VI

Technical Characteristics of Smalltalk-76

Since the first Smalltalk was outlined in 1972, we have produced four implementations of significance (over 1000 hours of use each) on several different hardware systems. To va degrees, the language has supported the object-oriented framework, always making it poss us to explore the nature of a useful and usable personal computing environment. The lan kernel that is defined in this chapter was proposed in 1976 after several years of desig implementation. We have thus termed the language, Smalltalk-76. We have separated the description of this particular language kernel, as provided in this chapter, from those and later Smalltalks: the history of implementations can be found in Chapter X; discus user interface issues is provided in Part 2; all of Part 3 is devoted to the future.

System Layering: A Review

The system can be viewed as consisting of several layers, each built on the lower layers bottom is the *implementation machine*, a set of interconnected physical devices. On this is bu a *kernel system* that provides the essential functions of Smalltalk: the creation and mainte of objects and their interaction by exchanging messages. The kernel of Smalltalk is an system made up of objects organized into classes. It includes objects called *classes* that and maintain other objects, objects called *methods* that describe the interaction of object objects called *activations* that perform the interaction described by methods. A method is a sequence of message descriptions and/or descriptions of transformations of state informa Note that we use the word "method" to avoid connotations associated with the more tradit

"program" or "procedure". These objects--classes, methods, and activations--make it pos bootstrap the behavior of other objects since these other objects can be created by a cl then can have any kind of behavior as the result of an activation performing a method th describes that behavior.

The next level of the system is a *basic system* that provides certain conveniences for intera with objects and for describing new kinds of objects. In specifying the basic system, w a syntax for expressions and for the ways in which methods are built from expressions. basic Smalltalk-76 system includes a compiler and graphical facilities for handling user interactions. Additional objects provided in the basic system include class Dictionary; Number, with its subclasses Integer and Float; and class Array, with its subclasses Stri UniqueString, and Vector; and, to support user interaction on a display screen, the clas Point, Rectangle, and UserView. A number of classes exist in the basic Smalltalk-76 system in order to support the compiler; they are presented in the next chapter on comp methods (Chapter VII, Encoding). Descriptions of classes for user interactions are give chapters in Part 2.

Program execution was dealt with in Chapters II and IV; we concentrate here on the kern basic Smalltalk-76 systems. In the next sections, we provide a definition of the syntax Smalltalk-76 with simple examples. More extensive examples are found in Chapter ViII.

The Kernel System

As described above, the kernel system consists of classes, instances, methods, and activ

An Example Method

A class only understands a message that has been included in its message dictionary, i.e message for which a corresponding method has been specified. For example, class Diction employs the following method to respond to lookup messages.

```
lookup: name | x

        [x _ self find: name    [  values   x]    false]
```

Smalltalk-76 is easier to discuss if the special characters are pronounced as follows.

: (silent)
| with temporary variables
[begin
] end
_ gets
then only
return value
subscripted by or sub

A method begins with a *message pattern*, in this case, lookup: name. The colon character (:) which is not pronounced, indicates that an argument follows, i.e., name. There is also temporary variable called x that is not an argument but is used within the method to ass execution.

The body of a method is a *block* of Smalltalk *statements* that provides a procedural implementation of the method. Expressions in Smalltalk are usually embedded in methods i class definitions.

The body of the method above consists of a single *conditional statement*.

```
x _ self find: name   [  values   x]    false
```

A conditional statement has three parts: a condition followed by a right arrow, ; a t alternative enclosed in square brackets; and a false alternative. In the above example parts are:

x _ self find: name

This is the condition part.

In this statement, the message find: name is sent to the dictionary itself (named using the pseudo-variable self). We expect this message to return the location of name within the field objects, i.e., an integer subscript i such that objects i name. If there is no such i, it returns the special constant false.

The value returned is assigned by the symbol _ to the temporary variable x. An assignment statement has a value, which is the value assigned.

The condition of this conditional statement is considered false if the value of th
assignment was false, and is considered true otherwise.

values x

This is the true alternative part.

This statement is executed if and only if the condition is not false. It sends th
message x to the field values. We expect that this message returns the xth elem
of values. The symbol causes the result to be returned as the value of the curr
method.

false

This is the false alternative part.

This statement is executed if and only if the condition is false. It returns fals
value of the current method.

The Pseudo-variable super

The pseudo-variable super is, like self, provided automatically to every method. A mess
sent to super goes to the currently running instance with the caveat that the message-di
lookup starts at the superclass of the class whose method is being performed, instead of
class of self.

