Ruby The Smalltalk Way

by Huw Collingbourne
(Bitwise Courses)

http://bitwisecourses.com/

A short overview of the big ideas of the Smalltalk language and some similarities and differences between Smalltalk and Ruby.

Ruby The Smalltalk Way

written by
Huw Collingbourne

Second edition: May 2015

Introduction

This eBook has been written as part of my Advanced Ruby programming course. Unlike the rest of the course, this is not intended to be a step-by-step tutorial on Ruby programming. Instead it aims to try to give some historical perspective to Object Orientation in general and to Ruby's version of Object Orientation in particular. It does this by comparing various elements of the Ruby language with similar elements of the Smalltalk language. Smalltalk was the language which introduced Object Orientation to the broader programming world. All modern Object Oriented languages from Java to Ruby owe a huge debt to the pioneering work of Smalltalk.

Smalltalk is a 'pure' Object Oriented language and even though some ideas (such as inheritance and 'encapsulation') may already be familiar to you, I'm hoping that some of Smalltalk's ideas may challenge your view of Object Orientation – how it works, how it *should* work and how programmers use it in practice. In particular, if you have no previous experience of Smalltalk you may be surprised by some key ideas such as message-passing and black-box encapsulation. It is not my aim in this this little book to teach you how to program Smalltalk. Instead I'm hoping that it will give you a solid foundation in the ideas that led to the development of Smalltalk's Object Orientation which were later adapted (and, often very significantly changed) by other programming languages including Ruby.

There is a certain amount of programming theory in this book. It is not vital that you understand it all at first reading. This may at least provide you with food for thought and discussion. You may even be stimulated to explore Smalltalk in more depth. Personally, I think at least a basic knowledge of Smalltalk is useful to any programmer who uses an Object Oriented language. You may never use Smalltalk to create a finished application. But you may find that even a fairly superficial knowledge of Smalltalk will cause you to think about Objects, Classes and Methods in a different way when you go back to programming in another language such as Ruby.

MAKING SENSE OF THE TEXT

In **Ruby The Smalltalk Way**, any source code is written like this:

```
puts "hello world".upcase
```

Any output that you may expect to see on screen when a program is run is shown like this:

```
HELLO WORLD
```

When there is a sample program to accompany the code, the program name is shown in a little box like this:

example9.rb

When an important name or concept is introduced, it may be highlighted in the left margin like this:

FUNCTIONS

Explanatory notes (which generally provide some hints or give a more in-depth explanation of some point mentioned in the text) are shown in a shaded box like this:

> This is an explanatory note. You can skip it if you like – but if you do so, you may miss something of interest…!

Ruby and Smalltalk source code is provided in the downloadable code archive. The Smalltalk code is intended for use either by *Dolphin Smalltalk* for Windows (use the file **chapter1.st**) or by *Pharo* for Windows, OS X and other platforms (use the file **chapter1pharo.st**). It will not run 'as is' in other versions of Smalltalk. If you do not wish to use Dolphin Smalltalk or Pharo, you can still follow this tutorial by referring to the explanations in the text. *You do not need to program Smalltalk in order to follow the explanations given in this book.*

SMALLTALK DOWNLOADS

If you want to give Smalltalk a try, you can download a free Smalltalk IDE. A great sustem for is *Dolphin Smalltalk*. The free 'Community Edition' of Dolphin Smalltalk can be downloaded from the Object Arts web site, here:

http://www.object-arts.com/products/dce.html).

Alternatively, for cross-platform development you could use *Squeak Smalltalk* which is available for all major operating systems: http://www.squeak.org or *Pharo* http://pharo.org/ .

> The Smalltalk code in this tutorial was originally written for **Dolphin Smalltalk** for Windows. I have converted the code for use with **Pharo** and you may run the examples using either of these.

THE 'RUBY THE SMALLTALK WAY' COURSE TEXT

In this series, I won't assume any previous knowledge of Smalltalk. In order to give Smalltalk newcomers some background reading material and, moreover, to provide a structure for the series itself, I'll be constantly referring to a little tutorial which was written for a commercial Smalltalk product, *Smalltalk/V*. You can think of this as the 'course text' of the series. The *Smalltalk/V Tutorial* provides an easy introduction to the fundamental ideas of Smalltalk.

> DOWNLOAD THE SMALLTALK/V TUTORIAL
>
> The **Smalltalk/V Tutorial** is available for free download (along with a number of other fine Smalltalk books) from Stéphane Ducasse's site: http://stephane.ducasse.free.fr/FreeBooks.html.

Preface

I've lost count of the number of times I've heard people comparing Ruby and Smalltalk. In programming books, blogs, newsgroups and magazines articles, the close affinity with the two languages often seems to be taken for granted. But how close are they really...?

The first Smalltalk system I ever used was *Smalltalk/V* from a company called Digitalk. That was back in the late '80s and early '90s. Prior to that I had programmed mainly in Pascal with occasional forays into C, Modula-2 and Prolog. None of this had prepared me for Smalltalk.

OBJECTS EVERYWHERE

With its tightly integrated environment (all graphics, overlapping windows, class browsers, interactive program development and mouse-control) the *Smalltalk/V* IDE was totally unlike the text-based, keyboard-centric *write-compile-debug* tools I'd used previously.

The Smalltalk *language* was just as much of a shock to the system as the environment. It had no function calls, just '*message passing*'; no free-standing procedures, just tightly-bound *methods*; and no 'main' loop to define the entry point of a program. In the Smalltalk world all classes are equal and a program can, in theory, start anywhere it likes - just create an object of your choice and off you go!

Some of Smalltalk's big ideas have subsequently permeated into other languages in the programming mainstream; some, but not *all*... For me, it is certainly the case that programming in Java, say, or C# feels like a totally different activity from programming in Smalltalk.

So what about Ruby?

> The creator of Ruby, Yukihiro "Matz" Matsumoto, acknowledges the influence of a variety of languages on Ruby – including Perl, Python, LISP and Smalltalk. Here I shall concentrate on Ruby's Object Orientation which was inspired predominantly by Smalltalk (and possibly other OOP languages that were themselves inspired by Smalltalk).

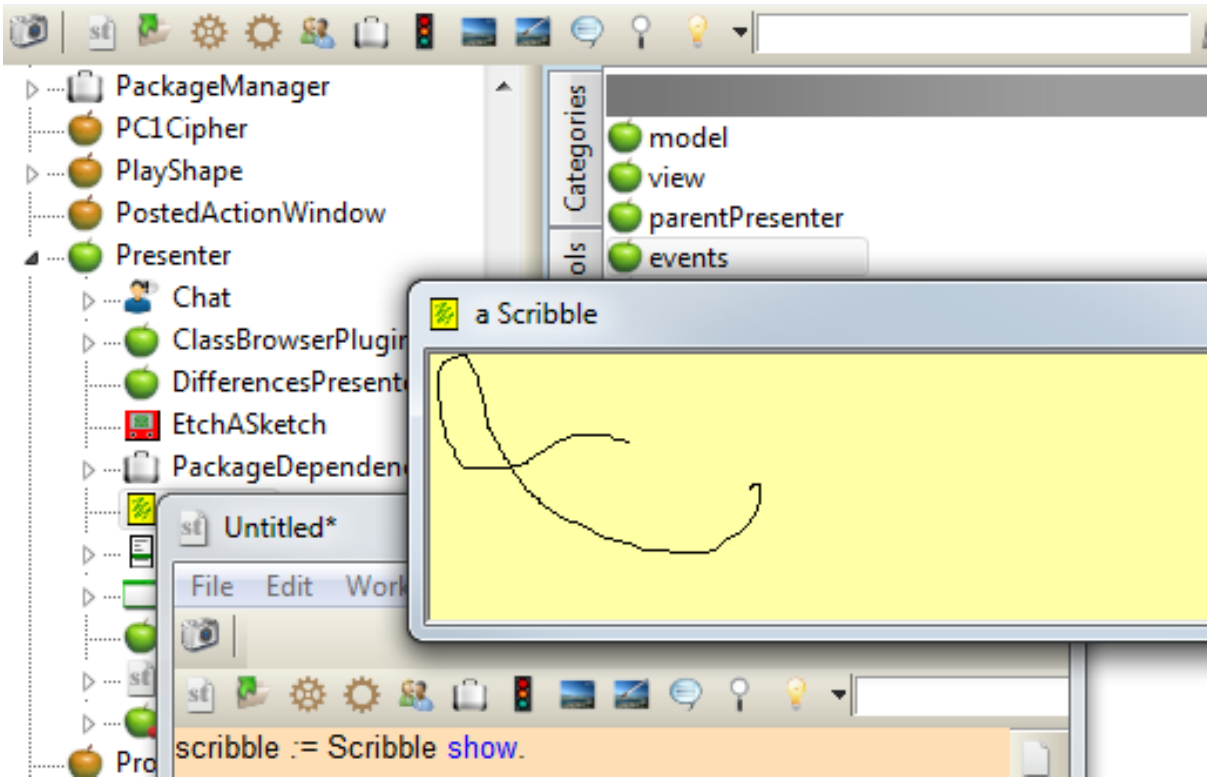
Is Ruby The Heir to Smalltalk?

