Methods

BY DIGITALK, INC.

Smalltalk Language Guide

Contents

About This Manual

Smalltalk is an object-oriented programming language which executes under the Methods program development environment on the IBM personal computers and compatibles. This manual, the Smalltalk Language Guide, is an introduction to Smalltalk programming and a reference manual for the language. Accompanying this manual is the Methods Environment Guide, a user's guide and reference manual for the environment itself.

This manual explains:

- What Smalltalk is and how to learn it (Chapter 1)
- How to read Smalltalk if you know Pascal (Chapter 2)
- The concepts: objects, classes, messages and methods (Chapter 3)
- The syntax of Smalltalk methods (Chapter 4)
- The meaning of Smalltalk variables (Chapter 5)
- The nature of Smalltalk control structures (Chapter 6)
- How to specify classes (Chapter 7)
- What the key system classes do (Chapters 8, 9 and 10)
- How to write interactive applications (Chapter 11)
- An example of a complete Smalltalk program (Chapter 12)

Chapter 1 - Getting Started

What is Smalltalk?

Smalltalk is a programming language for developing interactive applications. It is object-oriented, a characteristic which simplifies programming for many types of problems, including simulations and graphical user interfaces.

Smalltalk is extensible. The "built-in" part of the language is small and therefore, easy to learn. Extensions which you add have the same syntax and semantics as the built-in language. Dialects for new application areas can easily be developed.

How do You Learn Smalltalk?

First, use it, and don't be afraid about breaking anything. Smalltalk can be used on the IBM PC and compatibles in the Methods program development environment. This is an interactive environment in which you can execute Smalltalk expressions and create and maintain Smalltalk programs. You can easily test small pieces of your program and view the results. It is unlikely that you will crash the Methods system with errors in the code that you write, although you can if you modify the system itself. Even if the system crashes, you can easily recover your work. Program errors are reported to you in pop-up windows which identify the erroneous code.

Second, become a code reader. Methods gives you access to most of the Smalltalk source code from which it is built. There are over 2000 routines (called methods) which you can browse, use and change. These manipulate numbers, dates, times, files, collections, windows, menus and more. By examining this code you can:

1) Learn the Smalltalk language

2) See how several problems were solved in Smalltalk

3) Become familiar with methods that you can use in your application programs

Third, read the Smalltalk-80 books written by the Smalltalk originators at the Xerox Palo Alto Research Center (PARC) and published by Addison Wesley. These present the history of Smalltalk and cover many topics in more depth than we do in this manual. The books are:

1) Smalltalk-80: The Language and Its Implementation
   by Adele Goldberg and David Robson, 1983

2) Smalltalk-80: Bits of History, Words of Advice
   Glenn Krasner, Editor, 1984

3) Smalltalk-80: The Interactive Programming Environment
   by Adele Goldberg, 1984

Chapter 2 - If You Know Pascal

Because you might be familiar with Pascal, this chapter presents an overview of Smalltalk by comparing examples of code in both languages. The left column below presents program fragments in Pascal, whereas the right column presents equivalent code fragments in Smalltalk.

1. Assignment to a scalar variable

```
a := b + c                    a := b + c
```

2. A series of statements/expressions

```
x := 0;                       x := 0.
y := 'answer';                y := 'answer'.
z := w                        z := w
```

3. A function call with one argument

```
a := size(array)              a := array size
```

4. A function call with two arguments

```
x := max(x1, x2)              x := x1 max: x2

y := sum(p, q)                y := p + q
```

5. A function call with 3 arguments

```
b := between(x, x1, x2)       b := x between: x1 and: x2
```

6. Subscripted variable access

```
x := a[i]                       x := a at: i

a[i + 1] := y                   a at: i + 1 put: y

a[i + 1] := a[i]                a at: i + 1 put: (a at: i)
```

7. If statements

```
if a < b then                   a < b
    a := a + 1                      ifTrue: [a := a + 1]

if atEnd(stream) then           stream atEnd
    reset(stream)                   ifTrue:  [stream reset]
else                                ifFalse: [c := stream next]
    c := next(stream)
```