The Basic System

Creating Objects: the Message new

A new instance of a class is created by passing its class the message new. The class al
the property dictionary for the new instance, initializes each value in the dictionary t
constant nil, and returns the object. Usually the new object is then passed additional
to initialize it fully.

A new class is created by creating a new instance of class Class, i.e., by passing the c
the message new. This message is followed by messages informing the new class of the na
of its title, its fields (*instance variables*), names of properties shared by all members of the

(*class variables*), names of any variables to be shared with other classes (*pool variables*), and references to a superclass.

Suppose we create a new class whose name is Customer as a subclass of Dictionary. It wi add instance variables name, address, telephone as the identication information for the dictionary; and one class variable, instCount, which will be the tally of all instances Customer ever. (Its superclass, Dictionary, you will recall, has one field, values; its superclass HashSet has one field, objects.) In order to create this new class, we send C a message.

```
Class   new title: 'Customer'
            subclassOf: Dictionary
            fields: 'name address telephone'
            declare: 'instCount'
```

Now a customer may be created by the statement

```
Customer new
```

The statement Customer new results in a new instance of the class. Suppose in the messa dictionary of the class Customer there is a message pattern init. Then a new instance m be passed an additional message requesting that the instCount be updated.

```
Customer new init
```

In Smalltalk-76, special consideration is given to this need to initialize instance vari modify class variables whenever a new instance is created. The following expression is equivalent to the one above

```
Customer init
```

Suppose we create two instances of class Customer which we will call A and B. Each instance contains a property dictionary such that if A changes its address field, the ad field for B does not change. Both A and B refer to the same class variable(s) so that if increases the value of instCount, then B will also be affected by that change.

Modifying the Message Dictionary of a Class

A class is told to associate a method with a message pattern by sending it the message understands: code classified: heading. Various designs for the user interface to the Smalltalk programming environment have provided different surface methods for specifying message dictionary of a class. In Chapter () on user aids, we describe a *Browser* design w provides a text editor and templates for creating classes and adding message patterns wi corresponding methods; in Chapter (), we describe a template language as the surface sy In each case, however, the class Class must ultimately be sent the message understands: code classified: heading informing it of the message pattern and its method.

Activations

Each method can declare *temporary variables* that can be used during the execution of the method. These variables are destroyed automatically when the method execution is termin The syntax for defining methods in Smalltalk-76 is detailed in a later section.

If we now send a message such as cust growby: 5, a new object that we call an Activation created. This new object contains a dictionary for storing the names and values of the temporary variables of the method that is currently being executed. Smalltalk-76 system supports four kinds of object variables in the creation and use of objects: class, pool and temporary (or method).

Object Life

No facilities are provided in Smalltalk-76 for explicit deallocation. An object is dest automatically when no reference to it exists. Thus, the programmer is generally freed f concern with deallocation. However, if an instance points to itself, or if it is part o structure, then Smaltlalk will never realize that it can be deallocated. Therefore, if structure includes cycles or back pointers, then when it is no longer needed, the progra must explicitly remove the pointers (typically by changing them to the constant nil).

The Basic System: Syntax

We had several goals in mind when we designed the syntax of Smalltalk-76. To begin with
had to be compilable (the previous Smalltalks were not). Of course it had to support th
of sending messages to objects. Our previous experience with Smalltalk and other system
the following additional considerations:

1. We liked the naturalness of infix expressions like 3+4.
2. We liked descriptive names for user-defined messages.
3. We liked the notion of keyword parameters to procedures rather than depending on t
   programmer's memorizing orderings for each argument.
4. We disliked parentheses because assuring proper matching can become frustrating.
5. We liked the notion of symmetric read/write messages.
6. We wanted to allow a series of messages to the same object to be sent concisely.
7. We wanted the syntax to include control, and to allow the user to define his own c
   messages.

Given these desires, the syntax which resulted is mostly a case of form following functi
this section we will first treat the syntax for expressions, and then describe the way i
methods are built from expressions.

Identifiers and Constants

In Smalltalk, an identifier can be any sequence of letters and digits which begins with a letter
By convention multiple word identifiers use capitals at the word breaks; variables for
definitions begin with a capital letter. Here are some examples.

```
a       r2d2       lastPictureShow       FSRectangle
```

Smalltalk-76 recognizes constant numbers, strings, names, and lists (as will be defined
are instances of class Number, String, UniqueString, and Vector, respectively).

A constant number is written as an unbroken sequence of digits. If the number is negative,
preceded by a "high minus" sign: . If the first digit is 0, the rest are in octal radi

```
0   6  32767   32766   0377   0177777   0100000
```

The constant 0940 is an incorrect number.

