

Data Abstraction and Object Orientation

9.6.1 The Object Model of Smalltalk

Smalltalk is heavily integrated into its programming environment. In fact, unlike all of the other languages mentioned in this book, a Smalltalk program does not consist of a simple sequence of characters. Rather, Smalltalk programs are meant to be viewed within the *browser* of a Smalltalk implementation, where font changes and screen position can be used to differentiate among various parts of a given program unit. Together with the contemporaneous Interlisp and Pilot/Mesa projects at PARC, the Smalltalk group shares credit for developing the now ubiquitous concepts of bit-mapped screens, windows, menus, and mice.

Smalltalk uses an untyped reference model for all variables. Every variable refers to an object, but the class of the object need not be statically known. As described in Section 9.3.1, every Smalltalk object is an instance of a class descended from a single base class named `Object`. All data are contained in objects. The most trivial of these are simple immutable objects such as `true` (of class `Boolean`) and `3` (of class `Integer`).

EXAMPLE 9.64

Operations as messages in Smalltalk

Operations are all conceptualized as *messages* sent to objects. The expression `3 + 4`, for example, indicates sending a `+` message to the (immutable) object `3`, with a reference to the object `4` as argument. In response to this message, the object `3` creates and returns a reference to the (immutable) object `7`. Similarly, the expression `a + b`, where `a` and `b` are variables, indicates sending a `+` message to the object referred to by `a`, with the reference in `b` as argument. If `a` happens to refer to `3` and `b` refers to `4`, the effect will be the same as it was in the case of the constants. ■

EXAMPLE 9.65

Mixfix messages

As described in Section 6.1, multiargument messages have multiword (“mixfix”) names. Each word ends with a colon; each argument follows a word. The expression

```
myBox displayOn: myScreen at: location
```

EXAMPLE 9.66

Selection as an `ifTrue:`
`ifFalse:` message

sends a `displayOn:` `at:` message to the object referred to by variable `myBox`, with the objects referred to by `myScreen` and `location` as arguments. ■

Even control flow in Smalltalk is conceptualized as messages. Consider the selection construct:

```
n < 0
  ifTrue: [abs <- n negated]
  ifFalse: [abs <- n]
```

This code begins by sending a `< 0` message (a `<` message with 0 as argument) to the object referred to by `n`. In response to this message, the object referred to by `n` will return a reference to one of two immutable objects: `true` or `false`. This reference becomes the value of the `n < 0` expression.

Smalltalk evaluates expressions left-to-right without precedence or associativity. The value of `n < 0` therefore becomes the recipient of an `ifTrue: ifFalse:` message. This message has two arguments, each of which is a *block*. A block in Smalltalk is a fragment of code enclosed in brackets. It is an immutable object, with semantics roughly comparable to those of a lambda expression in Lisp. To execute a block we send it a `value` message.

When sent an `ifTrue: ifFalse:` message, the immutable object `true` sends a `value` message to its first argument (which had better be a block) and then returns the result. The object `false`, on the other hand, in response to the same message, sends a `value` message to its second argument (the block that followed `ifFalse:`). The left arrow (`<-`) in each block is the assignment operator. Assignment is not a message; it is a side effect of evaluation of the right-hand side. As in expression-based languages such as Algol 68, the value of an assignment expression is the value of the right-hand side. The overall value of our selection expression will be the value of one of the blocks, namely a reference to `n` or to its additive inverse, whichever is non-negative. For the sake of convenience, Boolean objects in Smalltalk also implement `ifTrue:`, `ifFalse:`, and `ifFalse: ifTrue:` methods. ■

EXAMPLE 9.67

Iterating with messages

Iteration is modeled in a similar fashion. For enumeration-controlled loops, class `Integer` implements `timesRepeat:` and `to: by: do:` methods:

```
pow <- 1.
10 timesRepeat:
  [pow <- pow * n]

sum <- 0.
1 to: 100 by: 2 do:
  [:i | sum <- sum + (a at: i)]
```

