Smalltalk

From Wikipedia, the free encyclopedia
(Redirected from Smalltalk programming language)

Smalltalk is an object-oriented, dynamically typed, reflective programming language designed at Xerox PARC by Alan Kay, Dan Ingalls, Ted Kaehler, Adele Goldberg, and others during the 1970s, influenced by Sketchpad and Simula. The language was generally released as Smalltalk-80 and has been widely used since. Smalltalk is in continuing active development, and has gathered a loyal community of users around it.

Contents

History

Smalltalk was invented by a group of researchers led by Alan Kay at Xerox Palo Alto Research Center; Alan Kay designed the system, which Dan Ingalls implemented. The first implementation, known as Smalltalk-71, was created in a few mornings on a bet that a programming language based on the idea of message passing inspired by Simula could be implemented in "a page of code". A later version actually used for research work is now known as Smalltalk-72. Its syntax and execution model were very different from modern Smalltalk, so much so that it could be considered a different language.

After significant revisions which froze some aspects of executional semantics to gain performance, the version known as Smalltalk-76 was created. This version added inheritance, featured syntax much closer to Smalltalk-80, and had a development environment featuring most of the tools now familiar to Smalltalkers.

Smalltalk-80 added metaclasses, something which helps keep the "everything is an object" statement true by associating properties and behavior with individual classes (for example, to support different ways of creating instances). Smalltalk-80 was the first version made available outside of PARC, first as Smalltalk-80 Version 1, given to a small number of companies (Hewlett-Packard, Apple Computer, Tektronix, and DEC) and universities (UC Berkeley) for "peer review" and implementation on their platforms. Later (in 1983) a general availability implementation, known as Smalltalk-80 Version 2, was released as an image (platform-independent file with object definitions) and a virtual machine specification.

Two of the currently popular Smalltalk implementations are descendants of those original Smalltalk-80 images. Squeak is an open source implementation derived from Smalltalk-80 Version 1 by way of Apple Smalltalk. VisualWorks is derived from Smalltalk-80 version 2 by way of Smalltalk-80 2.5 and ObjectWorks (both products of ParcPlace Systems, a Xerox PARC spin-off company formed to bring Smalltalk to the market). As an interesting link between generations, in 2002 Vassili Bykov implemented Hobbes (http://wiki.cs.uiuc.edu/VisualWorks/Smalltalk-80+in+a+box), a virtual machine running Smalltalk-80 inside VisualWorks. (Dan Ingalls later ported Hobbes to Squeak).

IBM has indicated that VisualAge Smalltalk will be supported by partner company (and VisualAge implementors) Instantiations. Cincom, Object Arts, GemStone, and other vendors continue to sell Smalltalk environments. The open Squeak implementation has a relatively active community of developers, including many of the original Smalltalk community. GNU Smalltalk (http://www.gnu.org/software/smalltalk/) is a free (GPL) implementation of Smalltalk from the GNU project. There is even a Smalltalk cross-compiler [1] (http://www.pocketsmalltalk.com/) for the PalmPilot which runs on Windows (though it seems that development may have stagnated in recent years).

There is also a high-performance JITted modular Smalltalk implementation designed for scripting called S# (S-Sharp) which supports an extended dialect of Smalltalk.

More recently, Python and Ruby have reimplemented many of Smalltalk's ideas with more mainstream syntax.

Object-oriented programming

As in other object-oriented languages, the central concept in Smalltalk is that of an *object*. An object is always an *instance* of a *class*. Classes are "blueprints" that describe the properties and behavior of their instances. For example, a Window class would declare that windows have properties such as the label, the position and whether the window is visible or not. The class would also declare that instances support operations such as opening, closing, moving and hiding. Each particular Window object would have its own values of those properties, and each of them would be able to perform operations defined by its class.

A Smalltalk object can do exactly three things:

- Hold state (references to other objects).
- Receive a message from itself or another object.
- In the course of processing a message, send messages to itself or another object.

The state an object holds is always private to that object. Other objects can query or change that state only by sending requests (messages) to the object to do so. Any message can be sent to any object: when a message is received, the receiver determines whether that message is appropriate.

Smalltalk is a 'pure' OO language, meaning that unlike Java and C++ there is no difference between objects and primitive types. In Smalltalk, primitive values such as integers, booleans and characters are also objects, in the sense that they are instances of corresponding classes. A programmer can change these classes to add new behavior to their instances--for example, to implement new control structures--or even change their current behavior. This fact is summarised in the commonly heard phrase "In Smalltalk everything is an object".