A constant floating point number is written as a decimal-radix number constant immediately

followed by a decimal point (a period) and one or more decimal digits. After the last d
be an exponent of the form e followed by a decimal-radix number constant.

0.0 3.14159 32766.32767e 32766

Incorrect floating point numbers include

0. 6. .31415927e1

A constant string is written as an arbitrary sequence of characters enclosed in apostrophes.
include an apostrophe in the string, it is necessary to write two in a row.

'' 'a' 'it costs $4.50' 'They said, ''Yes!'''

It is not correct to write

''' 'They said, 'Yes!''

A constant name is either a sequence of letters, digits, and colons not starting with a dig
any other single character except a parenthesis. It is preceded by the symbol unless
embedded in a constant list.

+ , Help printon:

Incorrect constant names include

) 12

The difference between a constant string and a constant name is that no two instances of
contain the same characters. They are similar to *atoms* in the language LISP.

A constant list is written as an arbitrary sequence of constants enclosed in parentheses. I
preceded by a unless it is embedded in another constant list.

() (0 6 32767) ((14 Help) 'arbitrary text')

Incorrect constant lists include

```
(14 Help)  (1 2 (3 4]
```

There are a few predefined standard constants in Smalltalk; they are the only instances of class Object:

```
nil    It is the default initial value for an identifier.

false  Anything else is effectively "true".

true   It is useful as a "true".
```

To send a message, one composes an expression which has a receiver part (itself an expre and a message part.

```
receiver  message
```

As in other programming languages, an expression can be *evaluated* according to certain rule order to yield a *value*. The value of every Smalltalk expression is an object (or, more pr a reference to an object). The receiver part is something that evaluates to the object receive the message. The message part consists of a *message name* or *selector* followed by zer or more parameters. The message name can be any constant name.

There are three basic forms of a message: *unary*, *binary*, and *keyword*. A unary message contains no parameters, a binary message contains one parameter, and a keyword message contains one or more parameters. The *valence* of a message is equal to the number of parameters it requires.

Unary Messages

The name of a unary message is called a *unary selector*; syntactically, it is simply an ident

```
x    center    next    show    new
```

Examples of using these message parts in an expression are

```
pt x    rect center    strm next    rect show    Rectangle new
```

The convention of a left-to-right parse means that a sequence of simple messages such as

```
angle asFloat asRadians cos
```

is interpreted as:

```
((angle asFloat) asRadians) cos
```

that is, (1) send the unary message asFloat to the object identified as angle, obtaining number; (2) send asRadians to that number, performing a degree-to-radian conversion, an finally, (3) send cos to that result to obtain the cosine.

Binary or Infix Messages

To allow infix expressions like 3 + 4 and a < b, we define infix messages as those whose selectors are single non-alphabetic characters such as + - < = > and . (Note th last character, , is typically used to denote subscripting.) We adopted the convention first term of an infix expression is the receiver; the message consists of the infix op valence-1 selector, with the second term as its parameter. Thus a < b means that a will the message < with parameter b. One can see how to compile this in a perfectly reasonab but there is need of a further convention to say what 2 * a + 1 means. Since users woul adding their own infix operators, precedence would be too confusing here, and we establi that the parse would be left-to-right. Thus 2 * a + 1 means (2 * a) + 1.

Keyword Messages

In some conventional programming languages, the notion of passing parameters by keyword allows procedure calls of the form

```
call draw (thickness=2, angle=45, length=1.414*base)
```

This saves the programmer from having to know the order in which the procedure expects t parameters. It can also provide for defaulting (for instance, in the example, a fourth color, might be expected; if omitted, it would default to some color such as black). T approach has been used in several applications systems, and also in the IBM's Job Contro Language [reference]. We had been looking for a way to represent messages with multiple parameters, and we thought that by stringing all the keywords together, the agglutenatio represent a selector of multiple valence. For example:

```
pen thickness: 2 angle: 45 length: 1.414*base
```

Here pen is the receiver. It receives the message thickness:angle:length: which has a valence of 3. This form provides the readability of keywords, and is also free of the p and commas used in the conventional form. Leaving the matter of defaults aside, this te has another nice property: the valence of the selector is manifested by the selector it when a selector has been found in a message dictionary, the associated method is guarant be expecting the right number of arguments.

