Smalltalk

While we have discussed Smalltalk, this chapter will give a more focused description of the language.

Smalltalk is one of the original Object-Oriented Programming (OOP) languages and came from Xerox's Palo Alto Research Center (PARC) in the 1970s. It inspired much of today's graphical user interface design (Steve Jobs was given a demo of Smalltalk at PARC and went on to build the Macintosh) and influenced many of today's programming languages (especially Objective C and Ruby). While not as well-known as many subsequent languages, it has a vibrant community of users who continue to provide leadership in the software industry.

Like its influential predecessor Lisp, but unlike most of today's better-known languages, Smalltalk is **image-based** in that "programming" consists of modifying an existing system rather than creating a new system from scratch. That is, while the typical C program is created by compiling text files into an executable, a Smalltalk program is created by cloning an existing system and modifying it to provide new capabilities. The program itself is represented internally by objects (including classes and methods), and changes are made (i.e., code is written) by sending messages to existing objects.

An "image" is a snapshot (using a camera metaphor) of a live object space written to a binary file. An object space saved in this way can be exactly recreated by reading the image back into memory and continuing execution from the point of the snapshot (much like a fork() operation in C creates a new process with a copy of the existing data). This process of writing an environment to disk and reading it back is similar to the hibernate operation on a modern laptop computer—when the image is restored all the windows are open to the same place on the screen and they have the same content.

A typical running Smalltalk system consists of an operating system process (a virtual machine to interpret and/or compile source code) and an object space containing all the code and data (typically copied into RAM from the disk-based image). The virtual machine (VM) is also responsible for automatic garbage collection—reclaiming space used by unreferenced objects. If the VM terminates without saving an image of the object space, then changes made in the object space are thrown away and restarting from an earlier image will restore the object space to the prior state. Most Smalltalk dialects record programmer activity to a separate log from which code changes can be selectively reapplied so that programmers can experiment with minimal risk of losing work.

Because a Smalltalk program (classes and methods) is represented by objects in the object space, the act of writing a program involves manipulating the object space. You create a new class (a subclass) by sending a message to an existing class with arguments that describe the new class's schema and you create a new method by sending a message to a class with the source string for the new method. These operations are typically done using tools built into the environment. Because the tools and the code live together in the same object space, there is a rich tradition of extensions, such as refactoring tools (which originated in Smalltalk).

Smalltalk Syntax

Sending Messages (Instead of Issuing Commands)

Where other languages might have an elaborate syntax (including many keywords and operators), Smalltalk takes a simple idea—message passing—to an extreme. Instead of issuing a series of commands for the computer to follow, the programmer is scripting a series of polite requests that objects make of one another (and the receiver responds by sending other messages). The syntax of message sending is to name a variable (which always holds a reference to an existing object) and then specify the message with any arguments. This placement of the object first is backwards from the approach of most languages where the procedure is first. For example, to close a file, most languages would use a construct like the following (where the semicolon ends an expression):

```
Close(myFileHandle);
```

In contrast, Smalltalk puts the object first rather than the command first (and expressions end with a period or dot):

```
myFileHandle close.
```

Three Message Types

There are three types of messages: (1) **unary** messages (like the 'close' example above), (2) **binary** messages that take exactly one argument (like '+' and '*' in the example below), and (3) **keyword** messages where one or more arguments follow a word ending with a colon (like 'between:and:' below). The precedence is unary, binary, and keyword, with left-to-right within a particular message type. While elegant and consistent, this creates some confusion for people coming from other languages. The following expression (with two binary messages) evaluates to 20 in Smalltalk (using strict left-to-right evaluation for binary messages), while other languages would give 14 (treating '*' as a language-defined operator with higher precedence than '+'):

```
2 + 3 * 4.
```

As in other languages, parenthesis can be used to change the order of evaluation. If you want the above expression to evaluate to 14, use the following:

```
2 + (3 * 4).
```

Keyword messages use words rather than just the position to identify the argument. For example, to check for a value in a certain range a traditional language would use something like the following:

```
between(x, y, z);
```

Smalltalk makes more explicit which is the value being tested, which is the lower bound, and which is the upper bound. While this requires some more typing, it makes the code easier to read:

```
    x between: y and: z.
```

Five Reserved Words

In Smalltalk there are only five reserved words: *self*, *super*, *true*, *false*, and *nil*. The first two, *self* and *super*, are "pseudo-variables" that are used in a method to reference the receiver (an object) of the current message. When a message is sent to *self*, method lookup starts in the class of the receiver (without regard to the class where the method being executed is implemented). When a message is sent to *super*, method lookup starts in the superclass of the class where the method being executed is implemented. The remaining reserved words, *true*, *false*, and *nil*, are actually globals that always reference the single instance of the classes True, False, and UndefinedObject respectively.