Since everything is an object, classes themselves are also objects. Each class is an instance of the *metaclass* of that class. Metaclasses in turn are also objects, and are all instances of a class called 'Metaclass'. Code blocks (see below) are also objects.

Reflection

Smalltalk is a fully reflective system, implemented in itself. Smalltalk provides both structural and computational reflection. Smalltalk is a structurally reflective system whose stucture is defined by Smalltalk objects. The classes and methods that define the system are themselves objects and fully part of the system that they help define. The Smalltalk compiler compiles textual source codee into method objects, typically instances of CompiledMethod. The part of the class hierarchy that defines classes can add new classes to the system. The system is extended by running Smalltalk code that creates new classes and methods. In this way a Smalltalk system is a "living" system, carrying around the ability to extend itself at run-time.

Smalltalk also provides computational reflection, the ability to observe the computational state of the system. In implementations derived from the original Smalltalk-80 the current activation of a method is accessible as an object named via a keyword, thisContext. By sending messages to thisContext a method activation can ask questions like "who sent this message to me". These faclities make it possible to implement co-routines or Prolog-like back-tracking without modifying the virtual machine.

When an object is sent a message that it does not implement the virtual machine sends the object the doesNotUnderstand: message with a reification of the message as an argument. The message (another object, an instance of Message) contains the selector of the message and an Array of its arguments. In an interactive Smalltalk system the default implementation of doesNotUnderstand: is one that opens an error window reporting the error to the user. Through this and the reflective facilities the user can examine the context in which the error occurred, redefine the offending code, and continue, all within the system, using Smalltalk's reflective facilities.

But another important use of doesNotUnderstand: is intercession. One can create a class that does not define any methods other than doesNotUnderstand: and does not inherit from any other class. The instances of this class effectively understand no messages. So every time a message is sent to these instances they actually get sent doesNotUnderstand:, hence they intercede in the message sending process. Such objects are called proxies. By implementing doesNotUnderstand: appropriately one can create distributed systems where proxies forward messages across a network to other Smalltalk systems (a facility common in systems like CORBA and RMI but first pioneered in Smalltalk in the 1980s), persistent sysems where changes in state are written to a database and the like.

Syntax

Smalltalk syntax is rather minimalist, based on only a handful of declarations and reserved words. In fact, only five keywords are reserved in Smalltalk: **true**, **false**, **nil**, **self** and **super**. The only built-in language contructs are message sends, assignment, method return and literal syntax for some objects. The remainder of the language, including control structures for conditional evaluation and iteration, is implemented on top of those by the standard Smalltalk class library. (For performance reasons implementations may recognize and treat as special some of those messages; however, this is only an optimization, not hardwired into the language syntax).

Literals

The following examples illustrate the most common objects which can be written as literal values in Smalltalk methods.

Numbers. The following list illustrates some of the possibilities

```
42
-42
123.45
1.2345e2
2r10010010
```

```
16rA000
```

The last two entries are a binary and a hexadecimal number, respectively. The number before the 'r' is the radix or base. The base does not have to be a power of two; for example 36rSMALLTALK is a valid number (for the curious, equal to 80738163270632).

Characters are written by preceding them with a dollar sign:

```
$A
```

Strings are sequences of characters enclosed in single quotes:

```
'hello world'
```

To include a quote in a string, double it.

Two equal strings (strings are equal if they contain all the same characters) can be different objects residing in different places in memory. In addition to strings, Smalltalk has a class of character sequence objects called Symbol. Symbols are guaranteed to be unique--there can be no two equal symbols which are different objects. Because of that, symbols are very cheap to compare and are often used for language artifacts such as message selectors (see below).

Symbols are written as # followed by characters. For example:

```
#foo
```

Arrays:

```
#(1 2 3 4)
```

defines an array of four integers.

And last but not least, blocks

```
[... Some smalltalk code...]
```

Blocks are explained in detail further in the text.

Many Smalltalk dialects implement additional syntaxes for other objects, but the ones above are the bread and butter supported by all.

Variable declarations

The two kinds of variable commonly used in Smalltalk are instance variables and temporary variables. Other variables and related terminology depend on the particular implementation. For example, VisualWorks has class shared variables and namespace shared variables, while Squeak and many other implementations have class variables, pool variables and global variables.

Temporary variable declarations in Smalltalk are variables declared inside a method (see below). They are declared at the top of the method as names separated by spaces and enclosed by vertical bars. For example:

```
| index |
```

declares a temporary variable named index. Multiple variables may be declared within one set of bars:

```
| index vowels |
```

declares two variables: index and vowels.

Assignment

A variable is assigned a value via the ':=' syntax. So:

```
vowels := 'aeiou'
```

Assigns the string 'aeiou' to the previously declared vowels variable. The string is an object (a sequence of characters between single quotes is the syntax for literal strings), created by the compiler at compile time.