A keyword, then, is simply an identifier with a trailing colon. For example

```
deleteChars:  to:    by:    from:    paint:
```

The actual selector in the keyword message is the concatenation of the keywords, in orde example, in the expression

```
1 to: 10 by: 2
```

the selector is to:by:, its valence is 2, and the parameters are 10 and 2. Here, the me sent to the number 1, resulting in an ordered collection of numbers: 1, 3, 5, 7, 9.

> There are two aspects to the keyword organization which we have not actually pursued, although we did plan for them in the design. The first is the possibility of sorting the agglutenated parts in the compiler, so that they can be used in any order by the caller. One of the reasons is that much of our access to the system is done by selecting choices in a list of alphabetized messages, and this gets difficult if the selector parts get reordered.
>
> The other possibility which we have not implemented is that of furnishing defaults for omitted keywords. This we plan to handle in one of several ways, for example, by a procedure which intercepts the case of a message which is not recognized by the receiver. Before announcing an error, this procedure would look in the receiver's message dictionary for any other selectors which included the parts of the unrecognized selector. If so, it would prompt the user for a value to use for each of the missing parameters, and then compile these into a definition for the short version which then calls the full version.

Precedence

In order to allow complex expressions to be assembled without needing many parentheses t determine grouping, we assigned a different level of precedence to each of the three mes types. It seemed natural for unary messages to be sent first, then infix, and finally k messages. This meant that arithmetic expressions could be embedded in keyword messages

without needing parentheses; also, simple expressions such as source next, center x, ve
length could appear in arithmetic expressions without needing parentheses. Within a giv
level of precedence, evaluation runs from left to right.

Keyword messages can not be written consecutively, or else they would concatenate into a
selector with a longer name. Thus,

```
(a min: b) max: c
```

and

```
a min: (b max: c)
```

are quite different from

```
a min: b max: c
```

The first two expressions would invoke the selectors min: and max:, while the latter wou
invoke the single selector min:max:.

From the discussion above, the reader should be able to parse the following expression w
returns true only if the rectangle r1 is equal to the rectangle r2.

```
r1 origin = r2 origin and: r1 corner = r2 corner
```

This is interpreted by first sending origin to r1 obtaining a number n1, then sending or
r2 to obtain a second number n2, and then sending = to n1 with n2 as its parameter. The
result will be true or false depending on whether the two origins are equal. The same p
takes place on the right with the corners, and we are left with

```
<term> and: <term>
```

which produces the final result to be returned. The introduction of precedence between t
various message categories has been a success in reducing the need for parentheses to a
reasonable level.

Cascading Messages

Another extension to the message syntax allows for sending a series of messages to the s

receiver as in

```
receiver message1; message2; message3.
```

For example

```
figure erase; moveto: dest; show.

(artist pen) go: 100; turn: 90; go: 30.
```

Here the semicolon separates what are referred to as *cascaded messages* to the receiver. The messages are sent in sequence from left to right. In the first example, the receiver is receives three messages, in order: first, erase; second, moveto: dest; and third, sho The receiver in the second example is the result of sending the message pen to artist. that the parentheses are recommended in order to unambiguously define the receiver of th cascaded messages. The value of a series of messages is the response from its last mess however, a cascaded messge is not an expression, it is a statement, so, to use the value put it in brackets.

Read/Write Symmetry

As in most programming languages, there are two forms of variable access for reading and writing. One uses the identifier to read, and the identifier followed by a left arrow t

```
extent          extent _ 10
```

The identifier (extent) takes on the value of the expression (10). Thus we have a simpl assignment statement. Its value is the value of the assigned expression.

There are cases where message access comes in read/write pairs, such as:

object	read	write
ith part of a	a i	a i _ 12
next item in strm	strm next	strm next _ char
fifth field of x	x field: 5	x field: 5 _ strm next

In order to support this symmetry in a uniform way, the syntax allows any message to be extended by a single terminal store-part consisting of a left arrow and an accompanying parameter. The left arrow has a right precedence lower than any selector, so that any e

may follow it. The extension results in a totally new message which has a left arrow as character and has a valence greater by 1. The examples above illustrate extension appli each of the basic message categories. In particular, the new selectors formed are _, n and field:_, respectively. In the third example, we see that the expression strm next provides the object that is written into the fifth field of x.

The formation of a selector using the left arrow notation results in a message; of cour message only has meaning if it is found in the message dictionary of the receiver. For it is possible to have an expression such as

```
1+3 _ 5
```

The object 1 is sent the message +_ which has a valence of 2; the two parameters are 3 Unless the message dictionary for an integer includes the selector +_, this message has meaning.