8. Iterative statements

```
while i < 10 do begin           [i < 10]
    sum := sum + a[i];              whileTrue: [
    i := i + 1                          sum := sum + (a at: i).
    end                                 i := i + 1]

for i := 1 to 10 do             1 to: 10 do: [ :i |
    a[i] := 0                       a at: i put: 0]
```

9. Returning function results

```
functionName := answer;         ^answer
return
```

10. Storage allocation and deallocation

```
new(aFloat)                     aFloat := Float new

dispose(aFloat)
```

Chapter 3 - Objects, Classes, Messages and Methods

Although we have seen in Chapter 2 that Smalltalk is similar to Pascal in capabilities and appearance, the underlying concepts of the two languages are very different. This chapter introduces the key concepts of object-oriented programming: objects, classes, messages and methods.

A Smalltalk program is a collection of objects that compute by sending and receiving messages. Sending a message involves the following activities:

1) Identifying the object to which the message is sent (the receiver of the message).

2) Identifying the additional objects that are included in the message (the message arguments).

3) Specifying the desired operation to be performed (the message selector).

4) Accepting the single object that is returned as the message answer.

An example of sending a message is the Smalltalk expression "10+7". In this case, the receiver of the message is the number 10, the argument is the number 7, the operation to be performed is the selector "+", and the message answer is the number 17.

Each object has a collection of methods which define the messages that the object understands. When a message is sent to an object, the object's collection of methods is searched to find the one that has the same selector. If there is no matching method, the object is notified of the programming error by sending it the message "You do not understand" with a description of the original message as an argument. All objects understand this error notification message. An example of a message which illustrates this error is "'sam'+7". In this case, the operation "+" is not defined for objects like 'sam' (i.e., strings).

Objects are grouped into classes, where all objects belonging to a class have the same data structure and the same collection of available methods. An object may be created and manipulated only by the methods of its class.

A class specifies its member objects' structure by defining instance variables, the individually accessible components of an object. Instance variables either have a name or are referred to with an integer index. Instance variables have a type: they contain either pointers, words or bytes. All instance variables for objects belonging to the same class are the same type. If an object contains pointers, then its instance variables refer to other objects. If an object contains words or bytes, then its instance variables contain 16-bit or 8-bit integer values, respectively. Objects containing words or bytes are used to express elementary values such as strings or numbers.

Smalltalk classes are organized into a hierarchy. A subclass inherits all the instance variables of its superclass. A subclass also inherits all the methods of its superclass that it does not re-implement. An example of inheritance is class SortedCollection (class names always begin with a capital letter) which is a subclass of OrderedCollection. OrderedCollection defines named instance variables startPosition and endPosition. SortedCollection defines named instance variable sortBlock. An object which is a member of class OrderedCollection has named instance variables startPosition and endPosition; whereas an object which is a member of class SortedCollection has named instance variables startPosition, endPosition and sortBlock.

SortedCollection inherits method includes: from OrderedCollection. SortedCollection defines the method add:, so it does not inherit the method add: defined in OrderedCollection.

Some superclasses exist only to define methods to be inherited by their subclasses. They are referred to as abstract classes. Abstract classes never have any members. Class Object is an abstract class which is at the top of the class hierarchy, so all other classes inherit its methods.

Chapter 4 - Smalltalk Syntax

This chapter presents the syntax of Smalltalk methods, the executable program units in a Smalltalk system. In the preceding chapter we saw that each method belongs to a class and operates directly on objects belonging to the class. A class specifies both a collection of methods and the structure of objects belonging to it. There is no syntax specification for classes. Classes are defined using the programming environment or by sending a message to the class' superclass to specify class characteristics.

Smalltalk syntax is simple. A method consists of a series of expressions which can describe the following actions.

1) Send a message to an object to request execution of a method. For example: Scheduler resume.

2) Assign an object to a variable. For example: x := 5.

3) Terminate execution and return an object as the method answer. For example: ^count.

Smalltalk is an extensible language. New methods have the same syntax as built-in methods. New classes specify their behavior in the same way as built-in classes.