Messages

A message is the most fundamental language construct in Smalltalk. Even control structures are implemented as message sends. The following examples sends the message 'factorial' to number 42.

```
42 factorial
```

In this situation 42 is called the message *receiver*, while 'factorial' is the message *selector*. The receiver responds to the message by returning a value (presumably in this case a factorial of 42). Among other things, the result of the message can be assigned to a variable:

```
aRatherBigNumber := 42 factorial
```

"factorial" above is what is called a *unary message* because only one object, the receiver, is involved. Messages can carry additional objects as *arguments*, as follows:

```
2 raisedTo: 4
```

In this expression two objects are involved: 2 as the receiver and 4 as the message argument. The message result, or in Smalltalk parlance, *the answer* is supposed to be 16. Such messages are called *keyword messages*. A message can have more arguments, using the following syntax:

```
'hello world' indexOf: $l startingAt: 6
```

(which rather obviously answers the index of character 'l' in the receiver string, starting the search from index

6). The selector of this message is "indexOf:startingAt:", consisting of two pieces, or *keywords*.

Such interleaving of keywords and arguments greatly improves readability of code, since arguments are explained by their preceding keywords. For example an expression to create a rectangle using a C++ or Java-like syntax:

```
 new Rectangle(10, 20, 100, 200);
```

where it's unclear which argument is which--is it (left, top, right, bottom) or (left, right, top, bottom), or (left, top, width, height)?--in Smalltalk becomes:

```
Rectangle left: 10 top: 20: right: 100 bottom: 200
```

The receiver in this case is "Rectangle", a class, and the answer will be a new instance of the class with the specified parameters.

Finally, most of the special (non-alphabetic) characters can be used as what is called *binary messages*. These allow mathematical and logical operators to be written in their traditional form:

```
3 + 4
```

which sends the message "+" to the receiver 3 with 4 passed as the argument. Similarly,

```
3 > 4
```

is the message ">" sent to 3 with argument 4 (the answer of which will be false).

Notice, that the Smalltalk language itself does not imply the meaning of those operators. The outcome of the above is only defined by how the receiver of the message (in this case a Number instance) responds to messages "+" and ">".

A side effect of this mechanism is operator overloading. A message ">" can also be understood by other objects, allowing the use of expressions of the form "a > b" to compare them.

Expressions

An expression can include multiple message sends. In this case expressions are parsed according to a simple order of precedence. Unary messages have the highest precedence, followed by binary messages, followed by keyword messages. For example:

```
3 factorial + 4 factorial between: 10 and: 100
```

is evaluated as follows:

```
3 receives the message "factorial" and answers 6
4 receives the message "factorial" and answers 24
6 receives the message "+" with 24 as the argument and answers 30
30 receives the message "between:and:" with 10 and 100 as arguments and answers true
```

The answer of the last message send is the result of the entire expression.

Parentheses can alter the order of evaluation when needed. For example,

```
(3 factorial + 4) factorial between: 10 and: 100
```

will change the meaning so that the expression first computes "3 factorial + 4" yielding 10. That 10 then receives the second "factorial" message, yielding 3628800. 3628800 then receives "between:and:", answering false.

Note that because the meaning of binary messages is not hardwired into Smalltalk syntax, all of them are considered to have equal precedence and are evaluated simply from left to right. Because of that the meaning of Smalltalk expressions using binary messages can be different from their "traditional" interpretation:

```
3 + 4 * 5
```

is evaluated as "(3 + 4) * 5", producing 35.

Unary messages can be *chained* by writing them one after another:

```
3 factorial factorial log
```

which sends "factorial" to 3, then "factorial" to the result (6), then "log" to the result (720), producting the result 2.85733.

A series of expressions can be written as in the following (hypothetical) example, each ending with a period. This example first creates a new instance of class Window, stores it in a variable, and then sends two messages to it.

```
| window |
window := Window new.
window label: 'Hello'.
window open.
```

If a series of messages are sent to the same receiver as in the example above, they can also be written as a *cascade* with individual messages separated by semicolons:

```
Window new
  label: 'Hello';
  open
```

This rewrite of the earlier example as a single expression avoids the need to store the new window in a temporary variable. According to the usual precedence rules, the unary message "new" is sent first, and then "label:" and "open" are sent to the answer of "new".

