTrue:
  [ conditional messages ]
```

An Example. Let us take a simple example to illustrate their use. Imagine we want a turtle to move in the direction of the north until its y coordinate is smaller than 100 pixels. A solution using a conditional loop is shown by the method 6.1 as invoked in the script 6.1.

Script 6.1 (*Invoking* `upTo100`)

```
| caro |
caro := Turtle new.
caro upTo100
```

Method 6.1

```
upTo100
   "Make forward the receiver until its ordinates is smaller than 100"

   self north.
   [self center y > 100]
      whileTrue: [self go: 10].
   self color: Color green.
```

Let us carefully look at this method.

1. The expression `self north` is not part of the conditional loop, therefore it is executed once.
2. Then the condition expressed as a block `[self center y > 100]` is executed. When the result of the condition is true and only then the conditional messages specified as argument of the method `whileTrue:`, `[self go: 10]`, are executed. Once the execution of the conditional messages terminates the process restarts as in point 2.
3. When the result of the condition `[self center y > 100]` is false, the argument of the `whileTrue:` method is *not* executed and the loop stops. The messages following the conditional expression are executed: here the expression `self color: Color green` gets executed and the method terminates.

Sometimes programming conditional loops is difficult, because we forget to check carefully the condition and how the loop changes to tend towards the end of the condition. We strongly suggest you to open a transcript and include a trace to understand how the loop behaves as shown in the method 6.2. The Figure 6.2 shows the result of an execution.

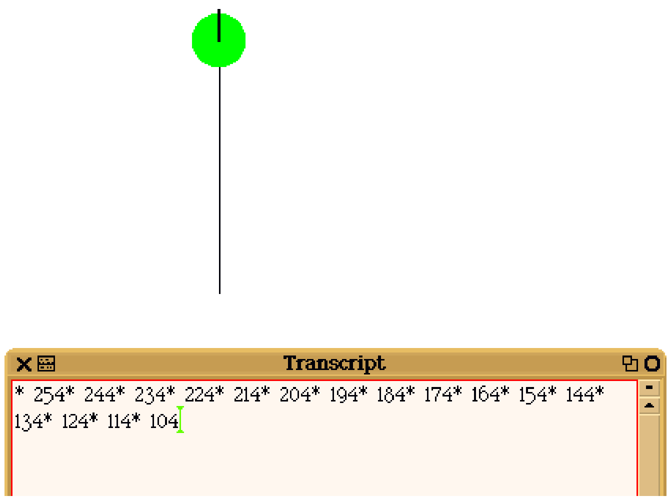
Method 6.2

```
upTo100
   "Make forward the receiver until its ordinates is smaller than 100"

   self north.
   [self center y > 100]
      whileTrue:
         [Transcript show: '* ' , self center y printString.
         self go: 10]
    self color: Color green.
```

As for `ifTrue:` and `ifFalse:`, `whileTrue:` can be substitued by `whileFalse:` by negating the condition. Use the method that helps you to understand the algorithm you are defining.

Further Experiences. The place where you introduce the logging line has also an impact on the resulting trace. As alternative experience, introduce the line `Transcript show: '* ' , self center y printString.` after the expression `self go: 10` or even in the first block before the first line as shown by the method 6.3.



```
* 254* 244* 234* 224* 214* 204* 194* 184* 174* 164* 154* 144*
134* 124* 114* 104
```

Figure 6.2: The trace of the execution of the method `upTo100`.

Method 6.3

```
upTo100
   "Make forward the receiver until its ordinates is smaller than 100"

   self north.
   [Transcript show: 'c ' , self center y printString; cr.
   self center y > 100]
      whileTrue:
         [Transcript show: '* ' , self center y printString; cr.
         self go: 10].
   self color: Color green
```

Method 6.4

```
upTo100
   "Make forward the receiver until its ordinates is smaller than 100"

   self north.
   [self center y > 100]
      whileTrue:
         [ self go: 10
        Transcript show: '# ' , self center y printString]
```

Compare the traces produced by the scripts 6.2, 6.3, and 6.4. Look in particular at the first values. Perform the following experience: move the turtle by using the black halo close to the top edge of the window, *i.e.*, be sure its y coordinate is smaller than 100. Then invoke the method `upTo100`, as you should see nothing happens. This is normal, the method is invoked and the expression `self center y > 100` is false as the position of the turtle is smaller than 100. Therefore the conditional messages are not executed.

2 About the use of `[]`

You may have difficulties to remember when to put `[]` and not. There are basically two rules in Squeak. You surround an expression with `[` and `]` when:

- you need to execute several times the same expression. For example,
  - `4 timesRepeat: [caro go: 10; turnLeft:90]` repeats 4 times the messages `caro go: 10; turnLeft:90`,
  - `1 to: 10 do: [:i | Transcript show: i printString ; cr]` repeats ten times the `[:i | Transcript show: i printString ; cr]` which prints the number to the transcript.
- the expression is not always executed. For example,
  - `dist < 200 ifTrue: [self color: Color red]` only executes `self color: Color red` under certain circumstances,
  - `[self center y > 100] whileTrue: [self go: 10]` repeats multiple times conditionally both `self center y > 100` and `self go: 10`, therefore the receiver and the argument are blocks.

3 Breaking a Loop

It is not exceptional to write endless loop, *i.e.*,expression where the sequence of messages is continously executed. Practically, this happens because the boolean expression never returns true for `whileFalse:` and false for `whileTrue:`. You can stop a loop by pressing Apple-. on Mac or Alt or Control-C on other platform. Then to understand why the loop does terminate you can use the debugger that pops up and by clicking its Debug button.

In fact a conditional loop can endlessly loops when the conditional messages do not perform an action that tends towards a state that would invalidate the condition. Let us illustrate this difficult point in the case of our example. When we look at the trace shown in the Figure 6.2 we see that the distance between the turtle and the vertical point having 100 as y value gets smaller and smaller. In the example, to be sure that the loop gets a chance to terminate, the second block of the loop should somehow reduce the distance.

Even if this may looks obvious the method 6.5 cannot terminate because the actions of the second block does not tend towards negating the first block. Here, the second block increase the value of the y coordinates, therefore the chance that the first block may evaluate to false reduces each turn. This example is exaggerated but it illustrates clearly the problem of specifying loops that terminate.

Method 6.5

```
upTo100Infinite
   "Make forward the receiver until its ordinates is smaller than 100"

   self south.
   [self center y > 100]
      whileTrue: [self go: 10].
   self color: Color green
```

When you are defining a loop always ask yourself if there is a possibility that the first test gets invalidated. This seems obvious but if the tests cannot be invalidated the loop will never finish.

Another Experiment. The method `whileFalse:` is symetric to `whileTrue:` therefore try to express the method `upTo100` but using the method `whileFalse:`. Try to make the turtle moving one pixel by one pixel and compare the exact position where it stops.

4 Deeper into `whileTrue:` and `whileFalse:`

In fact the condition does not have to only contain a boolean expression. It can contain a sequence of messages as soon as the last conditional message returns a boolean as shown by the following template. This allows one to express different conditional loops. Note that other programming languages do not allow this, but provide different conditional loops.

Important Messages 6.2