There is undoubtedly something about Ruby that 'feels' different from languages such as Java, C#, C++ or Delphi. To some extent, that might be explained by the fact that C++ and Delphi began life as procedural languages (C and Pascal respectively) onto which was 'bolted' an additional layer of object orientation. The end result is a hybrid which has some procedural features and some Object Oriented features. But Ruby (just like Smalltalk) was built on objects from the ground up. So is Ruby truly the naturally successor to Smalltalk? And if so, is that necessarily a good thing?

In order to answer those questions, we'll need to take a close look at Smalltalk itself. If you are unfamiliar with Smalltalk, fear not, I'll be providing a very gentle introduction. You won't have to do any hands-on Smalltalk coding if you don't want to (though if you are feeling adventurous, you might want to give it a try - Smalltalk really is a lot of fun!)

The Programming Environment

Before looking at the similarities between Smalltalk and Ruby, I need to mention one huge difference – the programming environment. In Smalltalk, the language and its Integrated Development Environment (IDE) are so tightly bound that the IDE itself (its windows and graphics) can be manipulated by executing Smalltalk code. Ruby, on the other hand, presupposes no environment and, even if you decide to use a dedicated Ruby IDE, the Ruby language has, so to speak, no 'built in knowledge' of its workspace.

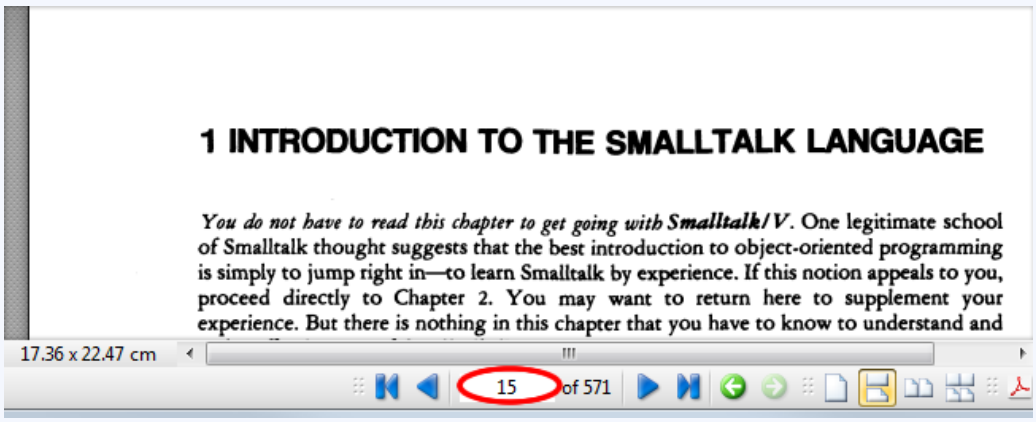


Here I have evaluated some program code (bottom window) to show a new 'Scribble' (drawing) window in the Dolphin Smalltalk IDE.

For a quick guide to using Dolphin Smalltalk (for Windows) see my article, Learn Dolphin Smalltalk. For an introductory guide to Squeak Smalltalk (cross-platform), see Smalltalk: A Beginner's Guide. Pharo Smalltalk comes with extensive documentation. See the list of Smalltalk tutorials at the end of the eBook.

From Smalltalk to Ruby

In this lesson, we'll take a look at some of the fundamental features of Smalltalk's Object Orientation Programming (OOP) features and see how they relate to Ruby. This lesson will be a fairly lightweight introduction to some of the ideas of Smalltalk and Ruby's OOP and some basic syntax of those two languages.

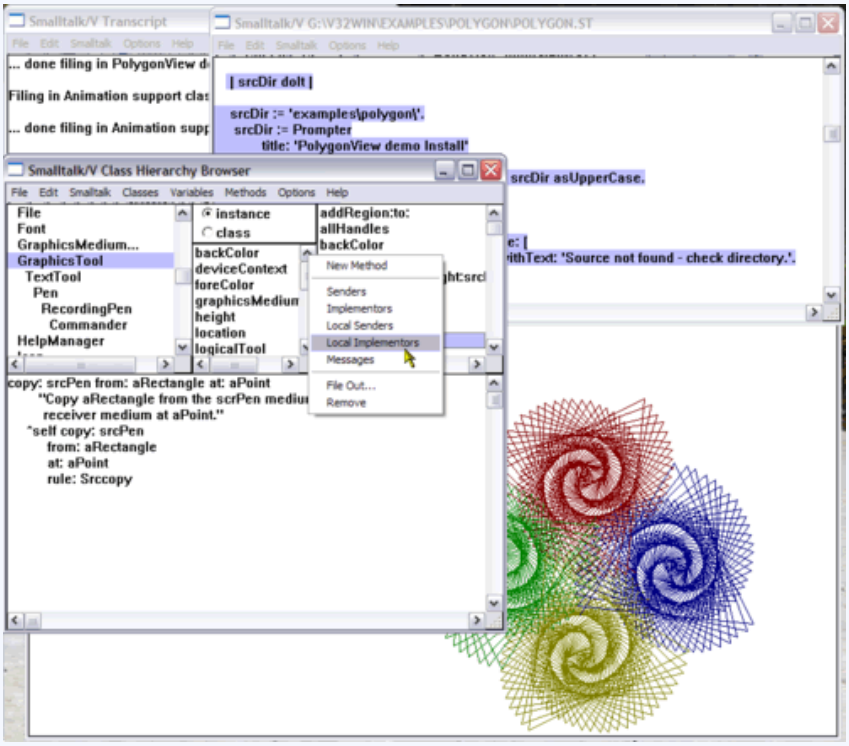


This chapter's background reading is the first part of *Chapter One* of the *Smalltalk/V Tutorial*. In the PDF download, that's pages 15 to 18 (as shown in the bar at the bottom of the Acrobat Reader), ending with the section named '*A Function Call With Three Arguments*'.

SMALLTALK'S BIG IDEAS

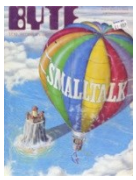
To understand just how revolutionary the ideas of Smalltalk were, you have to try to put yourself back into the period when it was developed, in the 1970s. This was at a time when computer screens showed nothing more than green text on a black background and mice were things that lived in skirting-boards and ate cheese. Computer programming, in those days, was a decidedly non-interactive pastime that involved editing-compiling-and-linking (if you were lucky) or punching holes into cards and feeding them into slots (if you weren't). Most programming languages and environments had no 'visual design' capabilities, very little if anything in the way of debugging and certainly no 'objects'.

And then along comes Smalltalk - with its slick graphic environment, its ability to 'mark and execute' code instantly, its overlapping windows, bitmapped fonts, context sensitive menus, mouse control and, as an added bonus, a thing called Object Orientation (to name just a *few* of its big ideas). The rest of the computing world took another decade even to *begin* to catch up. And today, almost four decades later, many of Smalltalk's ideas still seem cutting edge.



This is Smalltalk/V as it was in the early '90s. Not only was it ahead of the game at that time - what is even more remarkable is that the essential features of its graphic, interactive environment and object orientation were developed as far back as the 1970s...

THE SMALLTALK REVOLUTION



Smalltalk was developed by a team based at the Xerox Learning Research Group at the Palo Alto Research Centre (*Xerox PARC*) which, throughout the 1970s, released incremental versions of the language and development environment: Smalltalk-72, Smalltalk-74, Smalltalk-76 and Smalltalk-80. This last version, *Smalltalk-80*, might be regarded as the standard version from which subsequent implementations have been developed. The first I, and most other people, heard of this was when the magazine *Byte* devoted a special issue to Smalltalk way back in August 1981.

To give you some idea of just how revolutionary Smalltalk was, the *Byte* editorial even had to explain what a 'mouse' was - (I quote: "*The 'mouse' [is] a small mechanical box with wheels that lets you quickly move the cursor around the screen*").

There was one other interesting little thing about Smalltalk. Objects. While Smalltalk wasn't the first language to use Object Orientation (the Simula language can probably claim that) it was the first language to make a major impression with OOP.

Smalltalk was largely the brainchild of Alan Kay who conceived it as part of a hand-held networked computer system called the *Dynabook*. This was an audacious

concept at the time. You have to remember that portable computers back in those days often required two strong men to lift them. A hand-held computer with built-in networking and high resolution graphics seemed, to many of us, a pipedream which would never be seen in our lifetime. Now, of course, we have all this and more on our phones and tablets.