Control Flow (Loops and Conditionals)

The things that are typically done in other languages using reserved words, including class definition and flow-of-control, are done in Smalltalk by sending a message to an object. Thus, while a traditional language would use a construct like the following for a loop (where *for* is a command):

```
    for (i = 0; ++i; i < sizeof(array)) { DoSomething(array(i)); };
```

Smalltalk would use a message ('to:do:') sent to an integer to create a similar loop:

```
    0 to: array size do: [:i | (array at: i) doSomething].
```

Here the 'to:do:' message is sent to an instance of SmallInteger with two arguments, another SmallInteger and a Block (code that can be executed later). In fact, because this sort of iteration is extremely common, another message, 'do:,' is used to handle iteration without explicitly managing the counter (allowing the programmer and the code to focus on a higher level):

```
    array do: [:each | each doSomething].
```

In the above example, the code block (inside the square brackets) is an object passed to the 'do:' method and is evaluated repeatedly, once for each object in the array. Code blocks can have arguments, like 'each' in the above example (the preceding colon is syntax for naming a block argument).

Code blocks are also used for conditionals. In the following example, two code blocks are provided as the arguments to the message 'ifTrue:ifFalse:', where the receiver is a Boolean expression:

```
    (x < 0) ifTrue: [self doThis] ifFalse: [self doThat].
```

Compare the above Smalltalk code with the following from a more traditional language where *if* and *else* are reserved words in the language syntax rather than simply messages to objects:

```
    if (x < 0) then {doThis()} else {doThat()};
```

Assignment and Return

Smalltalk has a couple built-in operators that are not message sends. The first is variable assignment:

```
   x := 3.
```

The second built-in operator specifies an immediate return from a method with a particular value:

```
   ^x.
```

All methods return an object—either explicitly or implicitly. By default, the object returned is the receiver, but using the up-arrow (as above), a method can return an explicit value. If the goal is to return early from a method that would not otherwise return a value, then the return value is typically (by convention) the default return value, *self*:

```
   (x < 0) ifTrue: [^self].
```

Method Definitions

A method is always defined in the context of a class and starts with a template or prototype that includes the method name and names for the arguments. Temporary variables are defined in a method or block by enclosing the names in vertical bars before any expressions:

```
add: anObject
   | sum |
   sum := self + anObject.
   ^sum.
```

Message Cascades

Because it is often useful to send a series of messages to the same object, Smalltalk provides a shortcut so that you don't have to repeat the receiver in a method. In the following example, two messages are being sent to *self*, but the receiver is only identified once (note the semicolon):

```
   self doThis; doThat.
```

Classes are Objects

In Smalltalk, all values are objects that are instances of some class. Unlike many other OO languages, Smalltalk implements the classes themselves as objects. Methods can be defined in the metaclass for a class, so that messages sent to the class will find and evaluate the methods. This provides what in other languages might be characterized as a *static function*. In Smalltalk, we speak of having methods "on the instance side" of a class (where they will be evaluated if a message is sent to an instance of the class) and having methods "on the class side" (where they will be evaluated if a message is sent to the class). The tools typically make this easy to manage.

Constants

The Smalltalk language syntax provides for creating (or referencing) various constants in a method.

A **Number** is generally defined in a familiar manner. A series of digits without punctuation (and with an optional preceding minus sign) is interpreted as an instance of the class Integer. If a decimal point (or dot) is included, then the number will be interpreted as an instance of the class Float. Floating point numbers are also permitted to have an exponent component (with an optional sign). A number that ends with 's' followed by one or more digits will be treated as an instance of the class ScaledDecimal.

A **Character** can be one or more bytes long, depending on the dialect. The dollar symbol introduces the character constant. The following identifies a Character with the code point (ASCII value) of 65.

```
    x := $A.
```

A **String** is defined as a series of characters enclosed in a single-quote or apostrophe character ('). To include a single-quote inside a string, simple double it. Thus, 'ab''cd' is a five-character string where the third character is a single-quote character.

A **Symbol** is a subclass of String in which the system guarantees that only one instance with the specified sequence of characters will exist in the object space. This allows for more efficient equality comparison because rather than doing a character-by-character test, the system only needs to compare the object IDs of two symbols to see if they are the same. Symbols are often used as flags in the way that an enumeration or #define constant might be used in C. It is more efficient than a string and more readable than a number. To define a constant symbol, precede a string constant with a hash symbol. In many common situations (a string with only alphabetic characters), the quotes may be excluded. The following example shows two constant symbols:

```
    x := #'this is a symbol'.
    y := #thisIsAlsoASymbol.
```

A constant **Array** can be defined in a method using a hash and left parenthesis to begin the list and a right parenthesis to end the list. Inside the list may be any other constant (including Arrays). The following example shows a constant Array containing six objects, an Integer, a Float, a Character, a String, a Symbol, and another Array with three objects (elements are separated with whitespace):

```
    x := #(42 -1.25e-13 $A 'hello' #world #(1 2 3))
```