```
[self doThis.
anObject doThat.
self isStillWorking] whileTrue:
   [self grumbleAndKeepOnWorking]
```

Therefore we can change the method `upTo100` to be as the method 6.6. As its name tells us, while this method looks nearly the same as the `upTo100`, it is different. Try to understand where is the difference. For example, add a trace in the method 6.6 and analyze it. Note that it is equivalent to print as last expression of the second block since the first block here does not modify anything.

Method 6.6

```
notTheSameUpTo100
   "Make forward the receiver until its ordinates is smaller than 100"

   self north.
   [self go: 10.
   self center y > 100]
      whileTrue: [ ].
   self color: Color green
```

As the first block is at least executed once, the main difference is that the turtle will move even if it is already at position smaller than 100.

5 For User Input Control

Conditional loops can be used to ask some values to the user until correct values are given. In the script 5.5 if the user enters a string not representing a number, the script would detect it and ends without drawing the stair. Now the next step is to force a user to enter a value that is a number. We can do this by using a conditional loop that is repeatedly asking for a value if the previous value was not a number. The script 6.2 shows such a version.

Script 6.2 (*Interactive stair)*

```
| caro answer |
[ answer := (FillInTheBlank
              request: ’Number of steps’
              initialAnswer: ’10’).
answer isAllDigits] whileFalse: [].
caro := Turtle new.
answer asNumber timesRepeat:
   [caro
      go: 10 ;
      north ;
      go: 10 ;
      east]
```

6 Summary

Method	Description
`[aCondition] whileFalse: [SequenceOfMessages]`	Execute SequenceOfMessages only if *aCondition* is false.
`[ aCondition ] whileTrue: [SequenceOfMessages]`	Execute SequenceOfMessages only if *aCondition* is true.

7

Boolean and Boolean Expressions

Conditional expressions require the notion of boolean expressions, *i.e.,*expressions whose value are true or false. In this chapter we go deeper in the notion of boolean as it is a key concept of programming languages then show how to express boolean expressions and we also present some of the most common errors that happen with parentheses.

1 Booleans and Booleans Expressions

Boolean expressions are expressions that return true or false values. Such values are called boolean[1] as they can *exclusively* be true or false. In programming languages booleans are important because they serve as basis for conditional execution.

Booleans. Booleans represent true or false facts. For example, a true fact is the fact that *2 + 2 is equals to 4* or that *the sun gets up at the east*.

In Squeak, booleans as everything else are objects. In fact there are two objects for representing true and false. The object `true` that represents the meaning "it is true" and the object `false` [2] that represents the meaning "it is false". `true` is an instance of the class `True` that defines the behavior that the object `true` understands. `false` is created by the class `False` that defines the behavior that the object `false` understands. These behaviors are important because they are used to compose boolean expressions as we show below. Note that even if `true` and `false` are objects in the same sense that a turtle was created out of the class `Turtle`, they are so central to Squeak that `true` and `false` are special variables. Hence, you do not have to create them using `new`, `true` and `false` exist and you do not have to worry about their creation.

Boolean Expressions. Boolean expressions are expressions that manipulate and *return* booleans. We can think about a boolean expression as a question whose answer is true or false. The script 7.1 presents some examples of boolean expressions and the kinds of questions they express.

[1]The adjective *boolean* comes from George Boole, an English mathematician of the nineteenth century. He discovered that logical propositions could be manipulated like mathematical objects.

[2]Pay attention `true` and `false` start with a lowercase letter.

Script 7.1 (*Examples of simple boolean expressions)*

```
Turtle new color = Color red
```
Is the color of a newly created turtle red?

```
Turtle new center = 100@100
```
Is a turtle located at the position 100@100?

```
Time now > (Time new hours: 8)
```
Is the time now after 8 o'clock?

```
b2 isOnYellow
```
Is the robot named b2 on a yellow tile?

```
b2 canMoveForward
```
Can the robot named b2 move forward?

```
| aTurtle |
aTurtle := Turtle new.
aTurtle go: 100.
(Rectangle origin: 100@200 corner: 300@400)
    containsPoint: aTurtle center
```
Is a turtle inside the rectangle 100@200, 300@400?

Note that certain questions could be immediately answered while others require to know the context of the boolean expression execution. Evaluate and print the results of the boolean expressions, then change the context by modifying for example the turtle or the other participants of the boolean expressions.

The expressions presented in the script 7.1 are basic ones. It is often not sufficient and we need to compose them by negation (not), conjunction (and), or alternative (or). These three compositions are defined in Squeak using the messages `not`, |, and **&** as shown by the following templates.

Important Messages 7.1

aBooleanExpression **not**
aBooleanExpression **|** *anotherBooleanExpression*
aBooleanExpression **&** *anotherBooleanExpression*

The script 7.2 presents some examples of composed boolean expressions. We detail below the main way of composing boolean expressions.

Script 7.2 (*Examples of composed boolean expressions)*

```
(Turtle new color = Color red) not
```
Is the color of a turtle different than red?

```
| aTurtle |
aTurtle := Turtle new.
(aTurtle direction = -90) & (aTurtle direction = 90)
```
Is a turtle located at the position 100@100 and heading at the north?

```
Time now > (Time new hours: 8) |  (Date today  weekday = #Sunday)
```
Is the time now after 8 o'clock or are we Sunday?

```
Time now > (Time new hours: 8) |  (Date today  weekday = #Sunday) not
```
Is the time now after 8 o'clock or are we not Sunday?

Negation (not). Negation is useful to express the contrary of something. In Squeak it is based on the message `not`. The message `not` simply negates the boolean expression to which it is sent. In the last line of the script 7.3, the message `not` is sent to the expression `(aTurtle color = Color red)`. If such an expression is true then its negation will be false and vice versa.

Script 7.3 (*Example of negation)*

```
| aTurtle |
aTurtle := Turtle new.
aTurtle color: Color green.
(aTurtle  color = Color red) not
```

Is the color of a turtle something else than red?

Conjunction (and). The term *conjunction* literally means together. A conjunction is used to express that we want to know whether two boolean expressions are true. In Squeak, a conjunction is defined using the message & sent to a boolean expression with another boolean expression as argument. A conjunction is only true when both expressions that composed it are true. In the script 7.4, the composed expression will only be true, if `(aTurtle center = 100@100)` *and* `(aTurtle direction = 90)` are true simultenaously.

Script 7.4 (*Example of conjunction)*

```
(aTurtle center = 100@100) & (aTurtle direction = 90)
```

Is a turtle located at the position 100@100 and heading at the north?

Alternative (or). Alternative is used to express the notion of choice. An alternative is defined using the message | sent to a boolean expression with a another boolean expression as argument. An alternative is used to express that we want at least one of the boolean expressions to be true. Therefore a conjunction is true as soon as one the expressions it is composed of is true.

In the script **??**, the composed expression is true, as soon as one of the two expressions `(aTurtle center = 100@100)` or `(aTurtle direction = 90)` is true.

Script 7.5 (*Example of alternative)*