In order to present the syntax, we first define the meta-language used for syntax specification.

How Syntax is Specified

The Smalltalk syntax is presented using the Extended Backus Naur Formalism (EBNF) used in Programming in Modula-2 by Niklaus Wirth, Springer-Verlag 1982. We use this formal approach in order to precisely and concisely specify the syntax. We begin with a specification of EBNF syntax in EBNF.

```
<rule>  syntax = {rule}.
<rule>  rule = "<rule>" identifier "=" expression ".".
<rule>  expression = term {"|" term}.
<rule>  term = factor {factor}.
<rule>  factor = identifier | string | "(" expression ")" |
                 "[" expression "]" | "{" expression "}".
```

An EBNF specification is a sequence of syntax rules. The right hand side of each rule defines syntax in terms of other rule names and terminal symbols of the language. Parentheses "(" and ")" group alternative terms. The vertical bar "|" separates alternative terms. Brackets "[" and "]" identify optional expressions. Braces "{" and "}" identify expressions which may occur zero or more times. Character sequences in paired quotes, either (") or ('), identify terminal symbols of the defined language. An identifier is a sequence of letters and digits beginning with a letter. A string is a sequence of characters from the defined language.

The following is an example in which we define possible meals with a sequence of EBNF rules.

```
<rule>  appetizer = "artichoke" | "oysters".
<rule>  dessert = "ice-cream" | fruit.
<rule>  fruit = "apple" | "orange" | "pear".
<rule>  meat = "beef" | "lamb" | "fish".
<rule>  vegetable = "broccoli" | "carrots" | "peas".
<rule>  meal = [appetizer] meat ("potatoes" | "rice")
              {vegetable} [dessert].
```

Examples of meals defined by these rules are the following:

```
beef potatoes
artichoke fish rice peas broccoli ice-cream
lamb rice carrots carrots carrots peas broccoli pear
oysters beef rice orange
```

The Syntax of Variable Names and Literals

Variable names and literals are the elemental building blocks used in higher level syntax forms in Smalltalk.

Variable Names

```
<rule>  variableName = identifier.
<rule>  identifier = letter {letter | digit}.
```

A variableName identifies a variable in an object. A variable is a container for an object pointer. Example variableNames are:

```
OrderedCollection   aString   elements   x2
```

For a complete discussion of variables, see Chapter 5.

Literals

```
<rule>  literal = number | string | characterConstant |
                  symbolConstant | arrayConstant.
```

A literal defines an object of class Number, String, Character, Symbol or Array. Examples are given below where each of the possible literal forms are defined.

Numbers

```
<rule>  number = [digits "r"] ["-"] bigDigits ["." bigDigits]
                 ["e" ["-"] bigDigits].
<rule>  digits = digit {digit}.
<rule>  digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
                "8" | "9".
<rule>  bigDigits = bigDigit {bigDigit}.
<rule>  bigDigit = digit | capitalLetter.
```

If a number contains a decimal point it is an object of class Float. If it contains a negative exponent and no decimal point it is of class Fraction. Otherwise, it is of class Integer. If the number includes "r", the digits preceding "r" define the number radix. In this case, capital letters are used to represent digit values greater than 10, with "A" = 10, "B" = 11, etc. Example numbers are:

```
15   16rFF   3.1416   1e-3
```

Strings

```
<rule>  string = "'" {character | "''" | '"'} "'".
<rule>  character = letter | digit | selectorCharacter |
                    "[" | "]" | "(" | ")" | "^" | ";" | "$" |
                    "#" | ":" | "." | "{" | "}" | " ".
<rule>  selectorCharacter = "," | "+" | "/" | "\" | "*" | "-" |
                            "<" | ">" | "=" | "@" | "%" | "|" |
                            "&" | "?" | "!".
```

A string is an object of class String which is an indexable sequence of objects of class Character. Note that strings are not necessarily constant; their characters may be changed by sending a message to the string. Paired quotes within a string reduce to a single quote in the resultant string object. Example strings are:

```
'hello'   ''   'isn''t'   '"comment in string"'
```