Order of Evaluation

Besides the left-to-right rule for the order in which expressions of equal precedence ar evaluated, there is an additional degree of freedom in the order in which the parameters multivalent message are evaluated. This order is undefined in the Smalltalk-76 syntax, are cautioned against ever writing code which depends on it (i.e., which has side effect order is of significance to compilers and interpreters, and we wanted to reserve this de flexibility for future implementors.

Examples

<time for some examples that help the reader figure out precedence, message syntax, order of evaluation--they should be real--they should be runnable on the current system--but use message protocols ala chapter 3 until we define syntax for them in this chapter--will the Customer and BankAccount work?>

Statements, Blocks, and Control

Evaluation in Smalltalk is sequenced by use of *statements* and *blocks.* Any expression can serve as a statement. A block is a sequence of statements separated by periods and enclosed i of square brackets. As an example, take

```
[v _ dict lookup: 'twelve'. w _ dict lookup: 'six'. v+w]
```

in which v gets 12, w gets 6, and the value of the block is the value of v+w, or 18.

If a block ends with a period (i.e., a terminating expression is omitted), nil becomes i
In these cases, the block is executed for effect rather than value. If a return stateme
expression) is executed in the block, then the block terminates and its value is that of
terminating expression.

Conditionals

The value of a statement may be treated as a true-or-false condition to choose between t
alternative paths of execution. Such a conditional statement is of the form

```
expression   block.
```

For example,

```
x < y   [x] .
```

and

```
rect has: pt   [rect growby: 20] .
```

If the value of the expression is not false then the block is executed and its value bec
value of the *enclosing block,* whose execution thereby terminates. Otherwise, the block is
skipped.

The equivalent of the Algol

```
  if ca then sa else if cb then sb;
```

is obtained by

```
  [ca   [sa] cb   [sb]] .
```

Note that, in the above example, if both ca and cb are false, then the value of the bloc
nil. Syntactically, the first alternative of a conditional is a bracketed block, and th
alternative is everything following that up to the end of the block in which the conditi

appears. It follows that a conditional must always be the last statement in a block, an
of the executed alternative becomes the value of that block. An example is

```
z _ [x<y   [x] y]
```

which first determines if x is less than y and if so, assigns the value x to z; otherwi
y is stored in z.

The equivalent of the Algol

```
v _ if ca then sa else if cb then sb else sc;
```

is obtained by

```
v _ [ca   [sa] cb   [sb] sc] .
```

In the above example, if both ca and cb are false, then the value of the block, and ther
v, is sc.

One of the major uses for blocks in Smalltalk-76 is as a context for a special condition
construct. An example is

```
min _ [x < y   [x] y] .
```

If the value preceding is true, then control enters the following block, and when it e
block, it exits from the original block as well, thus allowing construction of simple ca
forms. The use of cascading messages makes the following statement form possible.

```
var   1   [pen paint: green]; = 2   [pen paint: blue]; = 3   [pen up];   4
     [pen erase].
```

This is equivalent to the statement

```
[var   1   [pen paint: green] var = 2   [pen paint: blue]
 var = 3   [pen up] var   4   [pen erase]].
```

Remote Parameters

In order to use message sending to accomplish control, it is necessary to provide for pa
to be passed unevaluated, so that the receiver has control of this evaluation. This fun
provided by keyword parts which end with an open colon ( ) rather than a closed colon (:

this case, the parameter transmitted is a code object, and the receiver can send the mes value to that object in order to cause its evaluation. The code may never be evaluated, may be evaluated many times; it is up to the receiver.

For example, the expression

user displayOffWhile [fileSource _ strm].

tells the Smalltalk user interface to make the display screen black during the time need evaluate the expression in brackets. The expression is a request to a *file* to store a str characters. The method employed to respond to displayOffWhile first blackens the scree then sends the message *value* to the expression, and then restores the screen information.

Another example is

user time [dict lookup: 'twelve']

which tells the Smalltalk user interface to return the time that it takes to evaluate th in brackets. The method for time expr is of the form

t _ currentTime. expr value. currentTime - t.

The first statement assigns to the variable t the current value of the time. The second evaluates the expression in question. And the third statement returns the difference be new time and the old. For illustration purposes, we have assumed here that there is suc variable, currentTime, that is being updated by another process.

If a simple variable is passed unevaluated, the object transmitted is a remote reference variable and, in addition to being read with value, it may be stored into with the messa value_, allowing the implementation of loops with induction variables. For example,

heights _ students transform each to each height

Here the message transform to is being sent to the array students with two unevaluated parameters, each and each height. What the array will do is make an empty copy of itsel and then fill that copy by successively binding each to each element of the original and into the corresponding element of the copy the value of each height. While this may see like an elaborate system function, the entire definition in Smalltalk-76 is much smaller

paragraph. We will see the definition at the end of this section.