```
(aTurtle direction = -90) | (aTurtle direction = 90)
```

Is a turtle located at the position 100@100 or heading at the north?

The last example of script 7.2 shows that we can compose boolean expressions multiple times, negate them, group them by alternative (or) or conjunction (and) to represent complex conditions. The following table shows the most common boolean operations.

Kind	Message	Results
Negation	`not`	
Examples	`false not`	`true`
	`true not`	`false`
	`(aTurtle color = Color red) not`	`true` or `false`
Conjunction (and)	`&`	
Examples	`true & true`	`true`
	`false & true`	`false`
	`true & false`	`false`
	`false & false`	`false`
	`(aTurtle center = 100@100) & (aTurtle direction = 90)`	`true` or `false`
Alternative (or)	|	
Examples	`true` | `true`	`true`
	`false` | `true`	`true`
	`true` | `false`	`true`
	`false` | `false`	`false`
	`Time now > (Time new hours: 8)` | `(Date today weekday = #Sunday)`	`true` or `false`

Hot Pepper

`true` and `false` start with a lowercase letter. `true` is created by the class `True` that defines all the behavior that the object `true` understands. `false` is created by the class `False` that defines all the behavior that the object `false` understands.

The classes `True` and `False` defines the behavior such as and (&, `not`, |) that allows one to compose expressions. Indeed, the result of boolean expressions is either true either false so the message &, `not`, or | are sent to the objects `true` or `false`.

Hot Pepper End

2 The Frequent Mistakes: Missing Parenthesis

It may happen that you get some trouble with the syntax of Squeak. Consider that everybody falls into this kind of mistakes. Even confirmed programmers do. The difference between a beginner and a confirmed programmer is not that one does mistakes and the other don't. The main difference is that a confirmed programmer goes much faster to identify and fix them.

Missing parenthesis is often a source of mistakes, therefore we show you how to analyze the errors you may get. Basically when you are composing a boolean expression you have to identify clearly to which expression the messages `not`, |, and & are sent to. Let us illustrate the problem.

2.1 A Case Study.

Script 7.6 (*Missing Parenthesis to Identify the receiver of a* `not`*)*

```
Turtle new  color = Color red not
```

The script 7.6 shows a boolean expression that fails to represent the following question: is the color of a newly created turtle different than red (not red)?. Executing this script leads to an error because parenthesis are missing as we will explain.

Execute the expression describe in the script 7.6, open the debugger on the error and select the first line in the top pane to obtain the Figure 7.1. The title window of the debugger already gives us some information. It states that the receiver does not understand the message `not`. Now when we select the topmost line of the top pane we see the body of the method `doesNotUnderstand:` which was called as the receiver did not understand the message `not`. When we click on `self` in the left bottom pane, we see that the receiver is not a boolean as it should be but a color! If you click on the `aMessage` on the right bottom pane, you will see which message was not understood as shown in the Figure 7.1. In our case, we get `a Message with a selector: #not and arguments: #()` which means that the message with the selector `not` did not have any arguments. This means that the message `not` has not be sent to the right receiver. The message `not` is sent to the result of the expression `Color red` which is a color and does not understand the message `not`.

The reason why the message `not` is sent to the expression `Color red` and not to the complete boolean expression is related to the way Smalltalk executes expressions. First the expressions enclosed by parentheses are executed, then the unary messages, then the binary and finally the keywords-based messages as explained in the chapter **??**. In our case the message `not` been an unary message is evaluated before the binary message =, therefore it is sent to the result of the expression `Color red`. To get the correct execution order, we have to enclose the expression in parentheses as shown in the script 7.3, this way the message `not` will be sent to the result of the = message.

The script 7.7 shows how the messages are executed in the wrong expression.

Script 7.7 (*How messages of the script 7.6 are executed)*

The expression: `Turtle new  color = Color red` **`not`**
is executed as it would be written fully parenthesized as follow:
```
(((Turtle new) color) = ((Color red) not))
```
Therefore first both parts of the binary method = are evaluated:
```
((Turtle new) color) -> aColor
((Color red) not) -> (aColor not) -> error
```
Then the = message would be executed but the error stops the execution process.

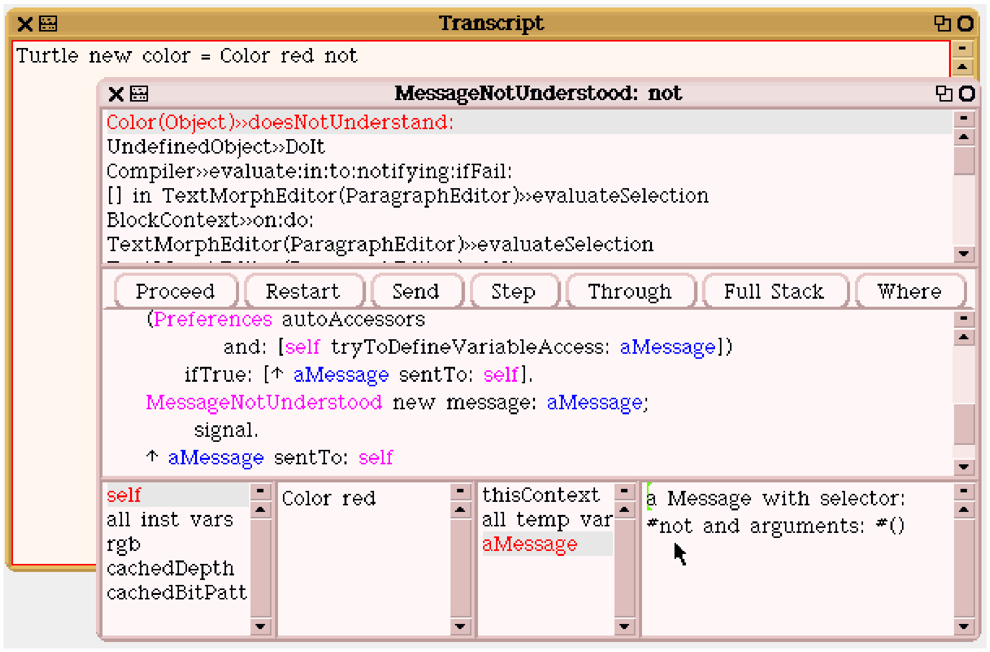
The script 7.7 shows how the wrong expression is executed by showing how it is equivalent to the same script fully parenthesized. Compare it with the script 7.8 which shows how the correctly parenthesized expression is evaluated.

Script 7.8 (*How messages are executed with the correct expression)*

The expression: **`(`**`Turtle new  color = Color red`**`) not`**
is executed as it would be written fully parenthesized as follow:
```
(((Turtle new) color) = (Color red)) not)
```
Therefore first both parts of the binary method = are evaluated.
Both return a color possibly equal
```
((Turtle new) color) -> aColor
(Color red) -> anotherColor
```
Then the = message is executed, i.e., send to the result of the right hand expression.
The execution of the message = returns a boolean.
```
(aColor = anotherColor) not -> aBoolean not -> aBoolean
```

2.2 Similar Problems and Solutions

It would be boring to explain the similar problems you may encounter with the other messages such as & and |. Try to execute the scripts 7.9 and 7.11, and to understand the problems. We show you the corresponding correctly parenthesized scripts 7.10 and 7.12.