Comments

```
<rule>  comment = '"' {character | "'"} '"'.
```

A comment is ignored anywhere within a method, except when occuring within a string. Example comments are:

```
"Answer the size of the receiver"   "goodBye"
```

Character Constants

```
<rule>  characterConstant = "$" character | "$" "'" | "$" '"'.
```

A characterConstant is an object of class Character. Example characterConstants are:

```
$$   $a   $'   $    $.
```

Symbols

```
  <rule>  symbolConstant = "#" symbol.
  <rule>  symbol = unarySelector | binarySelector |
                   keyword {keyword}.
  <rule>  unarySelector = identifier.
  <rule>  binarySelector = selectorCharacter [selectorCharacter] |
                           "_".
  <rule>  keyword = identifier ":".
```

A symbol is an object of class Symbol, an indexable sequence of objects of class Character. Symbols differ from strings in that their characters may not be changed. A symbolConstant identifies the associated symbol object. Example symbolConstants are:

```
    #-   #asOrderedCollection   #at:put:   #==
```

Arrays

```
  <rule>  arrayConstant = "#" array.
  <rule>  array = "(" {number | string | symbol | array |
                       characterConstant} ")".
```

An array is an object of class Array which may be indexed by an integer from 1 through the size of the array. An arrayConstant identifies the associated array object. Example arrayConstants are:

```
    #('red' 'blue`' 'green')   #(yes no)   #(1 'two' three $4 (5))
```

The Syntax of Smalltalk Expressions

The actions in a Smalltalk method are specified by a series of expressions.

```
<rule>  expressionSeries = {expression "."} [["^"] expression].
<rule>  expression = {variableName ":="} (primary |
                    messageExpression {";" cascadeMessage}).
<rule>  primary = variableName | literal | block |
                  "(" expression ")".
```

Each expression calculates a single object as its result. The expression may also include assignment of its result to one or more variables. Examples of expressions which assign to variables are:

```
i := j := 0.
weekdays := #(Monday Tuesday Wednesday Thursday Friday).
```

The final expression in an expressionSeries may be preceded by "^", meaning that method execution terminates answering the object computed by the expression. Examples of expressions which compute the method answer are:

```
^#(Saturday Sunday)
^sales - expenses
```

A messageExpression is a request to an object (the receiver of the message) to perform a computation and return an object as answer. There are three kinds of message expressions: unary, binary and keyword (n-ary). Each has a different precedence and a different syntax for its selector, the name of the message.

```
<rule>  messageExpression = unaryExpression | binaryExpression |
                            keywordExpression.
<rule>  cascadeMessage = unaryMessage | binaryMessage |
                         keywordMessage.
<rule>  unaryExpression = primary unaryMessage {unaryMessage}.
<rule>  binaryExpression = (unaryExpression | primary)
                           binaryMessage {binaryMessage}.
<rule>  keywordExpression = (binaryExpression | primary)
                            keywordMessage.
<rule>  unaryMessage = unarySelector.
```

```
<rule>  binaryMessage = binarySelector (unaryExpression |
                       primary).
<rule>  keywordMessage = keyword (binaryExpression | primary)
                         {keyword (binaryExpression | primary)}.
```

A unaryExpression sends a series of unaryMessages which are evaluated from left to right. A unaryMessage has no arguments. Examples of unaryExpressions are:

```
letter isVowel not
anObject class name size
10 factorial
```

A binaryExpression sends a series of binaryMessages which are evaluated from left to right. A binaryMessage has a single argument following the binarySelector. Examples of binaryExpressions are:

```
i + j * 2
aCharacter == Space
x @ y
```

The traditional arithmetic operators are implemented in Smalltalk using binaryExpressions. Since this makes all arithmetic operators the same precedence, expectations about evaluation order may not be met. In the first example above, the result of i + j is sent the message * with 2 as the argument.