Control Messages

When we designed the syntax, it seemed that there were certain control functions (the *for* for instance) which could certainly be expressed nicely as keyword messages, but for who seemed to be no appropriate receiver. We decided to allow the receiver to be elided wit convention that the message was essentially being sent to the interpreter itself [actually the context, q.v. below]. In Smalltalk-76, three control messages have been implemented in this way: *for*, *until* and *while* statements. We have abused this realm a little, as one would expect (return a value) to be an infix operator in this way; and yet is defined as having a low precedence so that it will pick up everything to its right.

> The reader will probably enjoy finding more uniform ways to do this. We have had occasion to review these decisions, but have sat with our ad-hoc solutions because they have served so well, and because of a few other petty details. One of these is the ad-hoc choice of several control messages which the compiler converts into in-line tests and jumps, and other optimizations which are, in fact, essential to adequate performance. The most important consideration, which we *have* followed, is to limit these special cases to lie within a framework which is totally consistent with things the user can do. You can, in fact, define your own version of a for-loop in Smalltalk, and it will run in a manner identical to that which is defined in the system.

The For Statement

The *for* statement repeats a block of statements once for each of an ordered set of values During each repetition, the next value from that set is bound to an *iteration* variable. Th iteration variable is specified after the keyword for . Note again that naming a variab open colon not only defers evaluation, it also permits the control method to assign to t variable.

There are four forms of the *for* statment. They obtain the set of values for the iteratio in different ways. One form assigns to the iteration variable the successive integers b and a stop-value.

```
for  i to: 10 do  [i print].
```

will print the numbers 1 through 10, in order. Notice that the actual message is for to

with valence 3. Upon sending this message, evaluation of the first and third parameters deferred, while the second parameter is immediately evaluated.

Variations of this form allow a different initial value and a step value.

```
for  i from: 2 to: 10 do  [...]
for  i from: 2 to: 10 by: 2 do  [...]
```

The other form assigns to the iteration variable the successive elements of an object th responds to the message asStream in order to obtain an object that responds to the mesag next. As we will discuss more precisely later, several predefined classes in Smalltalk- able to do so, such as Stream, File, HashSet, and Array.

for variable from: term do [statements].

Examples are

```
for  color from: rainbowVector do  [screen flash: color].
```

and

```
for  prime from:  (2 3 5 7 11) do  [x\prime=0   [ prime]]
```

in which each of the first five prime numbers are tested as divisors of x (the \ message "modulo"), and the first divisor which works (if any) is returned from the surrounding m without further testing.

The Until Statement

The *until* statement repeats a block of statements until a condition is true. The test of condition is made before ech performance of the block.

```
until  term do  block.
```

Of course, if the block executes a return statement, it terminates not only itself, but method.

The While Statement

The *while* statement repeats a block statement until a condition is flase. The test is mad

each performance of the block.

```
while  term do  block.
```

Examples

<present the transform of an array example here

note: for, until, while could serve as examples by showing how the reader can program them himself>

Classes, Methods, and Subclasses

A Special Class: `Object`

In the Smalltalk-76 basic system, the class Object is implemented as the superclass of a
classes. It is an abstract class in that it has no state; its main function is to prov
foundation message protocol for its subclasses. As such, it provides a default set of c
for all objects in the basic system.

The message protocol of class Object includes the ability to respond to queries about wh
kind of class an object is or belongs to, to requests to convert an object to a Vector o
Stream (two classes provided in the basic Smalltalk-76 system), to attempts to inspect t
of an object or to print an object on the current output device or on a disk file. The
the class Object also provides default methods for responding to requests of the compile
of the process scheduler. The following is part of the message dictionary for class Obj

```
sameAs: object    are self and object the very same object?
class             to which class do I belong?
is: x             am I a member of the class x?
Is: x             is the class x a superclass or class of me?

copy              return a copy of me

print             print a representation of me on the display screen
printon: strm     print a representation of me on device strm
```

Defining a Class

As outlined in the section entitled *Capabilities of the Class Class*, the full representation of a
class includes a prelude which gives the name of the class and the format of its instanc

followed by a number of textual method descriptions. Access to the prelude and selectio
given method to work on are the subject of the chapter on user aids -- one seldom compos
full class description as linear text. The reader interested in the full textual repres
class may refer to any of the class descriptions printed in the appendices.