```
Turtle new color = Color red not
```

```
Color(Object)»doesNotUnderstand:
UndefinedObject»DoIt
Compiler»evaluate:in:to:notifying:ifFail:
[] in TextMorphEditor(ParagraphEditor)»evaluateSelection
BlockContext»on:do:
TextMorphEditor(ParagraphEditor)»evaluateSelection
```

```
    (Preferences autoAccessors
            and: [self tryToDefineVariableAccess: aMessage])
        ifTrue: [↑ aMessage sentTo: self].
    MessageNotUnderstood new message: aMessage;
        signal.
    ↑ aMessage sentTo: self
```

```
self
all inst vars
rgb
cachedDepth
cachedBitPatt
```

```
Color red
```

```
thisContext
all temp var
aMessage
```

```
a Message with selector:
#not and arguments: #()
```

Figure 7.1: The message `not` is not sent to the complete boolean expression `Turtle new color = Color red`, but sent to the expression `Color red` which returns a color. Therefore it is not understood.

Script 7.9 (*Missing Parenthesis to identify the receiver of a &.*)

```
| aTurtle |
aTurtle := Turtle new.
aTurtle center = 100@100 & aTurtle penSize = 5
```

Script 7.10 (*Identifying the receiver of a & using parenthesis.*)

```
| aTurtle |
aTurtle := Turtle new.
(aTurtle center = 100@100) & (aTurtle penSize = 5)
```

Script 7.11 (*Missing Parenthesis to identify the receiver of |.*)

```
| aTurtle |
aTurtle := Turtle new.
aTurtle center = 100@100 | aTurtle direction = 90
```

Script 7.12 (*Identifying the receiver of | using parenthesis.*)

```
| aTurtle |
aTurtle := Turtle new.
(aTurtle center = 100@100) | (aTurtle direction = 90)
```

3 Lazy Boolean Methods

For completeness we show you now a particular aspects of boolean methods in Smalltalk. In a first reading, we suggest you to skip this part.

In Smalltalk, booleans offers another kind of conjunction and alternative methods `and:` and `or:` called *lazy* ones. They are called lazy because they only evaluate their argument if required. The lazy and (`and:`) only evaluates its argument if the receiver is true. The lazy or (`or:`) only evaluates its argument if its receiver is false. The arguments of these lazy methods are expressed as block which this way defer their evaluation.

The following examples illustrate the difference between lazy and non lazy messages. In the following example, the first expression `(1=2)`, the receiver of the message `and:` is false because 1 is not equals to 2, so the argument is not evaluated because the two conditions will never be true together regardless the value of the argument. Therefore it does not raise a division by zero. In the second example, the & message evaluates both the receiver and the argument automatically regardless the receiver value so the division by zero raises an error. The third example raises an error because the receiver is true and we have to evaluate the argument to determine the value of the composed expression.

```
(1=2) and: [10 / 0]
```
Does not raise a division by zero error because the receiver is false so the conjunction will never be true therefore there is no need to evaluate the argument.

```
(1=2) & (10 / 0)
```
Raise a division by zero error.

```
(1=1) and: [10 / 0]
```
Raise a division by zero error because 1=1 is true and the second condition is evaluated.

Important Messages 7.2

```
aBooleanExpression and: [ anotherBooleanExpression ]
```

Similarly for the lazy `or:`, one does not need to evaluate the second condition if the receiver is true. Therefore, the first example does not raise an error because the receiver is true. In the second example dim evaluates the receiver and the argument regardless the value of the receiver, so an error is raised. In the third example, the receiver is false so we need to evaluate the argument to get the value of the composed expression. So an error is raised.

```
(1=1) or: [10 / 0]
```
Does not raise a division by zero error because the receiver is true and there is no need to evaluate the argument as at least one true is enough for an alternative to be true.

```
(1=1) | (10 / 0)
```
Raise a division by zero error.

```
(1=2) or: [10 / 0]
```
Raise a division by zero error.

Important Messages 7.3

```
aBooleanExpression or: [ anotherBooleanExpression ]
```

Lazy messages are interesting because they allow one to express conditions in easier way. The second condition does not have to be valid all the time. Moreover, they are useful to avoid to compute unnecessary computation and are good to optimizing conditions. The method `containsPoint: aPoint` defined for rectangles illustrates this behavior. This method determines whether a rectangle contains a point.

Method 7.1

```
Rectangle>>containsPoint: aPoint
   "Answer whether aPoint is within the receiver."

   ^origin <= aPoint and: [aPoint < corner]
```

If the origin (the topleft corner) of the rectangle is bigger than the point, there is no point to check whether the point is smaller that the rectangle corner (bottom right corner).

4 Summary

Booleans are true or false.

Booleans expressions are expression that manipulate booleans and whose values are booleans.

Complex boolean expressions can be composed of simple boolean expressions using conjunction (and), alternative (or), and negation (not).

Kind	Message	Results
Negation	`not`	
Examples	`false not`	`true`
	`true not`	`false`
	`(aTurtle color = Color red) not`	`true` or `false`
Conjunction (and)	`&`	
Examples	`true & true`	`true`
	`false & true`	`false`
	`true & false`	`false`
	`false & false`	`false`
	`(aTurtle center = 100@100) & (aTurtle direction = 90)`	`true` or `false`
Alternative (or)	`|`	
Examples	`true | true`	`true`
	`false | true`	`true`
	`true | false`	`true`
	`false | false`	`false`
	`Time now > (Time new hours: 8) | (Date today weekday = #Sunday)`	`true` or `false`

8

Various Points about Conditions

In this chapter we present some extra points about conditional such as how to use conditional to debug your code. The material presented here is more advanced and can be skip in a first reading.

1 Debugging using Conditional

Often one would like to check whether a conditional code has been executed. One way to do that is to insert expression such as `Transcript show: 'I pass there' ; cr.`. Another approach is to insert `self halt` that when executed opens a debugger. A third way is insert some messages that produce sounds.

Script 8.1 (*Only clink)*

```
| caro |
caro := Turtle new.
caro distanceDetector.
caro color = Color red
   ifTrue: [Smalltalk beep].
caro jump: 250.
caro distanceDetector.
caro color = Color green
   ifTrue: [Smalltalk beep].
```

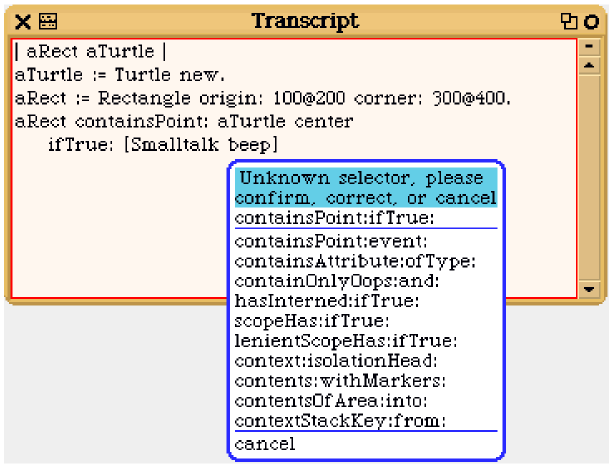
Another interesting use of conditional is to specify conditional breakpoints. For example, when we want to stop when a loop arrives at a given point we just have to include an expression similar to `myCondition ifTrue: [self halt]`.

An interesting way to have a conditional stop of a program execution is to check when certain keys are pressed such as control or shift. The expression `InputSensor default shiftPressed ifTrue: [self halt]` only opens a debugger when the shift key is pressed. The class `InputSensor` defines other messages such as `commandKeyPressed`, `controlKeyPressed`, or `anyButtonPressed` that can be used to define conditional break points.

2 Tricky Aspects of Parenthesis

It may happen that you get some trouble with the syntax when using the methods `ifTrue:` and its companion methods `ifFalse:`, `ifTrue:ifFalse:`, and `ifFalse:ifTrue:`. Indeed you have to take care that the the boolean expression may be parenthesed.

In the script **??** there is no problem because the expression used in the boolean expression, `Time now > (Time new hours: 8)` are not keywords messages, so they are evaluated prior to the `ifTrue:` that is executed if the result of the expression is true.