A keywordExpression sends a single keywordMessage with one or more arguments. The arguments to a keywordMessage are evaluated from left to right. Examples of keywordExpressions are:

```
anArray at: 1
index between: start and: stop
Rectangle origin: point1 corner: point2
```

The selector of a keywordMessage is the concatenation of all the keywords in the message. In the examples above, the selectors are at:, between:and:, and origin:corner:.

Unary expressions have highest precedence, followed by binary and then keyword. Parentheses may be used to specify a different evaluation order. An example of a mixed expression is:

```
anArray at: anArray size - 1 put: element class
```

This expression is evaluated in the following steps:

1) The message size is sent to (the object identified by the variable) anArray.
2) The message - is sent to the result of the size message with 1 as argument.
3) The message class is sent to the variable element.
4) The message at:put: is sent to the variable anArray with the results of the - and class messages as arguments.

Cascaded messages are a series of messages to the same receiver. Each message after the first is preceded by a semicolon. The following two examples are equivalent:

```
pen up;                    pen up.
    go: 10;                pen go: 10.
    down;                  pen down.
    go: 5;                 pen go: 5.
    turn: 90;              pen turn: 90.
    go: 5.                 pen go: 5.
```

A block is a part of a method enclosed in square brackets. It is an object describing executable code. Blocks may be nested.

```
<rule>  block = "[" [{":" variableName} "|"]
                expressionSeries "]".
```

A block may have arguments. These are specified between the left bracket and vertical bar by preceding each argument variableName with a colon. Examples of blocks are:

```
[^true]
[:argument | argument isVowel]
[ :a :b | a < b ]
```

The result of block execution is the final expression in the block. A block with no arguments is executed by sending it the message value. The following examples compute the same result.

```
a := b + c.
a := [b + c] value.
x := [b + c]. a := x value.
```

A block with one argument is executed by sending it the message value:. The argument to the value: message is assigned to the block argument upon block execution. The following examples compute the same result.

```
count := #(1 2 3) size.
[:aCollection | count := aCollection size] value: #(1 2 3).
count := [:element | element size] value: #(1 2 3).
```

A two-argument block is executed by sending it the message value:value:. The value:value: arguments are assigned to the block arguments. For example:

```
[:a :b | a < b] value: 2 value: 10
```

The expression above returns the value true.

A block may contain an expression preceded by an up-arrow. Evaluation of such an expression causes termination of execution for both the block and the method in which the block appears. For example:

```
^dictionary at: name ifAbsent: [^nil]
```

In the example above, there are two possibilities for terminating the method containing the at:ifAbsent: message. If the block is executed, the method returns nil. Otherwise the method returns the answer of the at:ifAbsent: message.

Blocks are the basis for control structures in Smalltalk. Since control structures conform to keyword message syntax, there is no special syntax for them. See Chapter 7 for a discussion of control structures.

The Syntax of Smalltalk Methods

A complete method specification includes a messagePattern, optional primitiveNumber, optional temporaries and an expressionSeries. The messagePattern specifies how to send a message to request method execution. It includes the method selector and the variable names used to refer to arguments within the method.

```
<rule>  method = messagePattern [primitiveNumber] [temporaries]
                 expressionSeries.
<rule>  messagePattern = unarySelector |
                binarySelector variableName |
                keyword variableName {keyword variableName}.
<rule>  primitiveNumber = "<" "primitive:" number ">".
<rule>  temporaries = "|" {variableName} "|".
```

The following are examples of complete methods:

method class: Character

```
method:         isVowel
                    ^'aAeEiIoOuU' includes: self
```

description: The message includes: is sent to the string containing all vowels with the receiver of class Character (identified as self) passed as argument.

```
example use:    aLetter isVowel
                $X isVowel
```

method class: Magnitude

```
method:         max: aMagnitude
                    self > aMagnitude
                        ifTrue: [^self]
                        ifFalse: [^aMagnitude]
```

description: The receiver self and the argument aMagnitude are compared. If the receiver is greater, it is returned as method answer; otherwise, the argument is returned as answer.

