Smalltalk

Smalltalk was originally developed at the Xerox Palo Alto Research Center.

- Smalltalk-74. BitBlt graphics added.
- Smalltalk-76. Major redesign. Passed from the realm of a prog. lang. for children.
- Smalltalk-80. Dorado, Daisy, Dolphin impls.

Smalltalk was released to non-Xerox research labs.

- Tektronix.
- Apple.

Later, the designers of Smalltalk-80 formed ParcPlace Systems.

They went on to release ObjectWorks\Smalltalk, and then VisualWorks\Smalltalk.

Meanwhile, another dialect of Smalltalk was developed by Digitalk. Digitalk and ParcPlace merged in 1995.

IBM joined the market in 1995 with its own version of Smalltalk.

In 1997, ParcPlace changed its name to ObjectShare and abandoned—but later restarted—VW development.

In 1999, ObjectShare sold VisualWorks to Cincom.

Approximately the same time, the IBM Smalltalk Group moved to a location near the NCSU campus.
**Basic concepts**

[Liu, Ch. 1–3] In Smalltalk, “everything is an object,” for example,

- 
- 
- 
- 

Each object is a member of some *class*.

- Classes are a lot like types in other programming languages, but not exactly.

The classes are arranged in a hierarchy, with the class *Object* at the top.

- A class immediately above a class in the hierarchy is called its *superclass*.

- A class immediately below a class in the hierarchy is a *subclass* of it.

A member of a class is called an *instance* of that class.

- 2.718281828 is an instance of

It is important to understand the distinction between subclass and instance [Liu §2.3].

- Subclasses are typically *specializations* of their superclasses. Ask yourself, is an Integer a *kind of* Number? If so, it could be a good subclass of Number.

- Instances are not specializations, but rather have the same attributes and behavior.
Similarly, do not confuse \textit{specialization} (is-a) with \textit{aggregation} (has-a):

Objects interact by sending \textit{messages}.

- Messages are a lot like function and procedure calls in other languages.

  \begin{verbatim}
  checking withdraw: 100
  \end{verbatim}

Messages cause \textit{methods} to be invoked.

- A method is like a \textit{procedure or function in other languages}.

\texttt{rounded}

\begin{verbatim}
"Answer the integer nearest the receiver"
  self >= 0.0
ifTrue:  [^((self+0.5) truncated)
ifFalse: [^((self-0.5) truncated]
\end{verbatim}

(The operator “\(^\uparrow\)” means “return a value”.)

Notice that \texttt{if-then-else} is performed by sending a message.

Looping is also done that way, as we shall see.
Messages

All actions in a Smalltalk program are performed by sending messages, even simple arithmetic.

\[ 3 + 4 \]

sends the message “+” to 3 with the single argument 4.

The object to which the message is sent is called the receiver.

In general, a message has the syntax—

Syntax

With respect to syntax, there are three kinds of messages:

- Unary messages
- Binary messages
- Keyword messages

**Unary messages** have no arguments.
The name of a unary message is a character string beginning with a lower-case letter.

\[
0 \cos \\
#(a b c) \text{ printString} \\
\text{Transcript cr}
\]

**Binary messages** have a single argument.
The name of a binary message is either one or two special characters.

\[
1 + 2 \\
\text{cond1} & \text{cond2} \ | \ \text{alternative1} \\
9 \// 4 \\
'\text{abc}', '\text{def}' \\
12.34 @ 56.78
\]

**Keyword messages** have any number of arguments.
The name of a keyword message has the same form as the name of a unary message.

Each argument is preceded by a symbol followed with a colon.

```
myScore between: 90 and: 100
'string' at: 4 put: $0
255 printStringRadix: 2
```

The name of a keyword message must include the keyword names.

How many arguments are there in the first keyword message above?

What are their keywords?

What is the name of the message?

What are the names of the other messages?