```
| aRect aTurtle |
aTurtle := Turtle new.
aRect := Rectangle origin: 100@200 corner: 300@400.
aRect containsPoint: aTurtle center
    ifTrue: [Smalltalk beep]
```

```
Unknown selector, please
confirm, correct, or cancel
containsPoint:ifTrue:
containsPoint:event:
containsAttribute:ofType:
containOnlyOops:and:
hasInterned:ifTrue:
scopeHas:ifTrue:
lenientScopeHas:ifTrue:
context:isolationHead:
contents:withMarkers:
contentsOfArea:into:
contextStackKey:from:
cancel
```

Figure 8.1: The environment is trying to recover an error due to missing parenthesis. The message is not `containsPoint:ifTrue:` and the boolean expression containing the message `containsPoint:` should be enclosed by parentheses.

In the script 8.2 the boolean expression uses a keyword message, `containsPoint:`, which has the *same weight* than the `ifTrue:` message. So the boolean expression will not be evaluated first as the system believes that the message is in fact named `containsPoint:ifTrue:`. To disambigate this situation parenthesis are required as shown in the script 8.2.

Script 8.2 (*Conditional Requiring Parenthesis)*

```
| aRect aTurtle |
aTurtle := Turtle new.
aRect := Rectangle origin: 100@200 corner: 300@400.
(aRect containsPoint: aTurtle center)
    ifTrue: [Smalltalk beep]
```

Important!

Surround boolean expressions with parenthesis when they contain keyword based messages to avoid message ambiguities

3 Factoring Commonalities

As we already mentioned it, duplicated logic is not a sign for potential improvements. The method `distanceDetector` exhibits some duplicated logic in each branch of the condition `self color:` . Here the duplication is not severe but we use this method as a pretext to illustrate our point.

Method 8.1

```
distanceDetector

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

Let us start by factoring commonalities out of the conditions. As we mentioned earlier, the code in the conditions does not have to be the same. The script 8.2 shows an equivalent version in which the expression `self color:` has been factored out of the condition. In this new version the conditional code returns the new color. The final line changes the receiver color using the selected color. This way is it easier to apply a consistent manipulation to the color such as darkening it.

Method 8.2

```
distanceDetector

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

4 About Method Returns

When a return statement, `^ anExpression`, is encountered, the execution of the method is terminated and the result of the expression following the return is returned to the caller of the method. No other piece of code is executed in the method containing the return statement. This behavior can be used to write condition is a different way. The method 8.3 shows the method 5.9 using explicit return. Here as soon as a boolean expression is true, the return statement is executed, *i.e.*,its expression is executed and the rest of the method is skipped.

Method 8.3

```
coloredTurn: anAngle
    "change the color of the turtle 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].
   self color: Color green]
```

Note that per default in Squeak, a method returns its receiver, even if the code does not show it as shown by the method 8.4 which is then equivalent to the method 8.5.

Method 8.4

```
doSomething

   Transcript show: ’doSomething’ ; cr.
```

Method 8.5

```
doSomething

   Transcript show: ’doSomething’ ; cr.
   ^ self
```

5 About Code Formatting

In Smalltalk, the code can be layouted in all kind of ways and the indentation (its shape regardings the left margin) has no semantics. However, using a clear indentation really helps the reader to understand the code. We suggest to follow the convention we chose to format `ifTrue:ifFalse:` expressions.

Again, the idea is that the block of expressions delimited by the characters `[` and `]` should form a visual and textual rectangle. First the boolean expression is always on one line and the keywords are always indented on the next line. Then when the block fits into a single line we just put it after the `ifTrue:` or `ifFalse:` keyword to gain vertical space and we close the block with the `]` at the end of the line as follow:

```
self direction = 90
      ifTrue: [self color: Color blue]
```

When the block cannot stand on one single line, we apply the same strategy that for `timesRepeat:`. We start the block with `[` on the next line and align all the expressions inside the block to one tab and finish by `]` that indicates that the block ends as follows:

```
dist < 200
      ifTrue: [self color: Color red.
              Transcript show: ’red’ ; cr]
      ifFalse: [self color: Color green
               Transcript show: ’green’ ; cr]
```

9

A Quick Look at Recursion

Recursive – Adj. Quality of something that is partly defined in terms of itself.
See recursive

Once we have conditionals we can express *recursive* methods, *i.e.*,methods that are partly expressed in terms of themselves. Recursive methods allows one to write extremely powerful algorithm in an elegant way. In this chapter we simply show you the main principle without into the details.

1 Picking Diamonds

Let us start with the following problem. Imagine we want to pick all the diamonds in the direction to which the robot is pointing at.

Now we can express

Method 9.1

```
In category recursion
Bot>>pickLine

   self canMoveForward
      ifTrue:
         [self safePick.
         self go.
         self pickLine]
```

How does it work? In fact writing recursive method is natural and simple. Writing recursive methods that do not loop endlessly is more complex. Let us look how this is working.

1. the method is invoked,
2. the condition is executed,
3. when the condition value is false the method terminates.
4. when the condition is true, the conditional messages are executed one by one, when the method `pickLine` is invoked the process starts as mentioned to the point 1.

The key point when defining a recursive method is to always check that the method contains at least one case that is not calling the method again. Here the non recursive part is implicit as we used the method `ifTrue:`, when the condition is false the conditional messages are not executed anymore and the method `pickLine` is not recursively invoked.

The second point is, as with conditional loops, that the recursive part of the method should somehow tend toward stopping the recursive calls. Here the fact that the area is bounded and that the bot moves always one step in the same direction ensure that at one point we are sure that the bot will not be able to move further.

The third point that will be shown in the section 5 is that a typical recursive method builds its result by composing the partial results obtained by recursive calls.

2 Learning from Error

While writing for the first time the method pickLine, we did a mistake, we forgot the expression `self go`. Can you imagine what was the result?

The robot could move so the method `canMoveForward` was true. Therefore the conditional messages were executed. A diamond was picked up if possible and then the method `pickLine` was called again. So we ended up in an endless loop.

What can we learned from this bug? An essential point, a recursive method have to always tend towards its non recursive part. This means that if we want that the method stops as it should, we should at least give me that chance to stop. When you are writing a recursive method always have in mind that you should (1) have a non recursive part, here the fact that we have a ifTrue: means that when the robot cannot move we will not be in the recursive part anymore, and (2) you should tend toward it, here we expect that by moving the bot forward it will end up bumping into the limits of its world and this will lead to executing the non recursive part, here doing nothing and ending the execution.

In fact the method `pickLine` is not working completely well as it will not pick a diamond that is just near a brick or near the limit of the world. First explain why by showing step by step why this situation occurs. Second propose a solution to this problem.

In fact picking a diamond can be done all the time therefore it does not require to only be executed within the conditional expression. The following method is

Method 9.2

```
Bot>>pickLine

   self safePick.
   self canMoveForward
      ifTrue:
         [self go.
         self pickLine]
```

The method `randomDirection` shows another recursive method. The method tries to find a new direction, *i.e.*,a point composed by -1, 0, or 1 with the extra constraint that x*y should always be equal to 0. When it did not find a point satisfying the constraint it just retries invoking itself. This behavior could be implemented using a conditional loops. Try to do it.

Method 9.3