Code blocks

A block of code can be expressed as a literal object. This is achieved with square brackets:

```
[ :params | <message-expressions> ]
```

Where :params is the list of parameters the code can take. This means that the Smalltalk code:

```
[:x | x + 1]
```

is equivalent to:

```
f(x) = x + 1
```

Technically, the resulting block object is a closure. It can (at any time) access the variables of its enclosing lexical scopes. Blocks are first class. That is, references to blocks can be passed as arguments, returned as values, or stored as a state, just like any other objects. Blocks can be asked to execute their code by sending them a "value"-message (with one argument for each parameter in the block).

The literal representation of blocks was an innovation which allowed certain code to be significantly more readable; it allowed algorithms involving iteration to be coded in a clear and concise way. Code that would typically be written with loops in some languages can be written concisely in Smalltalk using blocks, sometimes in a single line.

```
positiveAmounts := allAmounts select: [:amt | amt isPositive]
```

Note that this is very closely related to functional programming, wherein patterns of computation (here selection) are abstracted into higher-order functions. For example, the message select: on a Collection is equivalent to the higher-order function filter on an appropriate functor.

Control structures

Smalltalk control structures are notably absent from the language syntax. These are instead implemented as messages sent to objects. For example, conditional execution is implemented by sending the message ifTrue: to a Boolean object, with the block of code to be executed if and only if the Boolean is true as the message argument.

The following code demonstrates this:

```
result := a > b
    ifTrue:[ 'greater' ]
    ifFalse:[ 'less' ]
```

Blocks are also used to implement user-defined control structures, enumerators, visitors, pluggable behavior and many other patterns. For example:

```
| aString vowels |
aString := 'This is a string'.
vowels := aString select: [:aCharacter | aCharacter isVowel].
```

In the last line, the string is sent the message select: with an argument that is a code block literal. The code block literal will be used as a predicate function that should answer true if and only if an element of the String should be included in the Collection of characters that satisfy the test represented by the code block that is the argument to the "select:" message.

A String object responds to the "select:" message by iterating through its members (by sending itself the message "do:"), evaluating the selection block ("aBlock") once with each character it contains as the argument. When evaluated (by being sent the message "value: each"), the selection block (referenced by the parameter "aBlock", and defined by the block literal "[:aCharacter | aCharacter isVowel]"), answers a boolean, which is then sent "ifTrue:". If the boolean is the object true, the character is added to a string to be returned. Because the "select:" method is defined in the abstract class Collection, it can also be used like this:

```
| rectangles aPoint|
rectangles := OrderedCollection
  with: (Rectangle left: 0 right: 10 top: 100 bottom: 200)
  with: (Rectangle left: 10 right: 10 top: 110 bottom: 210).
aPoint := Point x: 20 y: 20.
collisions := rectangles select: [:aRect | aRect containsPoint: aPoint].
```

Classes

This is a stock class definition:

```
Object subclass: #MessagePublisher
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Smalltalk Examples'
```

Often, most of this definition will be filled in by the environment.

Methods

When an object receives a message, a method matching the message name is invoked. The following code defines a method publish, and so defines what will happen when this object receives the 'publish' message.

```
publish
    Transcript show: 'Hello, World!'
```

Note that the pairing of a message with a method is done by the object at runtime (while in many languages this is determined statically at compile time).

Instantiating classes

The following code:

```
MessagePublisher new
```

creates (and returns) a new instantiation of the MessagePublisher class. This is typically assigned to a variable:

```
publisher := MessagePublisher new
```

However, it is also possible to send a message to a temporary, anonymous object:

```
MessagePublisher new publish
```

Hello World example

In the following code, the message "show:" is sent to the object "Transcript" with the String literal 'Hello, world!' as its argument. Invocation of the "show:" method causes the characters of its argument (the String literal 'Hello, world!') to be displayed in the transcript ("console") window.

```
   Transcript show: 'Hello, world!'.
```

Note that a Transcript window would need to be open in order to see the results of this example.

Image-based persistence

Most popular programming systems separate program code (in the form of class definitions, functions or procedures) from program state (such as objects or other forms of application data). They load the program code when an application is started, and any previous application state has to be recreated explicitly from configuration files or other data sources.

Smalltalk systems, however, do not differentiate between application data (objects) and code (classes). In fact, classes are objects themselves. Therefore most Smalltalk systems store the entire application state (including both Class and non-Class objects) in an image file. The image can then be loaded by the Smalltalk interpreter to restore a Smalltalk system to a previous state.

Other languages that model application code as a form of data, such as Lisp, often use image-based persistence as well.

Smalltalk images are similar to core dumps and generally provide the same benefits, such as delayed or remote debugging with full access to the application state at the time of error.

Level of access

Everything in Smalltalk is available for modification from within a running program. This means that, for example, the IDE can be changed in a running system, without restarting it. In some implementations, the syntax of the language, or the garbage collection implementation can also be changed on the fly.