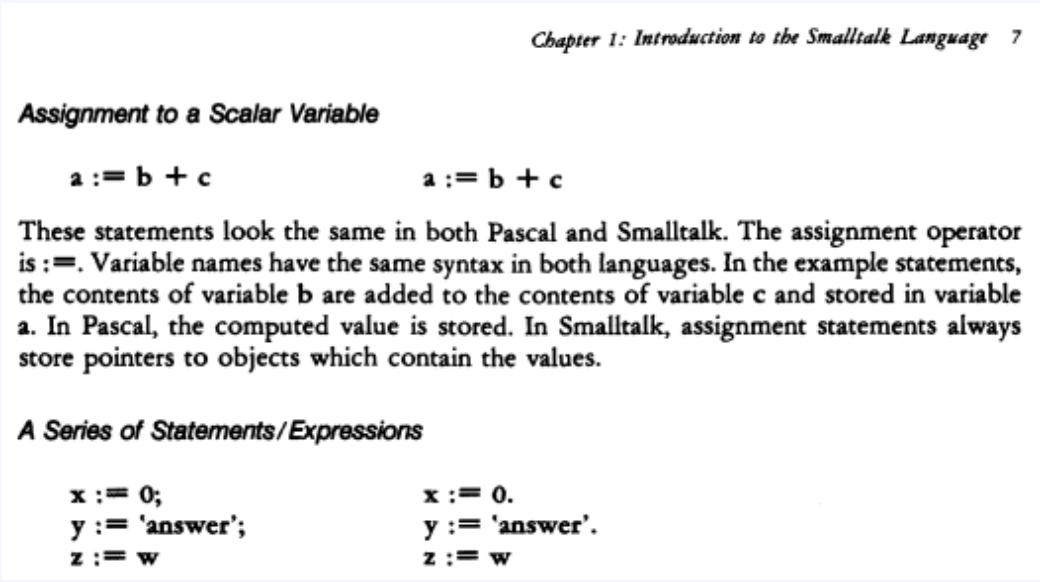
The first, and arguably the most important, influence of Smalltalk on mainstream computing was its graphical user interface. Steve Jobs acknowledged that the Mac interface was inspired by Smalltalk. This, in turn, undoubtedly inspired other graphical interfaces including Windows.

That alone would have been sufficient to ensure Smalltalk's claim to be a hugely important milestone in the history of computing. Given the fact that it also popularised object orientation and integrated development environments, you might say that Smalltalk represents not one but two milestones.

RUBY AND SMALLTALK VS. OTHER LANGUAGES

In this section, we look at some basic syntax. The *Smalltalk/V Tutorial* compares the syntax of the Smalltalk language with that of Pascal. Bear in mind that, at the time that tutorial was written (in the late '80s) OOP was not widely used or understood and so the tutorial had to explain ideas which at the time were extremely unusual but which are now commonplace. Even so, if you've never used Smalltalk, you may discover that other languages do not implement OOP in quite the same way as Smalltalk, so it's worth taking a little time to consider even the basics . In this section, I will make comparisons between Smalltalk, C# and Ruby. Let's first look at some simple assignment statements.

THE SMALLTALK/V TUTORIAL



Chapter 1: Introduction to the Smalltalk Language 7

Assignment to a Scalar Variable

```
a := b + c                    a := b + c
```

These statements look the same in both Pascal and Smalltalk. The assignment operator is :=. Variable names have the same syntax in both languages. In the example statements, the contents of variable **b** are added to the contents of variable **c** and stored in variable **a**. In Pascal, the computed value is stored. In Smalltalk, assignment statements always store pointers to objects which contain the values.

A Series of Statements/Expressions

```
x := 0;                       x := 0.
y := 'answer';                y := 'answer'.
z := w                        z := w
```

Be sure to refer to Chapter One the *Smalltalk/V Tutorial* when reading this eBook. I have used many of the same section headers such as '*Assignment To A Scalar Variable*' and '*A Series Of Statements/Expressions*' to help you compare my discussion of Ruby with the *Smalltalk/V Tutorial*'s discussion of 'pure' Smalltalk.

ASSIGNMENT TO A SCALAR VARIABLE

C#

```
b = 1;
c = 2;
a = b + c;
```

Smalltalk

```
b := 1.
c := 2.
a := b + c
```

Ruby

```
b = 1
c = 2
a = b + c
```

ASSIGNMENT OPERATOR

The assignment operator in Smalltalk is the same as in Pascal (`:=`). In Ruby, it is the same as in C# (`=`). In C# the statement separator is a semicolon; in Smalltalk it is a period. In Ruby a linefeed is sufficient to separate statements though a semicolon may also be used and is required when separating multiple statements on a single line, like this: `b = 1; c = 2; a = b + c`

A SERIES OF STATEMENTS/EXPRESSIONS

example2.rb

C#

```
x = 0;
y = "answer";
w = "hello";
z  = w;
```

Smalltalk

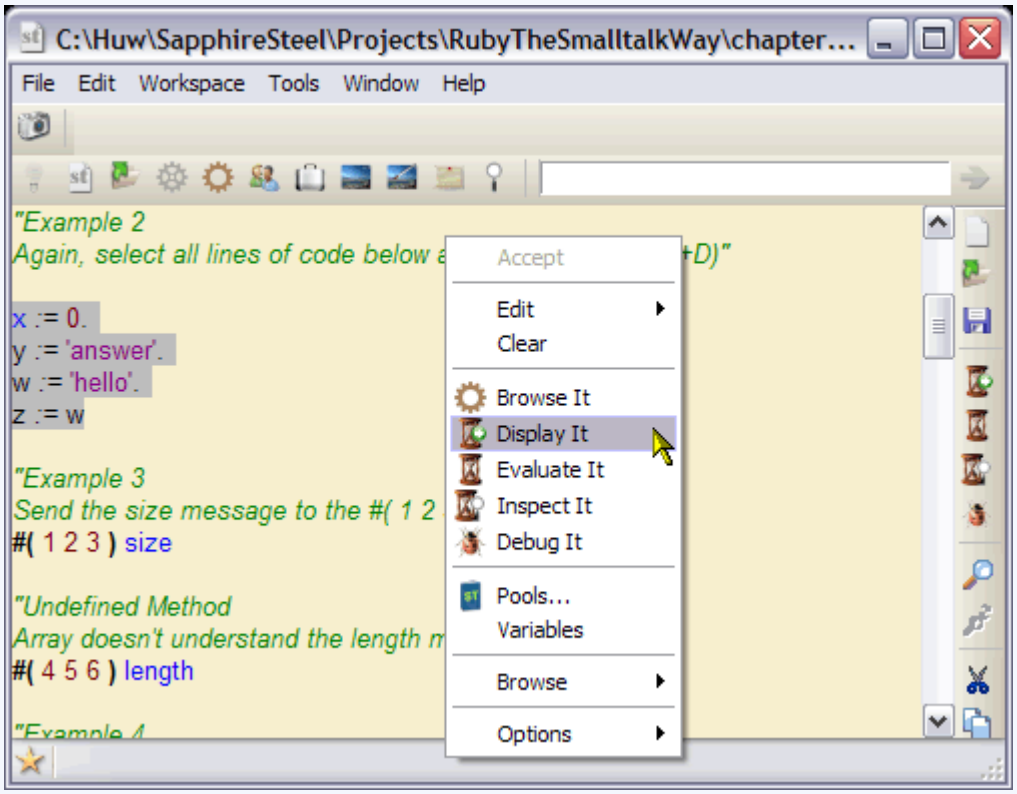
```
x := 0.
y := 'answer'.
w := 'hello'.
z := w
```

Ruby

```
x = 0
y = 'answer'
w = "hello"
z = w
```

Smalltalk/ chapter1.st

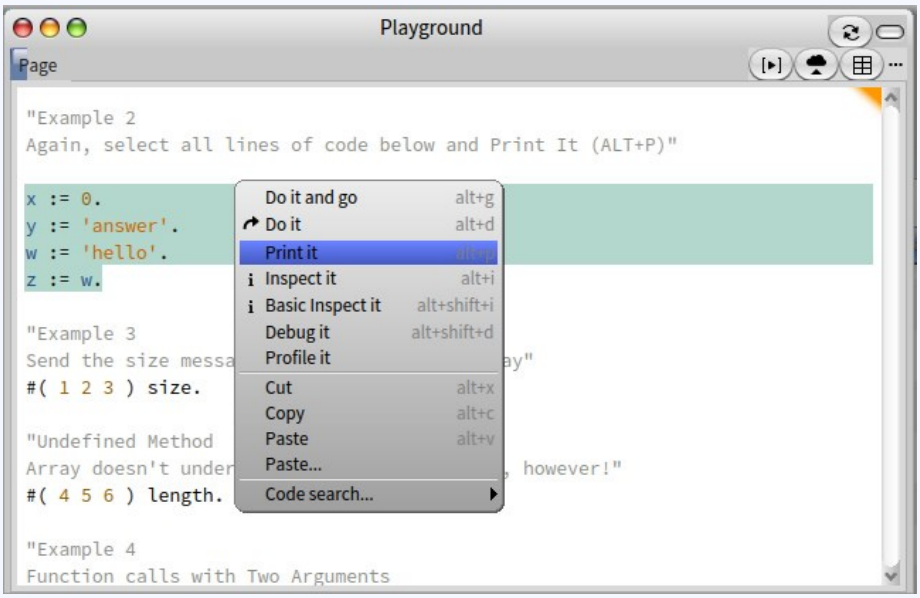
DOLPHIN SMALLTALK



To use the sample code using **Dolphin Smalltalk**, open the file **chapter1.st** in a Workspace window (if necessary, double-click the Workspace icon in the main command window and then select *File, Open*). To evaluate code, select it with the mouse and press *CTRL+E*. To evaluate and display the result, select the code and press *CTRL+D*. You can also right-click and select 'Evaluate It' or 'Display It' from a menu.