An example of the syntax of a class definition is

```
Class  new  title: 'BankAccount'
            subclassOf: Customer
            fields: 'acctNumber authorization transactions'
            declare: 'transactionMenu'
```

in which BankAccount is the name of the class, Record is its superclass, acctNumber,
authorization, and transactions are instance variables, and transactionMenu is a class
variable. Note that single quotes are placed around each argument except the superclass
messages must be in the order shown, but the subclassof: and declare: messages are
optional. The default superclass is Object.

Once a class has been defined, it may be redefined only with caution. If its fields (co
those of its superclass) have changed, then, in this system, all old instances become *obs*
they will fail to respond to messages. Furthermore, all the methods defined for the cla
bcome undefined until they are each updated and recompiled. When class redefinition is
attempted, the current user interface warns the user of any such impending trauma and gi
user a chance to withdraw the redefinition.

If a class is to share variables with other classes (*pool variables*), an additional clause, p
by a semi-colon, is added to the class definition for each such variable. The message p
be used is sharing: var.

To initialize class or pool variables, the class should include a message pattern classI
message is sent directly to the class, although in reality it is treated specially -- an
class is created and is then sent the message classInit. Because any instance of the cl
access to the class or pool variables, it can be used to initialize the variables. That
BankAccount classInit is identical to BankAccount new classInit. Earlier we saw
that the message init is treated as a special way to create an new instance, making it p

initialize instance variables or to update class and pool variables.

Defining a Method

The framework which takes us from the level of expressions to the definition of complete messages is

```
message pattern with parameter names | temporary variable names
    [code for implementing the response]
```

The brackets delimit a block, and within the block are statements which are expressions terminated by periods. An within this block indicates the return of a value and termi of the current method at that point.

Defining a Method makes or updates an entry in the message dictionary of the class, asso the procedure with the selector derived from the message pattern. A message pattern loo exactly like the message except that, instead of arguments, there are the names of varia (parameters) to which arguments will be assigned when a message is actually received.

If a particular argument variable is neither a class variable nor an instance variable o recipient, then it is automatically declared to be a temporary (method) variable. Addit method variables may be declared using the | construct.

Classes in the Basic System

The description of the objects in the system and the creation of new objects is accompli the Classes in the system. The Classes rely on certain other objects to accomplish this of these objects are part of the kernel system since they are necessary for the function system. This section lists the Classes of objects that make up the part of the kernel t Class Class. The discription of each Class includes an italicized statement of its purp followed by a description of its function followed by a description of the messages that relevant to its behavior in the kernel. The two interesting relationships between Class and subclass. These relationships are indicated in the following diagram, with class in red and subclass indicated in blue. Class Array is included in this diagram but not in following descriptions because it participates in the subclass heirarchy but not in the behavior of its two subclasses.

Object

Class

CollectionClass

Array

Vector

String

HashSet

Dictionary

Object

A primitive form for everything

Object is the ultimate superclass of all objects. It has no superclass. It defines mes
understood by all objects.

otherObject	is otherObject the same as the receiver
class	return the Class object that describes the receiver (note: that this is the deepest subclass for the receiver)
hash	return an Integer (this can be any number, but it must always be the same number for a particular object) (this message is used in the associative behavior of Dictionaries)

Vector

A primitive form of collection and a primitive form of association

A Vector has a set number of slots for objects. Each of the slots is numbered and the o
the collection are placed in specific slots.

length	how many objects are there room to collect
index _ value	include value in the collection associated with index (which must be an Integer between 1 and the length of the receiver)
index	return the object in the collection last associated with index

String

A primitive form of collection and association optimized for collecting Integers between 0 and 255

A String behaves like a Vector except that the objects collected must be numbers between 255. Strings are used to represent descriptions of behavior called methods. The method of instructions (primitive messages) for the interpretive machine. The instructions are represented as numbers between 0 and 255. Strings are also used for optimized collectio text. The text is represented as a collection of indices into a character set.

length	how many objects are there room to collect
index _ value	include value (which must be an Integer between 0 and 255) in the collection associated with index (which must be an Integer between 1 and the length of the receiver)
index	return the object in the collection last associated with index

HashSet

A non-duplicating, unbounded form of collection and a primitive form of inverted association

A HashSet does not have a fixed number of slots for the objects it collects, it will acc many objects as are inserted. If an object that is already in the collection is inserte appear once. Between insertions and deletions, the HashSet will associate each element contents with a unique Integer. These numbers may change when an object is inserted or deleted.