**Parsing messages**

Generally, messages in Smalltalk are parsed left to right, except—

The following precedence order is obeyed:

- Parenthesized expressions
- Unary messages
- Binary messages
- Keyword messages

In what order is the following expression evaluated?

```
1 ln + 2 * 3.14159 sin // (a - b * c) abs min: 5
```

Messages are delimited by a period:

```
total := 0.
total := total + 1.
Transcript show: total printString.
```

Messages may be cascaded. More than one msg. to same receiver.
myAccount deposit: 35; withdraw 20;
   withdraw:10.
Transcript cr; cr; show: 'Hello world'.

**Literals**

Five kinds of objects can be referred to by literal expressions. Numbers in Smalltalk look very much the same as in other languages.

3
30.45
-14.0
13772
1.586e–3

Numbers may be expressed in any radix

- The radix precedes the letter “r”.
- The digits (in that radix) follow it.

8r153
16rAB
16r1e10
2r1110110110

A character literal is preceded by “$”.

$a $A $+ $1 $$

Strings:

'This is a string.'
'A literal string is delimited by single quotes.'
'Embedded quotes like this '' must be doubled.'

Symbols are objects that represent strings used for names in the system.

#thisIsASymbol
#AStringOfCharactersPrefixedByaPoundSign
#1
An array is the last kind of literal.

- An array is a sequence of other literals
- preceded by a pound signs and surrounded by parentheses.
- The other literals are separated by spaces.
- Embedded symbols and arrays are not preceded by pound signs.

```
# (1 2 3)
#(a literal array of symbols)
#(123 4.5 $a (abc (def)) 'hello')
```

A *comment* is any string of characters delimited by (double) quotes:

```
"Here's an embedded "" double quote"
```

A *block* is any Smalltalk code delimited by square brackets.

- It is analogous to a Lisp lambda function.
- It may be used as an argument to a message.

```
[Pen example]
[:x :y | Pen goto: x@y]
```

**Scoping of variables**

Smalltalk has six kinds of variables.

- *Global variables* are universally accessible.
  
  An example is Transcript, which refers to the system transcript.

- *Pool variables* are accessible to a set of classes, but not to the entire system.

  An example is TextConstants, which is accessible to different classes that manipulate text.
• **Class variables** are accessible to all instances of the class in which they reside, and its subclasses.

An example is a CardDeck class that has an InitialCardDeck class variable

• **Class instance variables** are accessible to all instances of the class in which they reside, but not to its subclasses. (See Liu, pp. 230–231.)

• **Instance variables** are accessible to only one specific instance of a class.

In the CardDeck example, the variable cards is used to hold the current configuration of the deck.

• **Temporary variables** are accessible only to a specific activation of a given method.

In addition to these six kinds of variables, there are a few *pseudo-variables*:

  self refers to the receiver itself.
  super refers to the superclass of the current object.

The above pseudo-variables are accessible only to the instance.

The following pseudo-variables are universally accessible.

  nil       true false

*Smalltalk naming conventions*: Identifiers begin with a letter, consist of letters and digits.

Multiple-word identifiers

  areWrittenLikeThisExample
These begin with an uppercase letter:

- Class names
- Class-variable names
- Pool-variable names
- Global-variable names

These begin with a lowercase letter:

- Instance-variable names
- Temporary-variable names
- Message selectors
- Psuedo-variable names

**Flow-control messages**

Flow control in Smalltalk is accomplished by sending messages. There are two kinds of these messages:

- **if-then** messages
- **loop** messages

These are *keyword messages* that take *blocks* as arguments.

**if-then messages**: These messages are sent to receiver of class *boolean*.

Only *true* and *false* understand these messages.