Smalltalk/ chapter1pharo.st

PHARO



To load the code into **Pharo**, open a *Playground* (left-click in the main Pharo Workspace and select *Playground* from the popup menu). Now load **chapter1pharo.st** into a text editor such as Windows Notepad, select all the text it all and copy it. Then paste it into the *Playground* window. To evaluate code, select it with the mouse and press *ALT+D* (or right-click and select *Do It* from the menu). To evaluate and display the result, select the code and press *ALT+P* (or right-click and select *Print It* from the menu).

Note that Smalltalk strings are delimited by single-quotes, C# strings are delimited by double-quotes and Ruby strings may use either single or double quotes. Ruby single-quoted strings are treated literally but double-quoted strings evaluate expressions enclosed inside `#{` and `}`. So, if you run the following little program…

ruby_strings.rb

```
puts( '#{[1,2,3].length}' )
puts( "#{[1,2,3].length}" )
```

…the single-quoted string will be displayed literally but the double-quoted string will be evaluated, resulting in this output:

```
#{[1,2,3].length}

3
```

A Function Call With One Argument

In procedural languages such as C and Pascal, when you need to do something with a piece of data it is normal to pass that data as an argument to a function. For example, to find the size of an array in Pascal, you might pass an array variable to the `size()` function:

```
a := size(array);
```

Here, the `size()` function returns the size of the array and that value is assigned to the variable `a`. In Smalltalk, on the other hand, you 'send a message' called `size` to the array object itself:

```
a := array size
```

Message sending

The idea of 'message sending' rather than 'function calling' is fundamental to Smalltalk. Essentially, a message is a 'request' to an object. When you send to an object a message called `size`, that object takes a look to see if it has any way (literally any 'method') of responding to that message. In this case, the array has a method with the same name as the message (i.e. `size`) which means that it is able to respond.

> "The big idea is 'messaging' - that is what the kernel of Smalltalk/Squeak is all about."
> (Alan Kay)

The central idea of providing objects with 'methods' has now been adopted by most OOP languages, including Java, C# and Ruby, though the convention of simply placing the method name after a space (as in Smalltalk) has not been adopted. Instead, dot-notation is the norm:

example3.rb

C#

```
a = array.Length;
```

Ruby

```
a = array.length
```

Note, that C# pedants might complain that in the above example, `Length` is a 'property' rather than a method. For our purposes, that really doesn't matter.

However, for the sake of completeness, here's another example, using a slightly more verbose method:

C#

```
a = array.GetLength(0);
```

An interesting feature of Smalltalk's 'message-sending' mechanism is that you are free to send any message to any object. If the object hasn't got an appropriate method of dealing with a message, Smalltalk will inform you that the object does not understand the message and you will have the option to carry on anyhow or to debug the problem. If you try the same thing in C#, your program will refuse to compile. If you do it in Ruby, your program will grind to a halt with a '*NoMethodError*'. Ruby can, in fact, be a little more forgiving. It provides an `undefined_method` method. All you have to do is to implement that method in order to deal with any messages which may be sent to an object which has no appropriate method with which to respond. Try out the Ruby sample files *undefined_method1.rb* (which, since the `abc` method does not exist, results in an unrecoverable error)...

undefined_method1.rb

```
array = [1,2,3]
array.abc
```

Now run the code in *undefined_method2.rb* which produces a nicely formatted "Cannot find method" error message and carries on running anyhow..

undefined_method2.rb

```
def method_missing(method, *args)
   if args.length == 0 then
      puts( "Cannot find a method named '#{method}'")
   else
      print( "Cannot find a method named #{method}" )
      puts( " with #{args.length} argument(s)" )
   end
end
array = [1,2,3]
array.abc
array.xyz( 1, 2, 3 )
print( "Array length is: #{array.length}" )
```

"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."
(Alan Kay)

The rest of Chapter One in the *Smalltalk/V Tutorial* is concerned with providing examples of Smalltalk syntax. I don't propose to repeat all that information here. There are a few things worth noting, though. In particular, you will see that while Smalltalk syntax may look reasonably familiar when calling a function with one argument, things change when it calls functions with multiple arguments.

This may not be obvious when sending a message (or 'calling a method') such as `+` since this seems to behave just like an operator in any other language:

```
y := p + q
```

But how about this?

```
x := x1 max: x2.
```

This is an example of why 'message passing' matters in Smalltalk and how it is different from 'function calling'. Let's look at this closely. We'll start at the right-hand side of the expression and work back towards the left. Here `x2` is an object whose value is passed to `max:`. It turns out that `max:` is a method of the numeric object `x1` (in Dolphin Smalltalk, it is actually a method of the `Magnitude` class from which various types of number such as `Float` and `Integer` descend). This is Dolphin's definition of `max:`

```
max: operand
   ^self < operand
      ifTrue:  [operand]
      ifFalse: [self]
```

I don't want to dwell on the finer details of Smalltalk syntax for the time being. Instead, let's see what this method is actually doing. Essentially it performs a comparison between the value of an argument (here named `operand`) and the receiver object `self`. By 'receiver object' I mean the object to which this method belongs - in the example above, that is `x1`. In English, the workings of the `max:` method can be expressed thus: *if the value of the receiver object (`self`) is less than the value of some other object (`operand`) then return the operand otherwise return the receiver object* (in Smalltalk the `^` symbol means 'return').

Now, Ruby numbers don't have a `max:` method, so how can I 'translate' this Smalltalk method into Ruby? There are a few possible ways. On glancing through the ruby class library I note that enumerable objects, such as arrays, have a `max` method. So, if I put the numbers into an array, I can do this...

example4.rb

```
x1 = [100,200]
puts( x1.max )
```

That's a bit of a cheat though. What I really want is a `max` method in the `Numeric` class. Ruby lets me extend existing classes so I can easily add such a method:

```
class Numeric
   def max( aNum )
      if self <= aNum
         return aNum
      else
         return self
      end
   end
end
```

Now I can write something that looks much closer to the Smalltalk version...

```
x1.max( x2 )
```

Though, having done so, I am left wondering whether the effort was worthwhile. After all, it would be easy to achieve exactly the same results with a simple `if..else` test:

```
if x1 > x2
   puts( x1 )
else
   puts( x2 )
end
```

The above Ruby code is very similar to code you could write an procedural languages such as Basic, Pascal or C. It is most unlike Smalltalk, however. Smalltalk doesn't do `if..else` tests. The reason is simple - tests like that don't form a natural part of 'message sending'. If `if` is a message, then to which object is it being sent? There isn't one.

IS RUBY LESS STRICT THAN SMALLTALK?

Here then, is one simple example of a very un-Smalltalk-like feature of Ruby. Smalltalk is quite strict in its message-passing methodology; but Ruby makes concessions to convention. Most programmers simply *expect* to be able to use `if` and `else` so Ruby provides them. You might say that Smalltalk is more *rigorous* but Ruby is more *approachable*. This is, I suspect, one reason why programmers often find it much easier to get to grips with Ruby than with Smalltalk.

If the lack of `if` tests in Smalltalk strikes you as bizarre, I should explain that these tests *can* be done, it's just the *way* in which they are done that is different from other languages. Look again at Dolphin Smalltalk's code for the `max:` method:

```
max: operand
   ^self < operand
      ifTrue:  [operand]
      ifFalse: [self]
```

Here the comparison `self < operand` yields a result - which will be either true or false; that is, it will be either a `True` object or a `False` object. The `True` and `False` classes are defined in the Smalltalk class library and they have methods called `ifTrue:` and `ifFalse:`

A Function Call with Three Arguments

Smalltalk messages and methods can, in fact, be 'broken up' into several pieces, with arguments (other objects) placed between the pieces. The *Smalltalk/V Tutorial* gives this example:

```
b := x between: x1 and: x2.
```

Here `between:and:` is the message. Once again, in Dolphin Smalltalk, this method belongs to the `Magnitude` class and this is its implementation:

```
between: min and: max
   ^self >= min and: [self <= max]
```

This returns true if the value of the receiver object (`self`) is greater than or equal to the value of the first argument (`min`) and less than or equal to the value of the second argument (`max`); otherwise it returns false. In most other languages, including Ruby, method names have just *one* part (they can't be divided up into separate bits like `between:` and `and:`) and any arguments to a method are sent as a list, as in this Ruby example:

example5.rb

```
b = x.between?( x1, x2 )
```

So, already a number of *similarities* are emerging between Smalltalk and Ruby (for example, they both have a fairly simple, 'human readable' syntax and a broadly similar approach to OOP); but we are also finding some significant *differences* - particularly in their approach to message *passing* (the Smalltalk way) and method-*calling* (Ruby's more 'conventional' way of working).