Comments

Comments are included in a method by enclosing them with double-quote characters ("). To include a double-quote character in a comment, double it. The following line has a comment:

```
    x := Float pi. "3.14159265358979"
```

Scope for Name Lookup

As code (generally a method) is compiled, names are looked up using a succession of six (6) scopes, listed here from innermost (found first) to outermost (found last):

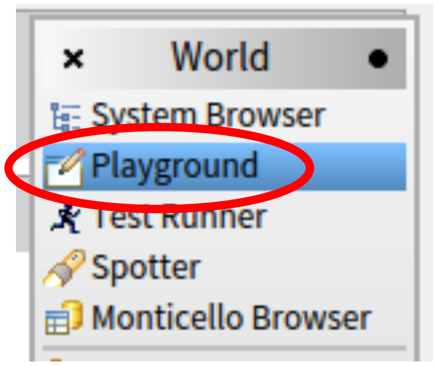
1. **Block arguments and temporaries**
    a. Block temps are initialized to nil (the sole instance of the UndefinedObject class) each time the block is evaluated
    b. Visible only within the block
2. **Method arguments and temporaries**
    a. Method temps are initialized to nil each time the method is called, even if called recursively
    b. Visible only to code within the method
3. **Instance variables**
    a. Initialized to nil when the object is instantiated
    b. References are preserved during the life of the object
    c. Values are visible only to methods defined in the class where the variable is defined and its subclasses
    d. No direct access is permitted in standard Smalltalk; the only way to get the value of an instance variable from outside the object is to send a message to the object
4. **Class variables**
    a. Initialized to nil when the class is first defined
    b. References are preserved during the life of the class
    c. Values are visible to all methods in the metaclass, in the class, and in all subclasses
5. **Pool Dictionaries**
    a. Each class definition can have zero or more key/value collections (called a Dictionary in Smalltalk, but typically called a Hash in other languages)
    b. A single dictionary can be shared among multiple classes
    c. Values are visible and shared among methods in all classes that reference the pool
    d. These are typically used for shared constants (e.g., map a color name to a number)
6. **Globals**
    a. Smalltalk provides a global namespace that is visible and shared by all methods
    b. These are generally used to hold classes

By convention, variables in scopes 1-3 are named with an initial lowercase letter and variables in scopes 4-6 have names that begin with an initial capital letter. As in other languages, it is considered a good engineering practice to use the narrowest scope that will work correctly. That is, a global is generally avoided when something like a class instance variable would do. And while a block can reference (read and write) method temporary variables, if the variable is only used within the block and does not need to preserve its value between block evaluations, it would likely be more efficient to move the variable to inside the block (as the compiler can generate simpler code).

Note that a class (an instance of Metaclass) may have instance variables. The values will be visible only to methods on the metaclass itself (i.e., class-side methods), not to instances, and not to subclasses.

The Workspace

In Chapter 2 we looked briefly at the three windows that are part of the Seaside One-Click Experience. The Workspace is a text area where Smalltalk expressions can be typed and evaluated. To get another Workspace you may left-click on the desktop to get a World menu and select 'Playground' from that menu.

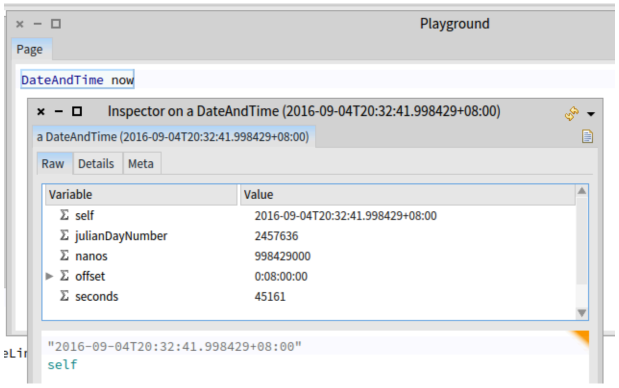


After you have the new Workspace, try entering a Smalltalk expression (like '100 factorial') and then type <Ctrl>+<P> (for 'print-it'). You should see a very large integer highlighted. Press <Backspace> or <Delete> while the text is selected to delete it.

If you type <Ctrl>+<D> (for 'do-it'), the expression will be evaluated but the resulting object will be ignored. This is useful when we want to evaluate an expression to cause some side-effect to occur (like registering a new Seaside application) and are less interested in seeing a printout of the result.