insert: value	include value in the collection but don’t duplicate it if it’s there already
delete: value	remove value from the collection (it should have been previously *inserted*)
contents	return a Vector containing the objects in the collection
find: value	return a unique Integer associated with *value*

Dictionary

An unbounded form of collection of general associations

A Dictionary is a subclass of HashSet that will collect an arbitrary number of associati pairs of objects.

key _ value	include value in the collection associated with key (which may be any object)

key	return the object in the collection last associated with key

Class

A general form of object description

A Class describes a set of objects called its instances.

new	return a new instance of the receiver
methodFor: selector	return the method (a String) that describes an instance's behavior when it receives a message whose selector is selector

CollectionClass

A form of description for optimized collection objects (Vector and String)

A CollectionClass describes a set of instances that can be of different sizes.

new: length	return a new instance of the receiver with length instance parts
methodFor: selector	return the method (a String) that describes an instance's behavior when it receives a message whose selector is selector

The Implementation of the Basic Classes

This section describes how the classes listed in the last section are implemented. This description of the parts of the objects and the methods for responding to the messages. the methods are what we call primitive methods. These are programs for the implementati machine as opposed to regular Smalltalk methods. They can directly interact with the vi memory of the machine. Some of the methods refer to pointers to objects. A pointer to object is a unique number that the virtual memory uses to refer to an object. For insta virtual memory represents the parts of objects by associating the pointers to the parts pointers to the object.

Object

A primitive form for everything

Most objects in the system are not direct instances of Object but rather instances of so subclass of Object. The three instances of Object itself are true, false and nil. The ob

called true and false are used to represent logical values and nil represents the default v
for a part of an object. Object does not contribute any instance parts so these three o
no parts at all. As a consequence, true, false and nil are identical except that they a
The only way to distinguish them is with the message. Any messages that they respond
differently to (like print) must ultimately send to them.

otherObject	if the pointer of the argument and the pointer of the receiver are the same, returns the object true; otherwise returns false
class	returns a pointer to the Class of which the receiver is an instance. This is known to the virtual memory
hash	returns an Integer object whose value is the pointer to the receiver

Vector

A primitive form of collection and a primitive form of association

Vector and String differ from the other Classes in the system in that their instances do
have the same number of parts. The parts of the instances represent the objects in the
and different collections have different sizes. This is a storage and access efficiency
consideration since collections could be represented as linked lists in the manner of Li
cells. Because a Vector has a fixed number of parts, this number must be specified when
instance is made. This is reflected in the fact that Vector and String are instances of
CollectionClass instead of Class. Instead of the new message for instantiation, Collect
uses a message called new: that takes an argument of the size of instance desired.

length	returns the number of parts in the receiver which is known to the virtual memory
index _ value	requests the virtual memory to replace the pointer representing the indexth part of the receiver with the pointer to the object supplied as an argument. Returns the argument
index	returns the pointer of the indexth part of the receiver

String

A primitive form of collection and association optimized for collecting Integers between 0 and 255

A String is a further optimized form of Vector. Since only Integers can be instance par
virtual memory stores the values of the parts instead of pointers to the parts. This is

efficient since pointers are longer than the eight bits necessary to store a value betwe
255.

length	returns the number of parts in the receiver which is known to the virtual memory
index _ value	requests the virtual memory to replace the value of the indexth part of the receiver with the value of the Integer supplied as a argument. Returns the argument
index	return an Integer whose value is the indexth part of the receiver

HashSet

A non-duplicating, unbounded form of collection and a primitive form of inverted association

A HashSet has one instance part named objects. objects is a Vector in which the HashSet stores the objects it collects. The HashSet determines where to store objects in objects hashing scheme to make duplication of objects efficient to detect.

insert: value	if the argument is not in objects, include it
delete: value	remove the argument from objects
contents	return a copy of objects with any nils removed
find: value	return the index of the argument in objects

Dictionary

An unbounded form of collection of general associations

A Dictionary is a subclass of HashSet which adds another instance part named values, so Dictionaries have two parts named objects and values. values is a Vector of the same length as objects. When two objects are associated, the key is inserted in objects using HashSet insert: method described above. The value to be associated with the key is then stored in values in the part with the same index.

key _ value	include value in the collection associated with key (which may be any object)
key	return the object in the collection last associated with key

Class

A general form of object description

A Class describes a set of objects called its instances.

new	return a new instance of the receiver
methodFor: selector	return the method (a String) that describes an instance's behavior when it receives a message whose selector is selector