A Question of Style

In this chapter I want to consider whether it is possible (or, indeed, desirable) to program Ruby in a 'Smalltalk style'.

> Background reading for this chapter is the second part of Chapter One of the *Smalltalk/V Tutorial* - in the PDF version, that's pages 18 to 21 (as shown in the bar at the bottom of the Acrobat Reader) , ending with the section named '*The World According To Objects*'.

SUBSCRIPTED VARIABLE ACCESS

In Smalltalk, if `a` is an array and `i` is an integer, you can evaluate the expression `a at: i` to retrieve the value stored at the array index `i`. Let's say that value is 100 - in Smalltalk terms, 100 is an integer object (or some specific type of integer such as `SmallInteger`). You can now pass to this integer object the message `* 2` in order to multiply its value by two. This yields another integer object, 200, as a result. Now you can pass this new integer object (200) to the `at:put:` method (and so on). Here's a complete example:

example6.rb

```
a := Array new: 26.
i := 2.
y := 100.
a at: i put: y.
a at: i + 1 put: (a at: i ) * 2.
```

The end result is that the array `a` contains the following:

```
#(nil 100 200 nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil)
```

This would be more naturally written in Ruby using a Pascal-like syntax (compare with the Pascal code in the *Smalltalk/V Tutorial*):

```
a = Array.new( 26 )
i = 2
y = 100
a[i] = y
a[i+1] = a[i]*2
```

> THE CODE ARCHIVE
>
> Remember that all the code is supplied in the download archive 'ready to run' for Dolphin Smalltalk and Pharo. In most cases you may also run the code in other Smalltalk systems though compatibility is not guaranteed. To try out the code just select it and press CTRL+D (Dolphin) or ALT-P (Pharo). Or right-click and pick *Display It* or *Print It* from the menu. Ruby 'translations' of the Smalltalk code are also supplied in the archive.

IF STATEMENTS

I mentioned Smalltalk's lack of a conventional `if` statement in Chapter 1. When you need to take conditional action in Smalltalk, you can evaluate an expression which returns a true or false object to which messages such as `ifTrue:ifFalse:` may be sent. The code to execute when either a true or a false value is returned is enclosed within a block delimited by square brackets:

example7.rb

Smalltalk

```
a < b
   ifTrue: [^'a is less than b']
   ifFalse: [^'a is greater than or equal to b']
```

Ruby

```
if a < b then
   puts "a is less than b"
else
   puts "a is greater than or equal to b"
end
```

ITERATIVE STATEMENTS

Here are two examples showing one way of iterating through an array of 10 digits and adding them together. Note that Smalltalk arrays (usually) start at index 1, Ruby's start at 0 so I've made adjustments to allow for that. As with `if` tests, when Smalltalk runs through a `while` loop, it evaluates an expression (here `[i < 1]`) which yields an object (an instance of the `True` or `False` class) and then sends a message to that object `whileTrue:`

Smalltalk

```
a := #(1 2 3 4 5 6 7 8 9).
i := 1.
sum := 0.
[ i < 10]
   whileTrue: [sum := sum + (a at: i).
      i := i + 1].
```

Ruby, on the other hand, runs its `while` loop in the same way as procedural languages. Both `if` and `while` are keywords rather than methods - so Ruby once again sacrifices the strict 'purity' of the OOP message-sending paradigm in order to gain the benefit of familiarity for the vast majority of programmers who have moved to Ruby from some other language such as C, Java or Pascal:

See: example8.rb

Ruby

```
a = [1,2,3,4,5,6,7,8,9]
i = 0
sum = 0
while( i < 9 )
   sum += a[i]
   i += 1
end
```

RETURNING FUNCTION RESULTS

In *Smalltalk*, the value returned by a method is indicated by preceding it with the `^` character:

```
^answer
```

In *Ruby*, the value returned may be indicated by preceding it with the keyword, `return`:

```
return answer
```

If there is no explicit `return`, Ruby returns the last expression evaluated:

```
def x
   "hello world".upcase
end
```

The `x` method above returns: "HELLO WORLD".

STORAGE ALLOCATION AND DE-ALLOCATION

Just like Smalltalk, Ruby is a *garbage-collecting* language. That means that, while you may have to allocate memory for new objects, you don't need to deallocate it later on. When objects are no longer referenced they are automatically marked for garbage collection. From time to time, Ruby looks for objects that are no longer needed and frees up their memory. As with Smalltalk, you allocate memory by calling the `new` method. To create an array of 5 empty 'slots' (each containing `nil`):

example9.rb

Smalltalk

```
a := Array new: 5
```

Ruby

```
a = Array.new(5)
```

A Complete Program

To round off this section, you might care to glance at the example in the *Smalltalk/V Tutorial* of a character-frequency counter. I have to say that I have never found this program either very interesting or illuminating. The idea seems to be to show how verbose a procedural language (such as Pascal) is and how succinct Smalltalk is by comparison. There are two problems with this, however. First, the example is unfairly weighted to take advantage of built-in features of the Smalltalk class library and, secondly, the 'Smalltalk-style' version of the code (page 11 in the printed book or 21 in the page bar of Acrobat) is not "the same program" as the Pascal one (as it is claimed to be in the text). The first (Pascal-style) program displays a count of each letter in a 26-slot array; the second (Smalltalk-style) program merely adds each letter to an unordered collection called a Bag.

For completeness, I've slightly adapted the *Pascal-style* version for use with Dolphin and Pharo. I don't think that program is particularly relevant to my discussion, however, so let's move on to the *Smalltalk-style* version. This at least has a few points of interest since it highlights a few of the differences between Smalltalk and Ruby, which I'll talk about in more detail later on. Here is the code:

Smalltalk

```
| s f |
s := Prompter prompt: 'enter line'.
f := Bag new.
s do: [ :c | c isLetter ifTrue: [f add: c asLowercase] ].
^f
```

> If you are using Pharo, instead of creating a `Prompter` (which does not exist in the Pharo class library) use the `UIManager` like this:
> `s := UIManager default request: enter line'.`

This prompts the user for a string, then creates a new `Bag` (collection) and iterates through each character in the string, adding its lowercase version to the Bag only if the character is alphabetic. So if the string were "Hello 123 World", the Bag, `f`, would end up containing these characters (Smalltalk characters are prefixed by a `$`). Note that a Bag is unordered so the characters may not appear in the order shown here:

```
($o $o $e $d $r $h $l $l $l $w)
```

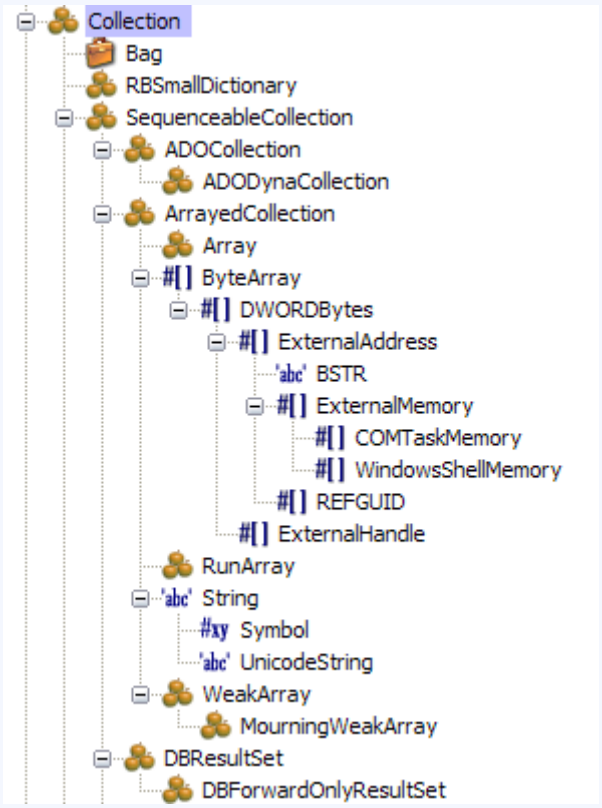
Here is an approximate Ruby equivalent:

example10.rb

```
print( "enter line: " )
s = gets().to_chars
a = Array.new
s.each do |c| if( c.isLetter ) then a << c.downcase end end
```

At first sight this looks pretty similar to the Smalltalk version. However, there are quite a few differences. First of all, Ruby doesn't have a `Bag` class. In fact, Ruby has rather few collection classes - and for most straightforward lists, the `Array` class is used. Smalltalk, on the other hand, has a great variety of collection classes. If you are using Dolphin Smalltalk, open the class browser (from the *Tools* menu). Now, in the top-left hand pane, scroll down to `Collection` and click the + signs to open up the branches. Here you will see a hierarchy of increasingly specialised collections:

SMALLTALK'S RICH CLASS HIERARCHY



For an example of the density of the Smalltalk class hierarchy, try expanding the Dolphin Smalltalk `Collection` class in the class browser (**CTRL+B**). Here I've expanded just a few of its sub-branches to show some of its descendants. If you scroll down you can carry on expanding yet more branches to reveal a very large `Collection` family...