The first of these code fragments calculates n^{10} . In response to a `timesRepeat:` message, the integer k sends a `value` message to the argument (a block) k times. The second code fragment sums the odd-indexed elements of the array referred to by `a`. In response to a `to: by: do:` message, the integer k behaves as one might expect: it sends a `value:` message to its third argument (a block) $[(t - k + b)/b]$

times, where t is the first argument and b is the second argument. Note the colon at the end of `value:`. The plain `value:` message is unary; the `value:` message has an argument; it is understood by blocks that have a (single) formal parameter. In our loop example, the integer 1 sends the messages `value: 1`, `value: 3`, `value: 5`, and so on to the block `[:i | sum <- sum + (a at: i)]`. The `:i |` at the beginning of the block is its formal parameter. The `at:` message is understood by arrays. For iteration with a step size of one, integers also provide a `to: do: method`. ■

EXAMPLE 9.68

Blocks as closures

```
b <- [n <- n + 1].           "b is now a closure"
c <- [:i | n <- n + i].      "so is c"
...
b value.                   "increment n by 1"
c value: 3.                 "increment n by 3"
```

A block with two parameters expects a `value: value:` message. A block with j parameters expects a message whose name consists of the word `value:` repeated j times. Comments in Smalltalk are double-quoted (strings are single-quoted). ■

EXAMPLE 9.69

Logical looping with messages

For logically controlled loops, Smalltalk relies on the `whileTrue:` message, understood by blocks:

```
tail <- myList.
[tail next ~~ nil]
  whileTrue: [tail <- tail next]
```

This code sets `tail` to the final element of `myList`. The double-tilde (`~~`) operator means “does not refer to the same object as.” The method `next` is assumed to return a reference to the element following its recipient. In response to a `whileTrue:` message, a block sends itself a `value` message. If the result of that message is a reference to `true`, the block sends a `value` message to the argument of the original message and repeats. Blocks also implement a `whileFalse:` method. ■

The blocks of Smalltalk allow the programmer to construct almost arbitrary control-flow constructs. Because of their simple syntax, Smalltalk blocks are even easier to manipulate than the lambda expressions of Lisp. In effect, a `to: by: do:` message turns iteration “inside out,” making the body of the loop a simple message argument that can be executed (by sending it a `value` message) from within the body of the `to: by: do:` method. Smalltalk programmers can define similar methods for other container classes, obtaining all the power of iterators (Section 6.5.3) and much of the power of `call_with_current_continuation` (Section 8.5.3):

EXAMPLE 9.70

Defining control abstractions

```
myTree inorderDo: [:node | whatever ]
```

It is worth noting that the uniform object model of computation in Smalltalk does not necessarily imply a uniform implementation. Just as Clu implementations implement built-in immutable objects as values, despite their reference

semantics (Section 6.1.2), a Smalltalk implementation is likely to use the usual machine instructions for computer arithmetic, rather than actually sending messages to integers. In a similar vein, the most common control-flow constructs (`ifTrue: ifFalse: to: by: do:`, `whileTrue:`, etc.) are likely to be recognized by a Smalltalk interpreter, and implemented with special, faster code.

We end this subsection by observing that recursion works at least as well in Smalltalk as it does in other imperative languages.

The following is a recursive implementation of Euclid's algorithm:

EXAMPLE 9.71

Recursion in Smalltalk

```
gcd: other                                "other is a formal parameter"
    (self = other)
        ifTrue: [↑self].                    "end condition"
    (self < other)
        ifTrue:  [↑self gcd: (other - self)]    "recurse"
        ifFalse: [↑other gcd: (self - other)]    "recurse"
```

The up-arrow (↑) symbol is comparable to the `return` of C or Algol 68. The keyword `self` is comparable to `this` in C++. We have shown the code in mixed fonts, much as it would appear in a Smalltalk browser. The header of the method is identified by boldface type. ■

✓ **CHECK YOUR UNDERSTANDING**

50. Name the three projects at Xerox PARC in the 1970s that pioneered modern GUI-based personal computers.
51. Explain the concept of a *message* in Smalltalk.
52. How does Smalltalk indicate multiple message arguments?
53. What is a *block* in Smalltalk? What mechanism does it resemble in Lisp?
54. Give three examples of how Smalltalk models control flow as message evaluation.
55. Explain how type checking works in Smalltalk.