Smalltalk's Influence on Modern Programming

Matt Savona. February 1st, 2008.

In September 1972 at Xerox PARC, Alan Kay, Ted Kaeher and Dan Ingalls were discussing programming languages in the hallway of their office building. Ted and Dan had begun to consider how large a language had to be to have "great power." Alan took a different approach, and "asserted that you could define the "most powerful language in the world" in "a page of code."" Ted and Dan replied, "Put up or shut up" (Kay). And with that, the bet to develop the most powerful language was on. Alan arrived at PARC every morning at 4am for two weeks, and devoted his time from 4am to 8am to the development of the language. That language was Smalltalk.

Kay had "originally made the boast because McCarthy's self-describing LISP interpreter was written in itself. It was about "a page", and as far as power goes, LISP was the whole nine-yards for functional languages." He was sure that he could "do the same for object-oriented languages" and still have a reasonable syntax. By the eighth morning, Kay had a version of Smalltalk developed where "symbols were byte-coded and the receiving of return-values from a send was symmetric" (Kay). Several days later, Dan Ingalls had coded Kay's scheme in BASIC, added a "token scanner", "list maker" and many other features. "Over the next ten years he made at least 80 major releases of various flavors of Smalltalk" (Kay).

The notion of object oriented programming had been expressed, and possibly implemented in less successful languages that may have predated Smalltalk. But Alan Kay was able to express the idea quite fluidly in his language, and this has had a tremendous impact on all languages that followed. Of the six main ideas that were outlined in the Smalltalk scheme, five of these ideas are of great significance, even today:

1. *Everything is an object.*
2. *Objects communicate by sending and receiving messages (in terms of objects)*
3. *Objects have their own memory (in terms of objects)*

4. *Every object is an instance of a class (which must be an object)*
5. *The class holds the shared behavior for its instances*

In Smalltalk, "everything is an object." This is slightly different than the later implementations of Java, C++ and other successors, in that even primitives (ints, bools, chars) are represented as objects in Smalltalk.

The language's contribution to object oriented programming was not the only impact it's left on computing, however. The Smalltalk-76 release featured a "development environment featuring most of the tools now familiar including a class library code browser/editor" (Wikipedia). This environment featured an overlapping window interface, which was emulated almost exactly by the first Macintosh desktop. In addition, the environment was a true IDE, which improved greatly on a purely text-editor approach to writing code.

Smalltalk-80 was the first publicly available release of the language, and while it has been expressed in many branded variations since that time, the syntax has remained fairly consistent throughout the years. In 1998 an ANSI standard was compiled for Smalltalk (Wikipedia).

Before we delve into the syntactical influences Smalltalk had on modern languages, let me briefly discuss the history Ruby. The language itself was born on February 24th, 1993. Japanese software engineer, Yukihiro "Matz" Matsumoto named it "Ruby" as a joke alluding to the name of the Perl scripting language (Takahashi, transcribed by Sieger). On December 21st, 1995 the first public release of Ruby was made (0.95), due to test failures however, Matz released three new versions of Ruby over the next two days. On Christmas, one year later, Ruby 1.0 was released. Then, in the summer of 1997, Matz was hired by Netlab to develop the language full-time. It wasn't until 2002 that Ruby became well documented enough in languages other than Japanese to be widely accepted around the world. Arguably, the biggest boost to Ruby's popularity came in July 2004, when the popular web framework Ruby on Rails was released to the public.

Implementation

Smalltalk is generally compiled into byte-code and interpreted by a virtual machine. While there are Ruby virtual machines in development, the language is usually single-pass interpreted. Both languages implement a garbage collector and are dynamically typed. They are also considered reflective because they can reflect and change on themselves at run-time (for example, in Smalltalk: `true become: false` is a valid statement) (Wikipedia).

The Basics

To understand either language, we'll need to equate their most basic features. What follows are simple examples to get us started. Please note that the examples below are not exhaustive, in both languages, there are likely several other ways to approach the task at hand.