```
Bot>>randomDirection
   "return a random direction i.e. a point composed by -1,0, or 1
   with the constraint that x * y should always be = to 0"

   | x y |
   x := 3 atRandom - 2.
   y := 3 atRandom - 2.
   ^ (x * y) isZero
      ifTrue: [x @ y]
      ifFalse: [self randomDirection]
```

3 Fun with Recursion

In this section we want to show you how to define intrigring curves, fractal curves whose definition is recursive. The idea of a fractal curve is to define (part of it) using its definition. Let us start to try to find the definition of the method that can draw the pictures shown in the Figure 9.1.

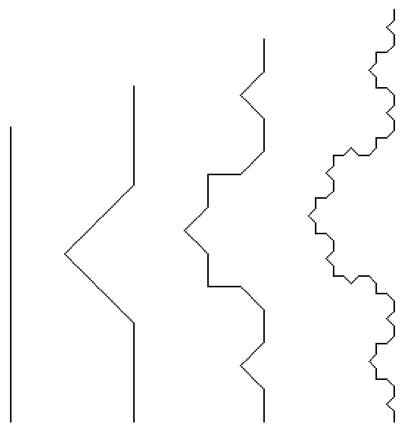


Figure 9.1: The `pic:n:` with n= 1, 2, 3, and 4.

The idea is that we take a single line, cut it in three equal parts and replace the middle segment by a shape, here simply two lines. Then we apply the same principle on the all the segments we create. We obtain the drawings shown in the Figure 9.1.

Let us start with a non recursive method. The method `pic:` shown hereafter 9.4 draws the second picture from the left in Figure 9.1: it replaces the middle segment of a line based on three segments by a simple shape, here a triangle.

Method 9.4

```
Turtle>>pic: size
   "self new north; pic: 200"

   | segSize |
   segSize := size / 3.
   self go: segSize.
   self turnLeft: 45.
   self go: segSize.
   self turnRight: 90.
   self go: segSize.
   self turnLeft: 45.
   self go: segSize
```

Note that we could have payed attention that the basis of the triangle would measure exactly a third of the original line but this does not change anything to what we want to show you and we suggest you to do it. Now we want to operate the same on each segments therefore we replace all the original `go:` calls by a call to the recursive method we are defining, we name it now `picRec:` to make it explicit and we obtained the method not working yet described in method 9.5.

Can you explain why this method is not working?

Method 9.5

```
Turtle>>picRec: size
   "Turtle clearWorld. self new north; picRec: 200 ; beInvisible"

   | segSize |
   segSize := size / 3.
   self picRec: segSize.
   self turnLeft: 45.
   self picRec: segSize.
   self turnRight: 90.
   self picRec: segSize.
   self turnLeft: 45.
   self picRec: segSize
```

There are two questions that this method should raise:

- How can it stop? Indeed there is no condition to stop the infinite recursive, each time `picRec:` is executed it has no alternative except to invoke itself infinitively.
- Where the job is done? Even if the method would contain a non recursive call. There should be a place where the line should be drawn.

Even if these two questions are quite rudimentary, every recursive method should provide an answer to have a chance to work. The final version of the method `picRec:` (method 9.6) first contains a non recursive branch, a reasonable condition as the length of the drawn segment is regularly divided, and a place where the curve is drawn.

What is worth to realize is that this method is only drawing from a single place `self go: size` while all the other invocations only modify the parameters and define a different execution contexts such as changing the direction of the turtle.

Method 9.6

```
Turtle>>picRec: size
   "Turtle clearWorld. self new north; picRec: 200 ; beInvisible"

   | segSize |
   size < 10
      ifTrue: [self go: size]
      ifFalse:
         [segSize := size / 3.
         self picRec: segSize.
         self turnLeft: 45.
         self picRec: segSize.
         self turnRight: 90.
         self picRec: segSize.
         self turnLeft: 45.
         self picRec: segSize]
```

Do the following experimentations:

- Replace some recursive calls to `picRec:` by `go:` and understand the result.
- Put a `self halt` in the middle to see that the all the recursive calls before the breakpoint are drwan but non after.
- Change the condition to understand.

To let you better experiment with the concept of recursive method, we replace the condition `size < 10` by an explicit counter as done in the method `picRec:n:` (method 9.7). Here the condition is now

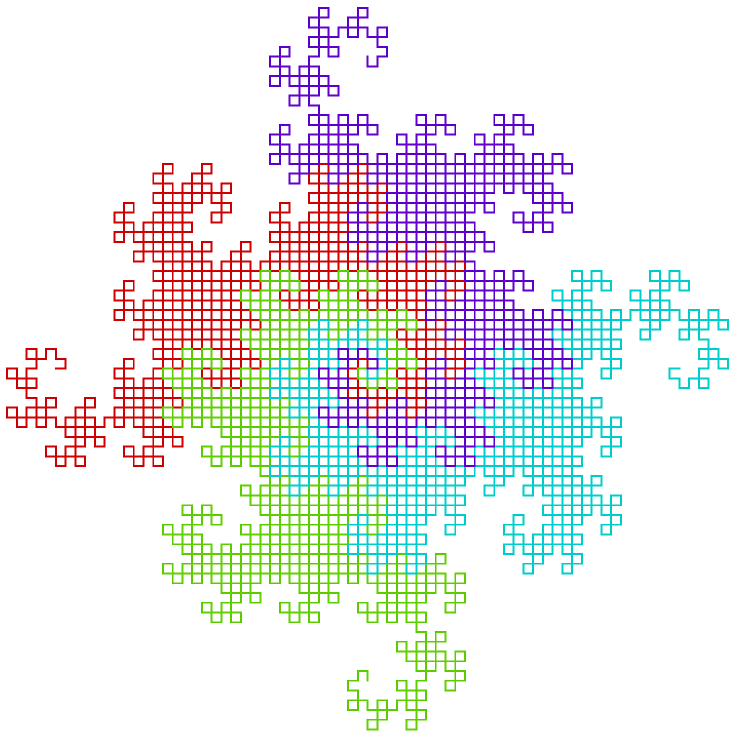


Figure 9.2: Four dragons filling the space produced by the script 9.1.

checking the value of the extra argument n which controls the number of times the recursive call will be applied.

Method 9.7

```
Turtle>>picRec: size n: n
   "Turtle clearWorld. self new north; picRec: 200 n: 1 ; beInvisible"

   | segSize |
   n = 1
      ifTrue: [self go: size]
      ifFalse:
         [segSize := size / 3.
         self picRec: segSize n: n - 1.
         self turnLeft: 45.
         self picRec: segSize n: n - 1.
         self turnRight: 90.
         self picRec: segSize n: n - 1.
         self turnLeft: 45.
         self picRec: segSize n: n - 1]
```

The dragon is a classic curve. It is a space filling self avoiding curve *i.e.*,that it fills space but without crossing its one path. We do not want to go into the details and let you discover its definition shown in the method 9.8. The Figure 9.2 results from the script 9.1.

Method 9.8