The density of the Smalltalk class hierarchy is one of the differences that really leaps out at you when you switch between programming in Smalltalk and Ruby. The Ruby `Array` class does not have all the same behaviour as the special-purpose collections of Smalltalk. The `Bag`, for example, groups together the same characters as they are added (it groups all the 'o' characters one after another, then all the 'l' characters and so on) whereas the `Array` adds them in sequence (in their 'original' order).

So, after running my code, Smalltalk's `Bag` contains something like this:

```
($o $o $e $d $r $h $l $l $l $w)
```

Whereas Ruby's `Array` contains this:

```
["h", "e", "l", "l", "o", "w", "o", "r", "l", "d"]
```

I could, of course, write some extra code to mimic the `Bag`'s behavior in Ruby. Indeed, I can get pretty close just by calling the `Array`'s `sort` method, to return an array like this:

```
["d", "e", "h", "l", "l", "l", "o", "o", "r", "w"]
```

Nevertheless, this illustrates a characteristic difference between Ruby and Smalltalk. While you *could* create dense hierarchies of increasingly specialised classes in Ruby, this is not usual and it is not the way the default Ruby class library is structured; in Smalltalk it is.

Another little difference worth pointing out is that Ruby has no `Character` class. When you wish to work with characters, you have two options - you can either work with strings with a length of 1 or you can work with `Fixnum` (integer) character codes. In older versions of Ruby, you could represent a character by indexing a string at 0 or by prefixing a character literal with a question mark:

chars.rb

```
"a"[0]
?a
p( "a"[0].class )
p( ?a.class )
```

Ruby 1.8 would display:

```
97
97
Fixnum
Fixnum
```

Note that Ruby 1.9 changed this behavior to treat individual characters as strings. The code shown above when executed by Ruby 1.9 or Ruby 2.0, for example, displays this:

```
"a"
"a"
String
String
```

In order to access a character's numeric code, you now have to use the `ord` method, like this:

```
p( "a"[0].ord )
```

Which shows:

```
97
```

In my *example10.rb* program, I've extended the `String` class itself by giving it a `to_chars` method which returns an array of characters (that is, one-letter strings), and an `isLetter` method which returns *true* if a string is one character in length and its ASCII value falls between 97 and 122 ('a'..'z') or between 65 and 90 ('A'..'Z'). Otherwise it returns *false*.

> Note because of the difference in the way Ruby deals with single characters, be sure to use the **example10-alt.rb** sample program rather than **example10.rb** with versions of Ruby earlier than Ruby 1.9.

Now, at first sight, it might seem that my implementation of `isLetter` is a bit of a hack (in the less positive sense of the word). Dolphin Smalltalk, I note, cheats a bit in its implementation of `isLetter` by calling out to an external library (presumably for reasons of efficiency). But when I checked on Smalltalk/V's implementation of the `isLetter` method I was surprised to find that it is remarkably similar to my own Ruby version. In particular, we both just do the old trick of comparing ASCII values.

Here's the test from my *Ruby* code:

```
if (self[0] >= 97 and self[0] <= 122)
  or  (self[0] >= 65 and self[0] <= 90)
```

And here's the test from *Smalltalk/V*'s `isLetter` method...

```
^((asciiInteger > 64 and: [asciiInteger < 91])
        or: [asciiInteger > 96 and: [asciiInteger < 123]])
```

The World According To Objects

Object Orientated Programming (OOP) is now so familiar to most programmers that it may seem superfluous to describe what it is and how it works. However, while most modern languages have adopted some features of OOP, they rarely, if ever, implement all the features defined by Smalltalk. For example, some languages such as C++, Delphi and Python implement a mix of OOP with procedural programming. C# and Java are more thoroughly object orientated but they do not fully implement a Smalltalk-like version of encapsulation.

> Background reading for this chapter is the third part of Chapter One of the *Smalltalk/V Tutorial* - in the PDF download, that's pages 21 to 29 (as shown in the Acrobat Reader bar - the actual page numbers at the tops of the pages themselves are 11 to 19).

What are objects?

An object is an item which contains data and a set of methods that act upon that data. Each object is an instance of a class. It is the class that defines the methods and the data. It is an important feature of Smalltalk that an object's data is invisible from outside the object itself.

Black box encapsulation

Unlike many other languages, Smalltalk does not let you directly reference an object's variables from the world outside that object (there is no 'dot notation' to let you write `ob.x` in order to get at a variable named `x` in an object called `ob`, for example). If you want to get or set the values of an object's variables you are obliged to send *messages* to the object (that is, to "call the object's methods"). There is no other alternative.

> "Related data and program pieces are encapsulated within a Smalltalk object, a communicating black box. The black box can send and receive certain messages. Message passing is the only means of importing data for local manipulation within the black box." (*The Smalltalk/V Tutorial*)

This is generally the case with Ruby too. In Ruby, an instance variable such as `@x` of an object, `ob`, cannot be 'got at' by calling `ob.@x` - that isn't even valid Ruby syntax. So, most of the time, Ruby is very close to Smalltalk in its insistence on

enforcing 'black box' encapsulation whereby the data inside an object is hidden from the outside world and access to that data can only be gained by sending messages to an object and having some answer returned by a method of that object.

But there are a few exceptions to the rule. To take just one example: if a Ruby object uses a method that modifies the receiver (that is, one that changes the *object itself* rather than yielding a new object), the *ingoing argument* to a method can be used like a '*byRef*' argument in a C or Pascal-like language. Here's a simple example:

```
def secretmethod( someVal )
   return someVal.reverse!
end
```

Here the programmer has written a method that reverses a string. The programmer expects people to call the method like this:

```
x = "hello world"
y = secretmethod( x )
puts y
```

However, one of the programming team notices that the ingoing parameter, `x`, can be used as a '*byRef*' argument. So, instead of creating another variable, `y`, to which the return value is assigned, he writes this:

```
x = "hello world"
secretmethod( x )
puts x
```

OK, no big deal. It turns out that the end result is the same either way:

```
x = "hello world"
y = secretmethod( x )
```

Now display the value of `x`:

```
puts x
```

This prints :

```
dlrow olleh
```

Or display the value of `y`:

```
puts y
```

This too prints :

```
dlrow olleh
```

But now, at some later date, it is decided that the method needs to return an *uppercase* reversed string. So the original programmer goes back and reimplements `secretmethod` like this:

```
def secretmethod( someVal )
   someVal.reverse!
   return someVal.upcase
end
```

Assuming that the principle of encapsulation protects the implementation details of methods from the outside world *('black box encapsulation')*, he believes that his new implementation ensures that the required operation (reversing and setting to uppercase the `someVal` argument) will have *exactly the same effect on all the code that uses this method.* But, now recall that one member of the programming team chose to use the *input* value of the argument, instead of the *return value* from the method. So now, we will have a situation where different members of the team are getting *different results* when they use the reimplemented method. The example below shows this. Here is the new version of the method:

```
def secretmethod( someVal )
   someVal.reverse!
   return someVal.upcase
end
```

Some programmers pass a string argument `x` and use the return value `y`:

```
x = "hello world"
y = secretmethod( x )
puts y
```

This is the result (as intended by the method's programmer):

```
DLROW OLLEH
```

But one sneaky programmer passes a string argument `x` and uses the changed value of `x` instead of the returned value:

```
x = "hello world"
y = secretmethod( x )
puts x
```

And this is the result (which is *not* what the method programmer intended):

```
dlrow olleh
```

You'll find this code (with the method bound into a class) in *secretmethod.rb*.

secretmethod.rb

```
class MyClass
    def secretmethod( someVal )
          someVal.reverse!
          return someVal.upcase
    end
end

ob = MyClass.new


x = "hello world"
y = ob.secretmethod( x )

puts x
puts y
```

Remember that previously the result was the same no matter how the code of the method was used. But by changing the implementation of the code inside the method, changes have also percolated through to code that uses that method. This violates the idea of black-box encapsulation. The inner workings of a method should be hidden from code outside the object containing that method. If a reimplementation of the code inside the method has side-effects on code outside the object containing that method, the object cannot be regarded as a 'black box'. That is because code outside that object is dependent on the *implementation details* of code inside the object. To operate as a 'black box' there should only be one way 'in' to each method (a message sent to an object) and one way out (the value which the method returns – in effect, the 'answer' it gives to the 'message').

> "I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages."
> (Alan Kay)

Note that, in any method which *modifies the receiver* (the append operator, << for example) will have the same effect. If it modifies an argument, those modifications can be used from outside the method and from outside the object containing that method. This is one example - and there are others - of Ruby giving the programmer the freedom to use or to ignore the *send-message and wait-for-answer* methodology of Object Oriented Programming whereas Smalltalk (generally) enforces that methodology.

BUT IS SMALLTALK *REALLY* ENCAPSULATED...?