example use:
```
1 max: index
$s max: $S
```

method class: Fraction

method:
```
<= aNumber
    ^(numerator * aNumber denominator) <=
        (denominator * aNumber numerator)
```

description: The identifiers numerator and denominator are used in two different ways in this method: as instance variables (Chapter 5) and as message selectors. A fraction has instance variables numerator and denominator. The argument aNumber is sent the messages denominator and numerator to return these values for arguments of class Fraction, Integer or Float. The result less-than-or-equal message will compare operands which are either class Integer or class Float.

example use:
```
(2/7) <= (3/11)
22/7 <= 3.1415
```

Chapter 5 - Variables

A variable contains a single object pointer. The variable name can be used in an expression to refer to the object whose pointer it contains. A variable may refer to different objects at different times. The object referred to by a variable is changed when an assignment expression is evaluated. For example:

```
collection := #('sales' 'expenses').

newCollection := collection
```

The first assignment above causes the variable collection to refer to the literal array. The second assignment causes the variable newCollection to refer to the same object as the variable collection. Note that an assignment does not copy an object, but only a pointer to the object.

Variables are either private or shared. Private variables are accessible only to a single object. Shared variables are accessible to multiple objects. Private variables are those with lower case first letter, while shared variables are those with upper case first letter.

There are five kinds of variables, the first two are private and the last three are shared.

1) Instance variables are the component parts of an object. They exist for the lifetime of the object.

2) Temporary variables are created during the activation of a method and are available only from within the method during the single activation. They include method arguments, method temporaries defined in vertical bars, and block arguments.

3) Class variables are shared by all objects of a class.

4) Pool variables are shared by objects belonging to one or more classes.

5) Global variables are shared by all objects.

Instance and class variables are defined as part of a class specification (see Chapter 7). Temporary variables are defined in the source code for a method (see Chapter 4). Global and pool variables are defined by entering them into a dictionary (see Shared Variables at the end of this chapter).

Instance Variables

Instance variables are the component parts of an object. They are implicitly initialized to the value nil upon object creation. There are two types of instance variables, named and indexed, which are declared and accessed differently. Named instance variables are accessed by name. Indexed instance variables are accessed only through messages (usually using at: and at:put: with integer indices).

Classes may specify both named and indexed instance variables for their member objects. The number of named instance variables is fixed for all members of the class. The number of indexed instance variables may differ among members of the same class. For example, #(1 2 3) and #('up' 'down') are both objects of class Array, but they have different numbers of indexed instance variables. Classes with indexed instance variables create new members with a message that specifies the number of indexed instance variables to create (usually the message new: with an integer argument). The number of indexed instance variables is obtained by sending a message (usually the message size).

Temporary Variables

Temporary variables include method arguments and temporaries, and contained block arguments. Method arguments are assigned the associated message arguments for the message which caused method invocation. Method arguments may not appear to the left of the assignment operator ":=" in an expression. Method temporaries are initialized to nil upon method invocation. Block arguments are assigned the associated message arguments for the value: message which caused block activation.

When a block is invoked while its containing method is still active, the block and the containing method share the same temporary variables. For example, consider the method size for class Bag:

```
size
    | count |
    count := 0.
    elements do: [:each | count := count + each].
    ^count
```

In the method above, the block is evaluated by the do: method, but it accesses the same copy of variable count as the rest of the size method.

Shared Variables

Shared variables are defined in dictionaries called pools. The system dictionary Smalltalk is a pool which contains all the global variables. An example of an expression which creates a new global variable Name and initializes it to an empty String is as follows.

```
Smalltalk at: #Name put: ''
```

Class variables for each class are implicitly collected into other pools, one per class. Additional pools must be explicitly constructed by the programmer. One example of such a pool is the dictionary CharacterConstants used in classes Character and Stream. The following expression creates and initializes the variable for the character line-feed.

```
CharacterConstants at: 'Lf' put: (Character value: 10)
```

Chapter 6 - Control Structures

Smalltalk control structures are invoked by sending messages with blocks as arguments. Three forms, with several variations, are built-in. Additional forms are defined in Smalltalk using these built-in ones.