- **ifTrue**: a**Block**
  
  \[
  1 = 1 \text{ ifTrue:} \text{[Transcript cr; show: 'True.']} \]

- **ifFalse**: a**Block**
  
  \[
  \text{num \prime = 0 ifFalse:} \text{[primes add: num]} \]

- **ifTrue**: a**Block1 ifFalse**: a**Block2

  \[
  \text{self passing ifTrue:} \text{[self takeItEasy]} \text{ ifFalse:} \text{[self studyHarder]} \]
• \texttt{ifFalse: \\ aBlock1 ifTrue: aBlock2}

\textit{Loop messages}: There are four kinds of messages that control looping.

• Repetition
• Indexed loops
• \textbf{do-while} loops
• Collection enumeration

\textit{Repetition}: \texttt{timesRepeat: aBlock}

\begin{verbatim}
self timesRepeat:
  [result := result * 2]
self numberOfDependents timesRepeat:
  [taxIncome deduct: 2750]
360 timesRepeat:
  [thePen turn: 1; go 10]
\end{verbatim}

Here is an example of Smalltalk code that converts all consonants to lower case and all vowels to upper case:

\begin{verbatim}
|string index c|
string := 'Now is the time'.
index := 1.
string size timesRepeat: [c := string at: index.
string at: index put:
  (c isVowel
    ifTrue: [c asUpperCase
    ifFalse: [c asLowerCase]].
  index := index + 1].
\end{verbatim}

\textit{Indexed loops}: Two messages—

• \texttt{start to: stop do: aBlock}

\begin{verbatim}
1 to: 10 do: [:i | sum := sum + i]
\end{verbatim}
In these examples, the "i", etc. is called a block argument. The loop is evaluated once for each value of the block argument.

- start to: stop by: increment
do: ablock

1 to: 10 by: 2 do: ...
1 to: −10 by: −2 do: ...
0.001 to: 1 by: 0.001 do: ...

do-while loops are written with four messages:

- aBlock whileTrue: aBlock
  [x < 100] whileTrue: [x := x + 1]

- aBlock1 whileFalse: aBlock2
  [x > 100] whileFalse: [x := x + 1]

- aBlock whileTrue
  [Display reverse. Sensor blueButtonPressed] whileTrue

- aBlock whileFalse
  [self parts add: Part new. self confirm: 'Finished adding parts?'] whileFalse

Collection enumeration: A block is executed once for each item within a collection.

This enumeration can also be performed on an instance of any subclass of a collection.
• **do: aBlock**

```smalltalk
#(a 1 $d (a b c) 'hello' 5.32) do:[:element |
  Transcript cr; show: element class
  printString]
```

• **reverseDo: aBlock**

```smalltalk
alphabetizedNames reverseDo: [:name |
  reverseAlphOrder add: name]
```

• **collect: aBlock**

This message creates a collection of the values produced by executing the block for each element of the original collection.

```smalltalk
#(2 4 6 8) collect:[:i | i*i]
```

• **select: aBlock**

This message creates a collection of the elements that satisfy some (boolean) criterion.

```smalltalk
employees select:
  [:e | e salary > 50000]
```

• **detect: aBlock ifNone: anotherBlock**

```smalltalk
samples detect:
  [:each | each isContaminated]
  ifNone:[Transcript show:
    'All samples are pure.']
```

**Assignments**

A Smalltalk statement may begin with an assignment prefix—

```smalltalk
i := 1
```

More than one assignment prefix may be used:

```smalltalk
i := j := 1
```
Smalltalk syntax summary:

• A Smalltalk identifier is a string of alphanumerics beginning with an alphabetic.

• Literals are like many other languages, except—
  ◦ character constants are prefixed with a “$”.
  ◦ array literals are prefixed with a “#”.
  ◦ a numeric literal can be prefixed with a radix (e.g., 2, 8, 16).

• There are five kinds of variables, each of which have different scopes
  ◦ Global variables, pool variables, and class variables are shared among different objects
  ◦ Instance variables and temporary variables are accessible only to a single object.

• All computing is done by sending messages.
  ◦ There are unary messages, binary messages and keyword messages.
  ◦ Several specific messages are used for conditionals and various forms of loops.