Sometimes we want to investigate a returned object in some detail. Most Smalltalk dialects provide an *inspector* to look at objects in more detail that simply printing some text in a workspace. In a Workspace enter the following and press <Ctrl>+<I> (for 'inspect-it'; on a Mac you might need to use <Apple/Command>+<I>).

```
DateAndTime now.
```



```
DateAndTime now
```

Inspector on a DateAndTime (2016-09-04T20:32:41.998429+08:00)

a DateAndTime (2016-09-04T20:32:41.998429+08:00)

Variable	Value
self	2016-09-04T20:32:41.998429+08:00
julianDayNumber	2457636
nanos	998429000
offset	0:08:00:00
seconds	45161

```
"2016-09-04T20:32:41.998429+08:00"
self
```

The window in front of the Workspace is an inspector and it has as its title the name of the class. The inspector has three tabs and the first tab has two panes: (1) a scrolling list of the object and its instance variables, (2) a text area with a printed representation of the selected instance variable and a place where expressions can be entered and evaluated.

If you right-click in the Workspace you will get a context menu of commands that apply to the Workspace. Some of these are similar to what you would expect in a text area (cut/copy/paste/etc). Others provide commands for executing Smalltalk code.

The System Browser

As we saw with the Workspace, you can get a new System Browser by clicking on the Browser icon on the desktop and dragging it to a convenient location. The System Browser gives you a way of looking at the classes and methods defined in the current object space (usually called the *image*). The first column lists categories used to group classes. Scroll through the list to see a sampling of categories:

- Kernel: Forms the innermost foundation for the Smalltalk library
- Collections: Classes like Array and Set; used to aggregate objects
- Files: Classes used to communicate with the native operating system's files
- Balloon, Graphics, Morphic: Classes used to create Pharo's Graphical User Interface (GUI)
- Network: Classes used to communicate with the native operating system's networking layer
- SUnit: The original XUnit testing framework that inspired various other unit testing frameworks
- Compiler, System: Classes that support the internal operation of Pharo Smalltalk
- Monticello: A source code management and version control system
- Refactoring: Classes to support automated refactoring of code
- Seaside: A framework for developing web applications in Smalltalk
- Scriptaculous: An add-on to Seaside to support this popular JavaScript library

If you right-click in the class category list you get a context menu that allows various category-related operations (add/delete/etc.). Perhaps most useful is the 'find class…' menu command. This will offer a dialog box into which you can enter a fragment of a class name. Pharo will then present a list of classes with a matching name. When you select one the browser will select the category and class.

The second column shows a list of classes in the selected class category. The list is organized in a hierarchy (and alphabetically when classes have a shared superclass). Right-click to get a context menu, including the following:

- new class template: Replace the text area at the bottom with a generic class definition
- subclass template: Provide a template for creating a subclass of the selected class
- browse: Open a new System Browser with this class selected
- browse hierarchy: Open a class hierarchy browser showing all superclasses (even those not in the current class category)
- browse references: Show a list of every method in the system that references this class
- chase variables: Open a browser showing instance variables and methods that reference a selected instance variable

Below the second column is a checkbox that allow you to view methods on the instance side and class side of the class. A frequent error is creating a method on the wrong side of a class. Most of the time you should be on the instance side; when you go to the class side be sure to come back.

The third column gives a list of method categories and has a context menu that allows you to manage method categories (add, rename, remove, etc.) and has other tools similar to the class context menu. The fourth column gives a list of the methods in the selected method categories for the selected class.

This list has a context menu that allows management of the methods and has other tools similar to the class context menu.

Below the four columns are a series of buttons that mostly open new browsers that are also available from the context menus. For example, when a method is selected it is often interesting to browse senders of that message or other implementers of that message.

Most Pharo windows (that you will interact with in the context of writing a Seaside Application) have various standard window manipulation capabilities. You can close a window by clicking in the red circle at the top left. You can move a window by clicking on the title bar and dragging. You can resize a window by dragging one of the four corners.

The Changes Browser

One very valuable tool is the Changes Browser. Left-click on the desktop to get a World menu, select 'Tools…' then 'Recover lost changes…'. This will present you with a list of events in reverse chronological order (the most recent will be at the top). The 'SNAPSHOT' event refers to saving an image of your object space and the 'QUIT' event refers to quitting Pharo without saving an image. If your Pharo image crashes (or you quit without saving), you can get back all of your work. Selecting one of the events (generally the most recent one) will show you a list of your activity including class definitions, method definitions, and expressions evaluated in a workspace. You can replay any or all of these actions to get your object space back to the point where you lost your work.

The Transcript

The Transcript is a special workspace that is associated with the global named 'Transcript.' From the World menu (left-click on the desktop) select 'Tools…' and 'Transcript.' From another workspace evaluate the following expression with <Ctrl>+<D> (for 'do-it').

```
Transcript cr; show: 100 factorial printString.
```

The string should appear on the Transcript. This is quite handy for debugging as an alternative to interrupting the code execution. You can dump a string to the Transcript at various places in your code to see what is happening. This is similar to what you would do with 'printf()' and standard output in other languages.