Conditional Execution

The following built-in conditional execution messages are available:

```
ifTrue:
ifFalse:
ifTrue:ifFalse:
ifFalse:ifTrue:
```

In all cases, the receiver expression must be of class Boolean and the arguments must be blocks with no arguments. The ifTrue: argument block (if present) will be sent the message value if and only if the receiver has the value true. The ifFalse: argument block (if present) will be sent the message value if and only if the receiver has the value false. The answer of the conditional messages is the last expression in the executed block or nil if no block is executed. Some examples are:

```
index < start
    ifTrue: [index := start]


maximum := (a < b ifTrue: [b] ifFalse: [a])
```

Iterative Execution

The following built-in iterative execution messages are available:

```
whileTrue:
whileFalse:
```

Both the receiver and argument of these messages must be no-argument blocks. For whileTrue:, the receiver block is sent the message value, and if it answers true the argument block is sent the message value. The iteration is continued until the answer of the first block evaluation is false. For whileFalse:, the sequence is the same but the iteration is continued until the answer of the first block evaluation is true. The answer of both whileTrue: and whileFalse: is always nil. Some examples are:

```
sum := 0.
index := elements size.
[index > 0]
    whileTrue: [
        sum := sum + (elements at: index).
        index := index - 1]


[inputStream atEnd]
    whileFalse: [outputStream nextPut: inputStream next]
```

Short Circuit Boolean Evaluation

The following built-in boolean operators are available:

```
and:
or:
```

The receiver of each of these methods must be of class Boolean and the argument a block. For and:, the block will be sent the message value and the answer of the message will be the last block expression if and only if the receiver is true. Otherwise, the answer of the and: message is false and the block is not evaluated. For or:, the block will be sent the message value and the answer of the message will be the last block expression if and only if the receiver is false. Otherwise, the answer of the or: message is true and the block is not evaluated. Examples are:

```
stream atEnd or: [stream next = $!]


start <= index and: [index <= stop]
```

Chapter 7 - Class Specification

The Smalltalk language syntax does not include class specification, only method specification. Classes are defined by sending a message to the new/modified class's superclass with class specification information as arguments. The class information that can be specified is the following:

1) The class name

2) Whether objects of the class contain pointers, words or bytes

3) Whether objects of the class can contain indexed instance variables

4) The names of the named instance variables for objects of the class

5) The names of the class variables available to all objects of the class

6) The names of the pool dictionaries which define shared variables available to objects of the class and possibly other classes

The message which specifies a class is sent to its superclass. There are four class definition messages. They are as follows:

1) subclass: subclassSymbol
   instanceVariableNames: instanceVariableNameString
   classVariableNames: classVariableNameString
   poolDictionaries: poolDictionaryNameString

2) variableSubclass: subclassSymbol
   instanceVariableNames: instanceVariableNameString
   classVariableNames: classVariableNameString
   poolDictionaries: poolDictionaryNameString