Variable Declarations and Assignments

Smalltalk

```
"this is a comment in Smalltalk"

| x y | "declare the variable x and y"

x := 1 "assign x a value of 1"

y := $q "assign y the character 'q'"
```

Ruby

```
# this is a comment in Ruby

x = 1 # declare and assign x a value of 1

y = 'q' # declare and assign y a value of 'q'
```

Printing to the Screen

Smalltalk

```
Transcript show: 'Hello World'.
```

Ruby

```
p 'Hello World'
```

Conditional Statements

Smalltalk

```
x > 1
ifTrue: [Transcript show: 'x is greater than 1'.]
ifFalse: [Transcript show: 'x is less than or equal to 1'.]
```

Ruby

```
if x > 1
 p 'x is greater than 1'
else
 p 'x is less than or equal to 1'
end
```

Looping

Smalltalk

```
| x |

x := 10

x timesRepeat: [Transcript show: 'this is an iteration'.]

1 to: x do: [Transcript show: 'this is an iteration'.]

[x > 5] whileTrue: [x := x + 1. Transcript show: 'this is an iteration'.]
```

Ruby

```
x = 10

x.times{ p 'this is an iteration' }

1.upto(10){ p 'this is an iteration' }

while i > 5 do
 i += 1
 p 'this is an iteration'
end
```

True Objects

As stated earlier, Smalltalk's major contribution to modern languages was its true Object-oriented nature. The language itself has only a handful of reserved words, all of which are singleton instances of some other object. These keywords are: nil, true, false, self, super and thisContext. Ruby builds on this list of reserved words, but is still true to the idea that every aspect of the language is an Object. In Ruby, if you were to print the class of a keyword, here's what you would see:

```
[msavona@fruity:~]$ irb
irb(main):001:0> p false.class
FalseClass
=> nil
irb(main):002:0> p true.class
TrueClass
=> nil
irb(main):003:0> p nil.class
NilClass
=> nil
```

In Matz own words, "I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python" (Matz). And indeed, both languages are so strongly rooted in fully Object-oriented design, that even operators are simply methods belonging to any given class. That is, to utilize a binary operator, such as == on an object (let's say, Char in Smalltalk, String in Ruby) requires only that the Char (or String) class implement a method named "==" and return a Boolean value (which is also an object!).

Arrays and Collections

Both Ruby and Smalltalk describe arrays in a similar fashion. In Smalltalk, arrays are fixed length collections, meaning that once they are declared their size may not change:

```
| myArray x |
myArray := #(1 2 3 $a $b $c).
x := myArray at: 4
"arrays indices in Smalltalk begin at 1, the above statement yields $a"
```

The above describes an array of mixed type, which includes the Integers 1, 2, 3 and the Characters 'a', 'b', 'c' (in Smalltalk $*x* [where *x* is any character] denotes a single character). In Ruby, the same is possible, but the programmer may dynamically size the array (like a linked list) as he sees fit:

```
myArray = [1, 2, 3, 'a', 'b', 'c']
```

```
myArray << 'd'
x = myArray[3] # yields 'a'
```

Smalltalk has several Collection classes which expose a number of methods that make them far more expandable than ordinary arrays. Some basic usage of a Collection in Smalltalk is as follows:

```
| myCollection x |
myCollection := OrderedCollection with: 1 with: 2 with: 3 with: $a with: $b
myCollection addLast: $c

Transcript show: myCollection at: 4. "$a"
```

In Ruby, any collection other than an integer bound array is considered a Hash, which is simply a collection with key => value pairs. These are very simply defined as:

```
myCollection = { :please => "give", :me => "me", :an => "A+" }

p myCollection[:an] # prints A+
```

The fancy keys, prefixed with a colon (:) are symbols (though they don't have to be). This brings us to our next topic…

Symbols

One of the most exciting features of both languages (as someone who has no prior knowledge of either) is the existence of "symbols". In today's popular languages (I'm referring to Java and C#), we don't have them (though we could come close). The idea of a symbol is baffling as you read code in a language like Ruby or Smalltalk, as you can clearly understand the application of the symbol, but struggle infinitely to understand its benefit. A symbol in these languages is a very efficient and logical way to represent an object by a name, where the name itself is not a string, but is readable to the programmer as one. There are many, many, many resources online that attempt to explain this concept. I think Jim Weirich does a good job of explaining their purpose best:

1. *Naming keyword options in a method argument list*

2. *Naming enumerated values*
3. *Naming options in an option hash table*

In Smalltalk, a symbol is represented as:

```
#thisIsMySymbol
```

In Ruby, the same:

```
:thisIsMySymbol
```

Take for example a hash of animal noises, referenced in two cases below, once with string keys, and again with symbols:

```
animalNoises = { "cat" => "meow", "dog" => "woof", "cow" => "moo" }

animalNoises = { :cat => "meow", :dog => "woof", :cow => "moo" }
```

As an observer of the code, their intent is identical, but the application of symbols is a more logical approach if the construct is available to you. That is, if you don't need to treat your keys as strings, why bother doing it? A symbol is a unique, readable identifier for some object you want to represent; they consume less memory (it is worth noting, however, that they are not garbage collected) and are immutable. Values can be retrieved from this hash table as follows:

```
dogNoise = animalNoises["dog"]

dogNoise = animalNoises[:dog]
```

The point is, symbols exist as a means to name objects in Ruby and Smalltalk, to treat them otherwise would be an inefficient use of the language.

Class and Method Declarations

And last, but certainly not least, we will cover how you might define new classes and methods in both languages. The following example is from Ulrik Schultz's Smalltalk Tutorial:

```
Employee subclass: #SalariedEmployee
   instanceVariableNames:  'position salary'
   classVariableNames: ' '
```

```
   poolDictionaries: ' ' !

! SalariedEmployee publicMethods !

position: aString
   position := aString !
position
   ^position !
salary: n
   salary := n !
salary
   ^salary ! !
```

In the above code, the class SalariedEmployee is defined as a subclass of Employee, and it has two instance variables (variables that exist uniquely for each instance of this class): position and salary. Those "instance variables" are actually public methods, in this case. If you are familiar with Java setter/getter methods or C# get/set properties, this is right along those same lines. A user may call the position method of the SalariedEmployee class to set or get the person's position, and likewise may do the same for their salary.

This code is easily translated into Ruby:

```
class SalariedEmployee < Employee
   attr_accessor :position, :salary
end
```

The `attr_accessor` permits a programmer to declare instance variables and their setter/getter methods with no effort at all. This single line produces two instance variables (@position and @salary) which can be set or get by those names once the class has been instantiated. It is also possible to utilize `attr_reader` or `attr_writer` for instance variables that may only be read (get) or written (set) - respectively. Since this example is an extreme simplification of the Smalltalk equivalent, let's write the long handed version:

```
class SalariedEmployee < Employee
   public
   def set_position(position)
```

```
      @position = position
   end

   public
   def get_position
      @position
   end

   public
   def set_salary(salary)
      @salary = salary
   end

   public
   def get_ salary
      @salary
   end
end
```

Note that the "public" access modifier is not required, and it is assumed by default, but Ruby is capable of permitting you to your methods as public, private or protected.

Conclusion

I hope that my comparisons have shown how revolutionary Smalltalk really was at the time it was developed. It truly presented programming paradigms that have lasted decades and have sincerely influenced newer languages, like Ruby. The more I use Ruby, the more excited I am to learn how it works, and knowing that its roots are largely in a language conceived almost 40 years ago is quite incredible. Looking forward, I wonder what new languages will build on this foundation, and how they might make programming faster, and more fun!

Work Cited

Kay, Alan. "The Early History of Smalltalk."

http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html.

Matsumoto, Yukihiro. "An Interview with the Creator of Ruby."

http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html.

Schultz, Ulrik. "Smalltalk Tutorial: Classes."

http://www.daimi.au.dk/~ups/OOVM/smalltalk-tutorial/chap5.html.

Wierich, Jim. "Symbols Are Not Immutable Strings."

http://onestepback.org/index.cgi/Tech/Ruby/SymbolsAreNotImmutableStrings.red.

Wikipedia. "Ruby (Programming Language)."

http://en.wikipedia.org/wiki/Ruby_(programming_language)

Wikipedia. "Smalltalk."

http://en.wikipedia.org/wiki/Smalltalk.