OK, let me be brutally honest. It would be wrong to give the impression that Smalltalk's encapsulation is rigorous and Ruby's is not. In some cases, Smalltalk too lets you 'hang onto' ingoing variables, thereby potentially breaking black-box encapsulation. This may happen in the few cases in which a Smalltalk object (the 'receiver') may be modified without yielding a new object. For example, in *Squeak* I can create a class, `MyClass`, with a method, `secretmethod:`, which modifies an array by putting 'hello' at its first index. The method then returns the number, 123:

```
secretmethod: someVal
  someVal at: 1 put: 'hello'.
  ^123
```

If I send an array `x` to this method and assign the method's response to `y`, I get 123. But if I 'hang onto' the ingoing argument `x`, I get an array with 'hello' at the first index:

```
ob := MyClass new.
x := #( 'a' 'b' 'c' ).
y := ob secretmethod: x.
```

`x` is now: #('hello' 'b' 'c')
`y` is now: 123

In essence, since the method modifies the ingoing argument, `x`, it gives me the choice of using that argument as a 'byRef' parameter. And when I do that, my code is able to bypass the `MyClass` object's encapsulation - in other words, my code becomes *implementation dependent*. If the implementation of the method changes, the behavior of my code also changes.

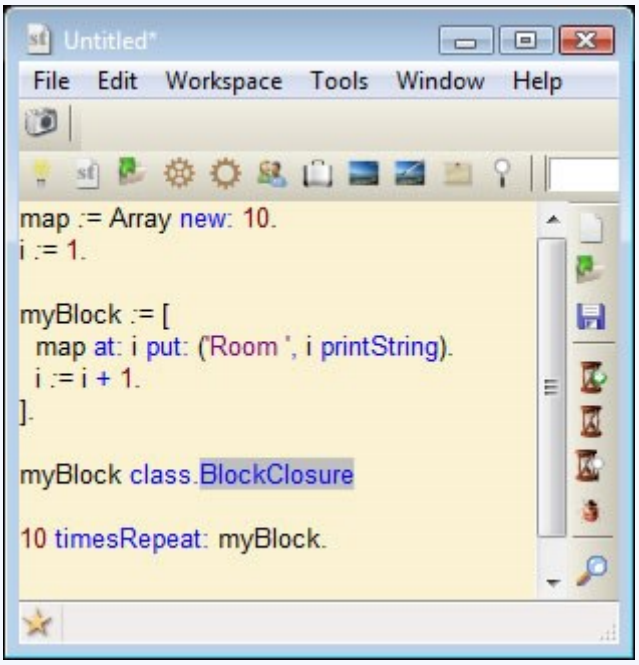
This is an exception to Smalltalk's general principle of 'black box' encapsulation whereby the implementation details of an object are hermetically sealed from the world beyond that object. All I can say is that this ability to 'dirty trick' your way around data-hiding is pretty rare in Smalltalk and it takes considerable programmer effort (perversity?) to accomplish. It is more common in Ruby as there are a great many receiver-modifying Ruby methods (for instance, all those methods that end with an exclamation mark, `!`). Moreover, even the ability to change data inside a Smalltalk array varies according to the implementation. Dolphin Smalltalk, for example, won't let me modify an array; if I try to do so, an error occurs warning me that I have attempted to modify a read-only object.

DOES INHERITANCE BREAK ENCAPSULATION?

Another interesting question to ask is whether Smalltalk's and (Ruby's) inheritance mechanism itself breaks encapsulation. In a paper entitled '*Object-oriented Encapsulation for Dynamically typed Languages*', Nathanael Schärli, Andrew P. Black and Stéphane Ducasse argue that both Smalltalk and Ruby fail to enforce encapsulation by making the methods and instance variables of a superclass visible to its subclasses so that "whenever a feature of a superclass is modified, the programmer must check all its (direct and indirect) subclasses to ensure that the change does not break existing code. This is because any subclass might use the modified feature and may rely on its old meaning."

WHAT KINDS OF OBJECTS CAN BE DESCRIBED?

It is a commonly made claim in OOP programming languages that 'everything is an object'. Closer scrutiny generally shows that this claim is not entirely true. In some languages 'primitive types' are not objects. In Ruby, while primitives are treated as objects, blocks are not (though they can be 'turned into' objects using special methods and classes). In Smalltalk blocks are objects - they are instances of a `Block` (or similar) class.



Here, in Dolphin Smalltalk, I have created a 'free standing' block object and assigned it to the variable, `myBlock`. I've verified that it is an instance of the `BlockClosure` class and I can now send this block object to the `timesRepeat:` method...

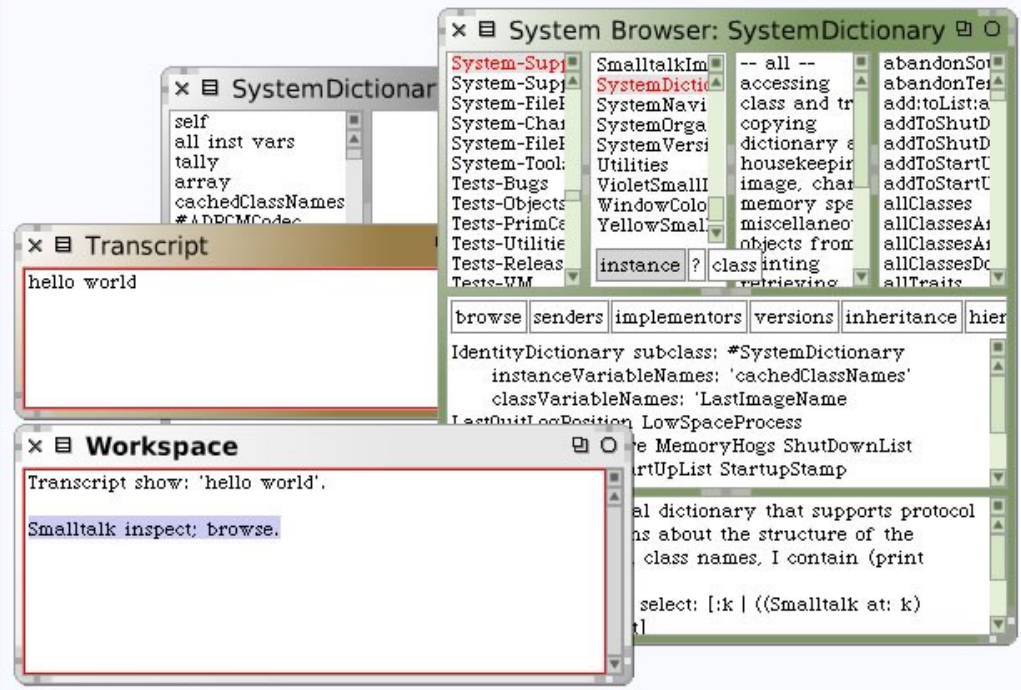
Even the Smalltalk environment and its component parts are treated as objects. For example, to print a 'hello world' in a `Transcript` window you would evaluate:

```
Transcript show: 'hello world'.
```

To anyone used to a visual programming environment such as C# or Delphi this may not seem a big deal. After all, this isn't too far removed from `textBox1.Text = 'hello world'`, is it? Well, how about this, then...?

```
Smalltalk inspect; browse.
```

Here the Smalltalk system itself is sent the two messages: `inspect` and `browse`. Evaluating this causes the Smalltalk 'system dictionary' first to appear in one window for inspection and then to be displayed in a hierarchical class browser. You can manipulate other features of the environment to display and write into system windows, for example, just by sending messages to the environment 'objects' using Smalltalk code. This is different from C#, Delphi and most other modern 'visual programming' languages. They provide you with visual objects that can be put into your own finished applications but they do not let you manipulate the objects of the native programming environment.



Smalltalk code can manipulate the Smalltalk environment. Here, in Squeak Smalltalk, I have evaluated two lines of code in order to display 'Hello world' in the Transcript window and pop up two special browser windows to inspect and evaluate the Smalltalk system dictionary.

Ruby comes pretty close to Smalltalk in providing access to all the objects in the Ruby system. But there is one obvious difference: Smalltalk defines its own programming environment; Ruby does not. Ruby development environments may know all about Ruby but Ruby does not know anything about them.

HOW DO OBJECTS COMMUNICATE AND BEHAVE?