Just-in-time compilation

Smalltalk programs are usually compiled to bytecode, which is then interpreted by a virtual machine or dynamically translated into machine-native code. This mechanism has been adopted by languages such as Java and C#.

Implementations

- Ambrai Smalltalk, see http://www.ambrai.com/
- Bistro
- Dolphin Smalltalk, see http://www.object-arts.com/content/navigation/home.html

- F-Script
- GemStone/S, see http://www.gemstone.com/products/smalltalk/
- GNU Smalltalk, see http://www.gnu.org/software/smalltalk/smalltalk.html
- IBM VisualAge for Smalltalk, see http://www.ibm.com/software/awdtools/smalltalk/, http://www.instantiations.com/VAST/
- ObjectStudio, see Cincom Smalltalk website (http://smalltalk.cincom.com/), Wiki (http://ostudio.swiki.net/1), Cincom Smalltalk Blog (http://www.cincomsmalltalk.com/blog/blogView).
- OSVM a small Smalltalk for embedded devices, see http://www.esmertec.com/solutions/M2M/
- LSW Vision-Smalltalk, see http://www.lesser-software.com/lswvst.htm
- Pocket Smalltalk which runs on a Palm Pilot, see http://www.pocketsmalltalk.com/
- S#, see http://www.ssharp.org
- Smalltalk MT, see http://www.objectconnect.com/
- Smalltalk/X, see http://www.exept.de/exept/english/Smalltalk/frame_uebersicht.html
- Squeak, see http://www.squeak.org/
- StepTalk (Uses Smalltalk language on top of the Objective-C runtime)
- Strongtalk
- Susie: Scripting Using a Smalltalk Interpreter Engine see http://sourceforge.net/projects/susie/
- VisualWorks, see Cincom Smalltalk website (http://smalltalk.cincom.com/), Wiki (http://wiki.cs.uiuc.edu/VisualWorks), Cincom Smalltalk Blog (http://www.cincomsmalltalk.com/blog/blogView).
- Bits of History (http://home.netsurf.de/helge.horch/esug/index.html) -- a Smalltalk-76 implementation as a Java applet.

External links

Wikibooks has more about this subject:
Programming:Smalltalk

- Smalltalk.org (http://www.smalltalk.org/) Advocacy site.
- Why Smalltalk? (http://www.whysmalltalk.com/) Developer community.
- GoodStart (http://www.goodstart.com/) Advocacy site.
- Open Directory: Smalltalk (http://dmoz.org/Computers/Programming/Languages/Smalltalk/)
- The Early History of Smalltalk (http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html) by Alan C. Kay.
- Smalltalk-72 Instruction Manual (http://www.bitsavers.org/pdf/xerox/alto/Smalltalk72_Manual.pdf)
- Smalltalk information visualization tool (http://www.softcentral.com/informationspace/)
- ESUG (European Smalltalk Users Group) (http://www.esug.org/): A non-profit organization which gathers both industrial and academics. Has various Smalltalk promotion activities including a yearly event since 1993.
- Wiki con documentación de Smalltalk en español (http://swiki.agro.uba.ar/small_land/)
- Design Principles Behind Smalltalk (http://users.ipa.net/~dwighth/smalltalk/byte_aug81/design_principles_behind_smalltalk.html) by Dan Ingalls from the BYTE August 1981 Special Issue on Smalltalk
- The Smalltalk-76 Programming System: Design and Implementation (http://users.ipa.net/~dwighth/smalltalk/St76/Smalltalk76ProgrammingSystem.html) by Dan Ingalls.

Books

- Downloadable books about Smalltalk (http://www.iam.unibe.ch/~ducasse/FreeBooks.html) Permission obtained to make these books freely available.

Major programming languages (more) (edit)

Industrial: ABAP | Ada | AWK | Assembly | ColdFusion | C | C++ | C# | COBOL | Delphi | D | Eiffel | Fortran | Java | JavaScript | Limbo | Lua | Objective-C | Pascal | Perl | PHP | Python | RPG | Scheme | **Smalltalk** | SQL | Tcl |

Visual Basic | VB.NET | Visual FoxPro	
Academic: APL/J | Haskell | Lisp | Logo | ML | Prolog | Scheme	*Other:* ALGOL | BASIC | Clipper | Forth | Modula-2/Modula-3 | MUMPS | PL/I | Ruby | Simula

Retrieved from "http://en.wikipedia.org/wiki/Smalltalk"

Categories: Class-based programming languages | Dynamically-typed programming languages | Object-oriented programming languages | Programming languages