```
Turtle>>dragon: n
   "Turtle new dragon: 8"
   "Draw a dragon curve of order n"

   n = 0
      ifTrue: [self go: 5]
      ifFalse:
         [n > 0
            ifTrue:
               [self dragon: n - 1;
                  turn: 90;
                  dragon: 1 - n]
            ifFalse:
               [self dragon: -1 - n;
                  turn: -90;
                  dragon: 1 + n]]
```

The definition of the dragon curve shows how complex it can be to prove that the non recursive part of the definition. Here the condition `n = 0` is not simply reached. Indeed when called n is positive and the first recursive call (`dragon: n-1`) decreases n which is going in the right direction. However the other recursive calls increase or negate the value of n. Still the recursive definition terminates. It should be noted that this curve definition is particularly complex.

Script 9.1 (*Creating the Figure 9.2)*

```
|colors|
Turtle clearWorld.
colors := Color wheel: 4.
1 to: 4 do: [:i |
    Turtle new
       penColor: (colors at: i) ;
       turn: 90*i; dragon: 10 ;
       beInvisible]
```

4 About Keeping Context

Now we propose you to define a method that draws tree, regularl and unrealistic trees but trees as shown in the Figure 9.3. With this problem we illustrate the fact sometimes we need a way to reset the context of execution before performing a recursive call. Let us start!

The principle is the following one: to draw a tree, we draw one segment then we turn and draw a smaller tree, turn again, and draw another smaller tree. We follow this principle for the number of levels expected. From this principle we define the method `plant:level:` as shown in method 9.9 where the level specifies the number of segments compose one branch (from the root to one leaf).

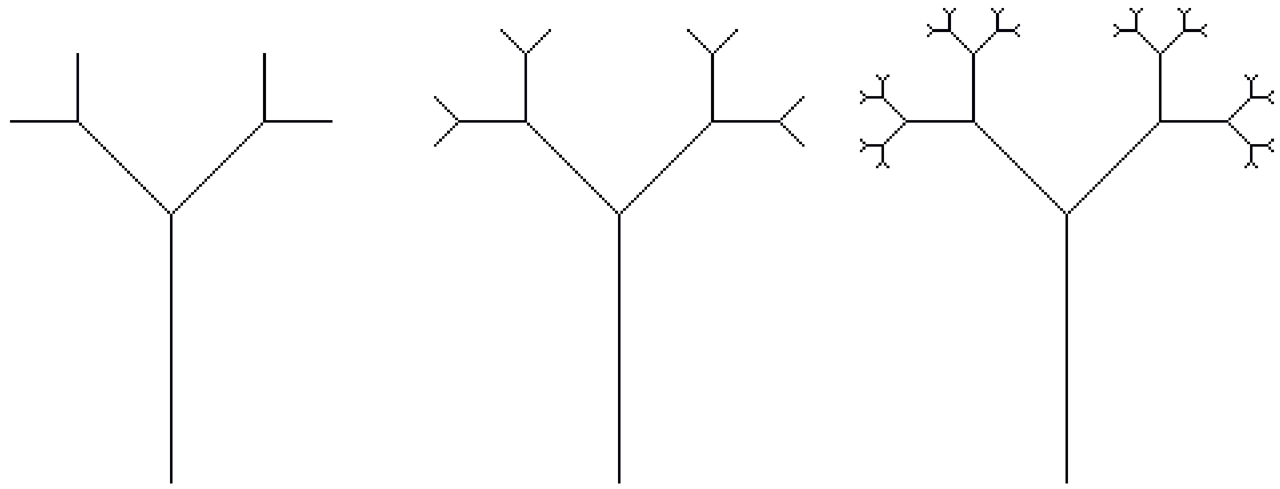


Figure 9.3: Some trees at different growing stages.

Method 9.9

```
Turtle>>tree: size level: n
   "Turtle clearWorld. Turtle new north; tree: 200 level: 1;
    beInvisible"

   n isZero
      ifFalse:
         [self go: size / 2.
         self turnLeft: 45.
         self tree: size / 2 level: n - 1.
         self turnRight: 90.
         self tree: size / 2 level: n - 1]
```

The definition of the method `plant:level:` answers the two questions we enonced for the method `picRec:`. There is a non-recursive part, here when the condition is true *i.e.*,n is equal to zero then nothing is executed and the method terminates. There is also a part of the method drawing the expression `self go: size /2`. Finally, the argument on which is built the condition is decreased so we are sure that at one point in time the condition will be true and that the recursion will stop. Executing the method with n = 1 works well. However, with n=2 we end up with the second drawing of the Figure 9.5 and not the third as we were expecting it.

Can you understand why? Step the execution of the message `Turtle new tree: 200 level: 2`. Have you noticed that the first branch is drawn correctly? Comment the second invocation to `tree:level:` to see it. What are the position and the direction of the turtle just before the second recursive call?

The problem with the current definition of `plant:level:` is that after the first recursive invocation, the turtle is not at right place ready to perform again the second recursion. Put a breakpoint to stop the execution as shown in the method 9.10 to obtain the same situation as the one illustrated in the Figure **??**. Imagine a solution to solve this problem.

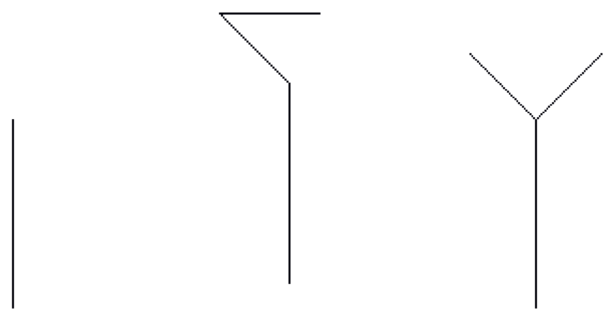
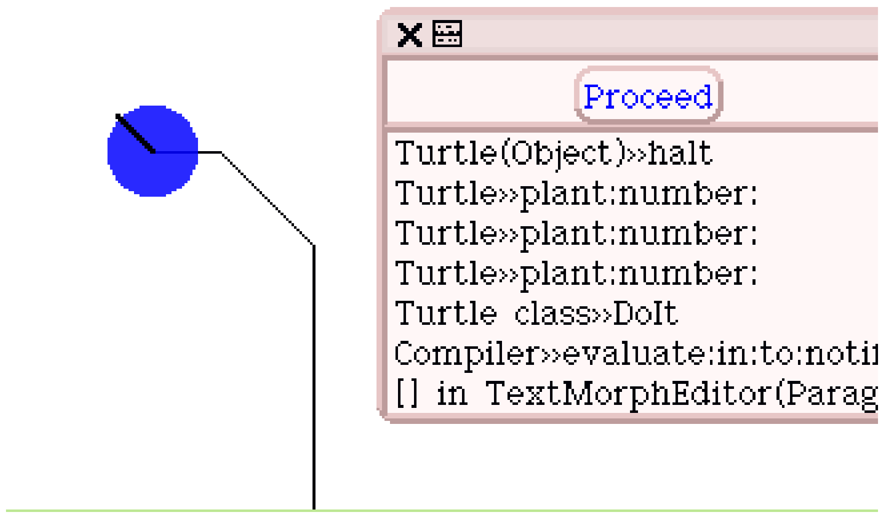


Figure 9.4: Trying to produce a tree.

Method 9.10

```
Turtle>>tree: size level: n
   "Turtle clearWorld. Turtle new north; tree: 200 level: 1;
    beInvisible"

   n isZero
      ifFalse:
         [self go: size / 2.
         self turnLeft: 45.
         self tree: size / 2 level: n - 1.
         *self halt.*
         self turnRight: 90.
         self tree: size / 2 level: n - 1]
```