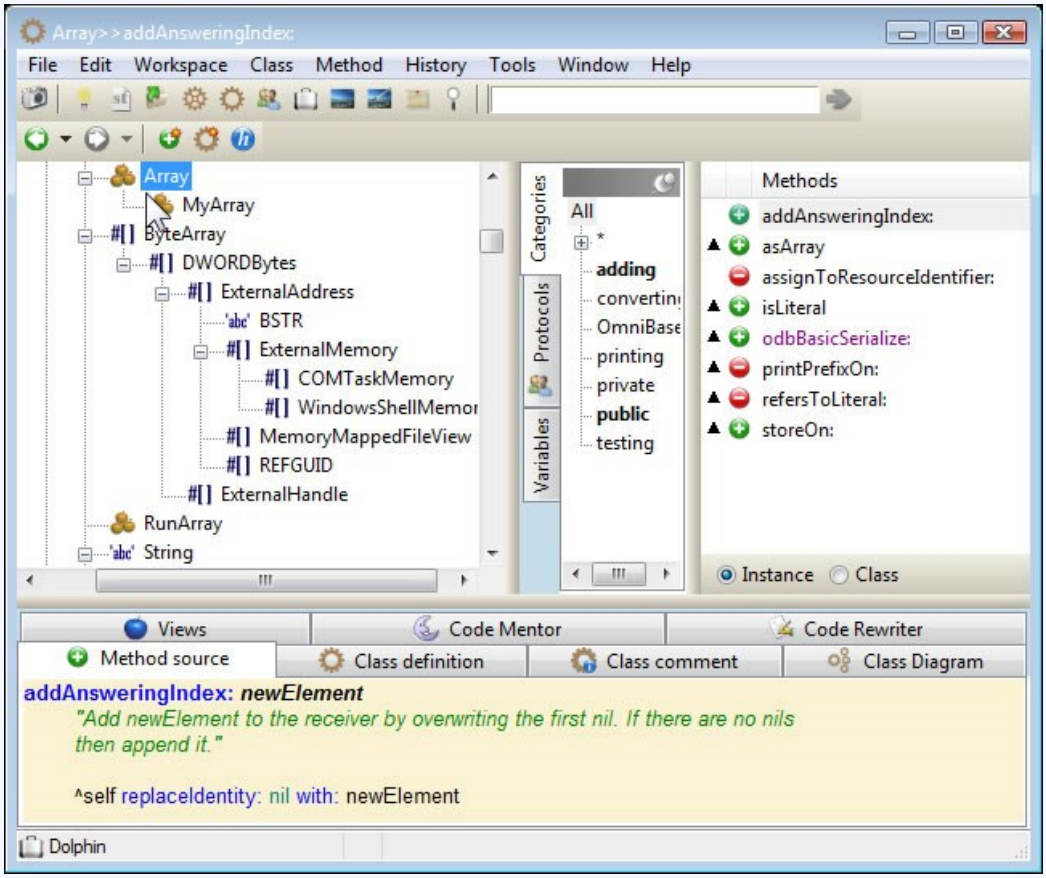
In general, both Smalltalk and Ruby provide similar ways of communicating with objects. Messages are sent to an object which then tries to find some way (a 'method') of responding. Often many different objects have methods of the same name - so that, in Smalltalk, the `printString` method is defined for all sorts of different objects just as, in Ruby, the `to_s` method is so defined.

POLYMORPHISM

The ability to invoke a method with the same name on different objects goes by the fancy name of 'polymorphism'. In the early days of OOP, polymorphism seemed a strange and mysterious concept. These days it is implemented in most OOP languages and is probably second nature to most programmers.

HOW DOES SMALLTALK (AND RUBY) ORGANIZE OBJECTS AND THEIR METHODS?

Here Smalltalk and Ruby are very different. In principle there is no reason why Ruby class hierarchies and Smalltalk class hierarchies should not be similar. But in fact, the standard Smalltalk class hierarchy is quite deep - with many levels of descent - while the Ruby hierarchy is very shallow - with rarely more than one or two levels of descent. In Smalltalk when a programmer needs a 'special version' of an existing class it is normal to create a new descendent of that class and then code any differences. In Ruby, many programmers make a habit of modifying the existing class - adding in new methods to extend the capabilities of `Array`, say, rather than created a new class `SomeNewKindOfArray`, to implement this new behavior. This is, I realize, a very broad generalization and is not true that all Smalltalk or Ruby programmers work in this way. However, it is certainly the case that the standard class library of Smalltalk is much deeper than that of Ruby.



Here is just one tiny section of the Dolphin Smalltalk class library showing the typical deep line of descent (here of special purposes arrays) in the left-hand Class pane.

To take a simple example. In Ruby this is the line of descent from the base class, `Object`, to the two collection classes: `Array` and `Hash` (the equivalent of a Smalltalk `Dictionary`). Note that both descend directly from `Object`:

```
Object->
   Array
Object->
   Hash
```

Now, here's the Smalltalk version (here I am using the Dolphin Smalltalk library but other Smalltalks take a similar approach). Note that both `Array` and `Dictionary` descend from a common `Collection` ancestor and that `Array` descends from several other ancestors each of which introduces new behavior that is inherited by `Array`:

```
Object->
   Collection->
         SequenceableCollection->
               ArrayedCollection->
                     Array
Object->
   Collection->
         Set->
               Dictionary
```

Moreover, in Ruby, `Array` and `Hash` are, by default, the end of the family tree (though programmers can, of course, create new descendent classes from them). In Dolphin Smalltalk, `Array` has many 'descendents') such as:

```
Object->
   Collection->
         SequenceableCollection->
               ArrayedCollection->
                     ByteArray
Object->
   Collection->
         SequenceableCollection->
               ArrayedCollection->
                     RunArray
Object->
   Collection->
         SequenceableCollection->
               ArrayedCollection->
                     String
```

In Ruby, incidentally, the `String` class is unrelated to `Array`. It too is an immediate descendent of `Object`. In Smalltalk, a `String` and an `Array` are close relatives (descendents of the `ArrayedCollection` class).

Dolphin Smalltalk's `Dictionary`, meanwhile, is the ancestor of several more specialized classes such as:

```
Object->Collection->Set->Dictionary->
    LookupTable
Object->Collection->Set->Dictionary->
    LookupTable->IdentityDictionary
Object->Collection->Set->Dictionary->
    LookupTable->IdentityDictionary->MethodDictionary
Object->Collection->Set->Dictionary->
    LookupTable->SharedLookupTable
Object->Collection->Set->Dictionary->
    LookupTable->SharedLookupTable->SharedIdentityDictionary
Object->Collection->Set->Dictionary->
    SystemDictionary
```

...and so on.

How do you maintain a Smalltalk (or Ruby) world of objects?

Smalltalk lets you work with source files in which you enter the text that defines your classes and objects. It also saves 'images' which store the state of your entire Smalltalk environment.

Smalltalk Images

If you save the Smalltalk image when you exit the environment, you can restart your work subsequently exactly where you left off. All your classes and objects will be in the same state as they were when you last left them; all the windows, text and graphics in the environment will be just as they were too. Saving a Smalltalk image is a bit like hibernating your PC or saving the state of a virtual PC.

Ruby has no equivalent of Smalltalk's image. It is debatable whether this is a good or a bad thing. On the one hand, Ruby's inability to store its state means that you cannot modify the state of Ruby dynamically and restore that state subsequently (well, unless you are running Ruby on a virtual PC, that is). On the other hand, this also means that you cannot accidentally 'bind into' Ruby changes that you had intended to be temporary rather than permanent. A Smalltalk image will preserve every change you make which means that you must be extremely careful when testing and trying out bits of code otherwise you may find that your Smalltalk environment permanently retains classes (even your mistakes) and objects from one session to the next.

This is not disastrous. You can clean up unwanted objects and delete unneeded classes and you can also, of course, save multiple images to get back to earlier versions of your environment. All the same, image saving does require a certain amount of care and attention if you are to avoid saving more than you intend.

MOVING ON...

That brings to an end this short series comparing the fundamentals of Smalltalk and Ruby. In these articles I have looked at specific features of the two languages - such as their syntax, style and use of objects. I hope it has given you at least a few insights both into the ideas that Ruby took from Smalltalk and that it may also have given you some food for thought about the pros and cons of object orientation, message-sending and encapsulation.

Smalltalk Resources

Here are some useful resources online where you can download free implementations of Smalltalk and find tutorials and eBooks.

Smalltalk Downloads

Dolphin Smalltalk (for Windows) – free community edition
http://www.object-arts.com/products/dce.html

Squeak Smalltalk (cross-platform) – free
http://www.squeak.org/Downloads

Cincom Smalltalk (commercial system, but with a free edition for personal use)
http://www.cincomsmalltalk.com/

Pharo ('A dynamic, pure object-oriented programming language in the tradition of Smalltalk')
http://pharo.org/

Smalltalk Tutorials

Learn Dolphin Smalltalk (my short tutorial)
http://www.bitwisemag.com/copy/programming/smalltalk/dolphintutorial.html

Dolphin Smalltalk video tutorials
http://www.object-arts.com/support/videos.html

Smalltalk: a Beginners Guide to Squeak (my short tutorial)
http://www.bitwisemag.com/copy/programming/smalltalk/smalltalk1.html

Squeak By Example – Squeak Smalltalk book (PDF download)
http://squeakbyexample.org/

Lawson English Squeak Tutorials (YouTube videos)
https://www.youtube.com/user/sparaig

Stef's Free Smalltalk Books – fantastic collection of classic books, now free
http://stephane.ducasse.free.fr/FreeBooks.html

Smalltalk.org – everything about Smalltalk!
http://smalltalk.org