```
Turtle(Object)>>halt
Turtle>>plant:number:
Turtle>>plant:number:
Turtle>>plant:number:
Turtle class>>DoIt
Compiler>>evaluate:in:to:noti
[] in TextMorphEditor(Parag
```

Figure 9.5: Stopping the method execution just after the first recursive invocation as described in the method 9.10.

This is a bit vicious. We cannot *simply* move back the turtle at its starting position after the first call is terminated because it depends on the number of levels it already drew. Still there is one solution which is to store the position of the turtle and its direction before invoking the method and restoring this state after. This is what the method 9.11 does. The variables `pos` and `dir` for every call ensures that the turtle is in the right position and heading to draw the second tree.

Method 9.11

```
Turtle>>tree: size level: n
   "Turtle clearWorld. Turtle new north; tree: 200 level: 3;
   beInvisible "

   | pos dir |
   n isZero
      ifFalse:
         [self go: size / 2.
         self turnLeft: 45.
         pos := self center.
         dir := self direction.
         self tree: size / 2 level: n - 1.
         self center: pos.
         self direction: dir.
         self turnRight: 90.
         self tree: size / 2 level: n - 1]
```

5 A more Traditional View on Recursion

Recursion has been a way to define suite of numbers for centuries. It is interesting to look at them, as they present a simple context to understand how a result is defined out of results obtained from recursive calls on smaller entities. We have a look at factorial and fibbonacci.

5.1 Factorial

Factorial is a classical mathematical function to present recursion. The mathematical definition of factorial is $!n = n*!(n-1), !0 = 1$. We consider that for negative numbers, !n = 1, even if the mathematical function does not allow this for simplicity.

Method 9.12

```
Integer>>factorial
   "Answer the factorial of the receiver."

   ^ self <= 0
      ifTrue: [1]
      ifFalse: [self * (self - 1) factorial]
```

What is interesting with this definition is that the result of the factorial is *composed of* the factorial of a smaller elements. In particular, the computing the final result requires a sub computation to be computed

1. the method is invoked,

2. the condition is executed,

3. when the condition value is true, the method terminates and returns 1.

4. when the condition is false, the conditional messages are executed one by one, `self -1` is executed, to its result factorial is again invoked, which is similar to the point 1. The part `self *` is "waiting" so that the sub call terminates by returning a value.

```
5 factorial
5 * 4 factorial
5 * 4 * 3 factorial
5 * 4 * 3 * 2 factorial
5 * 4 * 3 * 2 * 1 factorial
5 * 4 * 3 * 2 * 1 * 0 factorial
5 * 4 * 3 * 2 * 1 * 1
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120
```

Compare the first definition of `factorial` with the following one and make sure you understand their differences.

Method 9.13

```
Integer>>factorial
   "Answer the factorial of the receiver."

   self = 0 ifTrue: [^ 1].
   self > 0 ifTrue: [^ self * (self - 1) factorial].
   self error: ’Not valid for negative integers’
```

Factorial is a really simple function that can also be defined using a simple loops. Try to define it without using recursion.

5.2 Fibbonacci

Another well-know suite is the Fibbonacci suite. This suite is present in the plant or shell growth. Here it shows that a recursive method does not have to only call once itself but can do it several times as the dragon curve and the plant drawings where doing it.

The fibbonacci suite is defined as follow: $fib(0) = 1, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)$ which can be directly defined as shown by the method 9.14

Method 9.14

```
Integer>>fib

   ^ self < 2
      ifTrue: [1]
      ifFalse: [(self - 1) fib + (self - 2) fib]
```

The following trace shows how the value of the expression `5 fib` is computed, each recursive part is computed by summing two recursive sub computations which eventually ends on the non-recursive part $fib(1) and fib(0)$.

```
5 fib
= (4 fib + 3 fib)
= ((3 fib + 2 fib) + (2 fib + 1 fib))
= (((2 fib + 1 fib) + (1 fib + 0 fib)) + ((1 fib + 0 fib)  + 1)
= ((((1 fib + 0 fib)  + 1) + (1 + 1)) + ((1 + 1) + 1)
= ((((1 + 1) + 1) + 2) + (2 + 1))
= (((2 + 1) + 2) + 3)
= ((3 + 2) + 3)
= (5 + 3)
= 8
```

6 A Subtle Difference

In this section we want to show you a subtle difference between different form of recursion. In a first reading you can skip this part that we consider advanced for this book.

Often the result of a recursive call is used in a computation as this is the case in factorial. For example for computing `5 factorial` we need to use the result of `4 factorial` which needs the result of `3 factorial` and so on until `0 factorial` simply returns 1. We marked in bold when the method returns a result.

5 factorial
5 * 4 factorial
5 * 4 * 3 factorial
5 * 4 * 3 * 2 factorial
5 * 4 * 3 * 2 * 1 factorial
5 * 4 * 3 * 2 * 1 * 0 factorial
5 * 4 * 3 * 2 * 1 * **1**
5 * 4 * 3 * 2 * **1**
5 * 4 * 3 * **2**
5 * 4 * **6**
5 * **24**
120

Contrary to the other examples we show so far, this form of recursion requires to combine the results of the recursion to produce the end results. The idea is that the result on the parts should be composed to produce the result of the whole. When programming complex recursive function this composition is the key point because we need to combine partial result to produce the complete result. With factorial this is just a multiplication but it is often more complex.

While the computation is been performed, the memory of the computer is used to keep the rest of the computation. For example, the fact that `5 factorial` leads to compute `5 * 4 * 3 factorial` is kept in memory while `3 factorial` is computed. We say that the stack grows as we are adding new computation to be finished on the stack. When `3 factorial` terminates, the stacks of unfinished computation is popped. Once the computation of 3 factorial terminates, the `4 *` can be done and so on.

However recursive computation expressed this way are limited by the memory of the computer available. Quickly the computer memory can be filled up by all these computation suspended. However, the same recursive computation can be expressed differently without requiring to extra memory.

The idea is to use a parameter that acts as an accumulator in which intermediate results are passed from one invocation to the other. We define the method `fact` that invokes a new `factorial:` method that has an extra parameter.

Method 9.15

```
Integer>>fact
   "Answer the factorial of the receiver."

   self = 0
      ifTrue: [^ 1].
   ^ self factorialHelper: 1
```

Method 9.16

```
Integer>>factorial: res

   ^ self <= 0
      ifTrue: [ res]
      ifFalse: [self - 1 factorial: res * self]
```

The following trace shows how the computation evolves. Over times the argument contains the results of the intermediate computation and at the end is returned as result of the complete computation.

```
5 fact
5 factorial: 1
4 factorial: 5
3 factorial: 20
2 factorial: 60
1 factorial: 120
120
```

As the trace shows it there is no computation pending, all the computation is done during each recursive call. Hence there is no need for extra memory to keep the pending computation. This kind of recursion is called tail-recursive to denote the fact that there is no computation required after.

Do you guess the result of `5 fact` when we change the condition `self <= 0` by `self <0`.

To help you to understand how this works introduce a trace as follow.

Method 9.17

```
factorial: res

   self = 0
      ifTrue: [^ res].
   self > 0
      ifTrue: [Transcript show: self printString , ' res ' , res printString;
             cr.
         ^ self - 1 factorial: res * self]
```

7 Summary

recursive