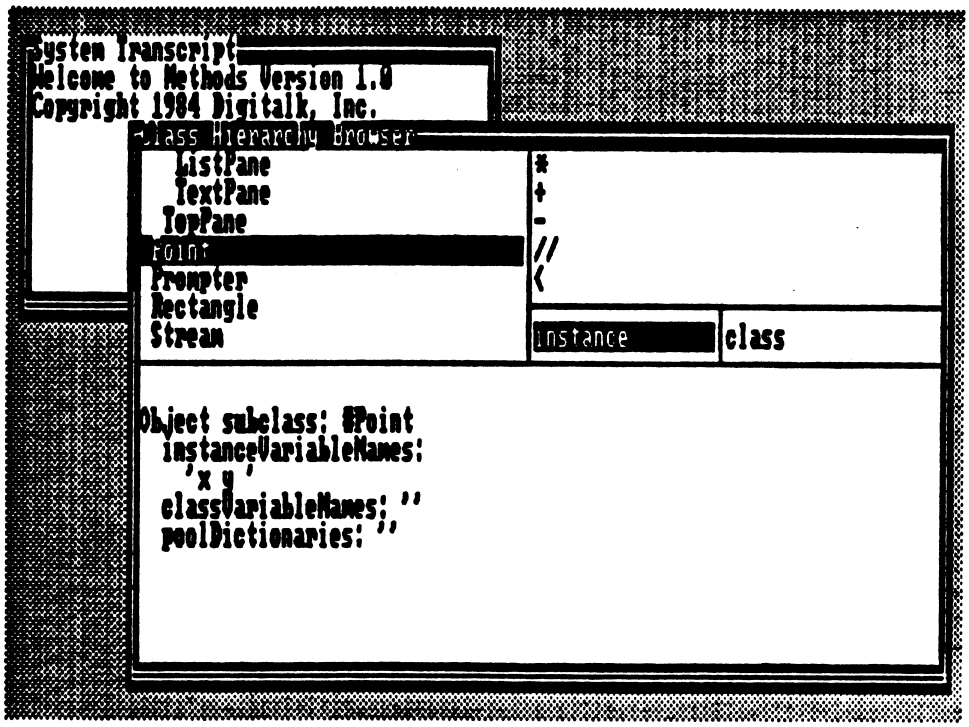
3) variableWordSubclass: subclassSymbol
   classVariableNames: classVariableNameString
   poolDictionaries: poolDictionaryNameString

4) variableByteSubclass: subclassSymbol
classVariableNames: classVariableNameString
poolDictionaries: poolDictionaryNameString

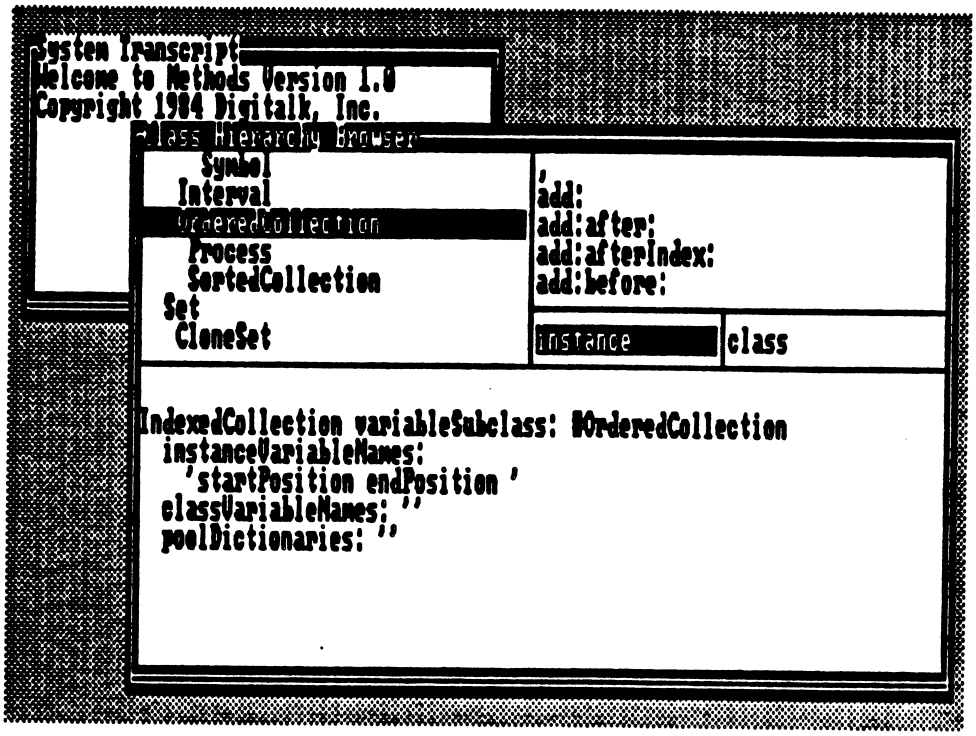
The first two messages define classes whose member objects contain pointers. The first message specifies objects with named instance variables (zero or more of them). The second message specifies objects with both named and indexed instance variables.

The third message defines classes whose member objects contain words. Objects with words contain only indexed instance variables, so there is no instance variable name string argument.

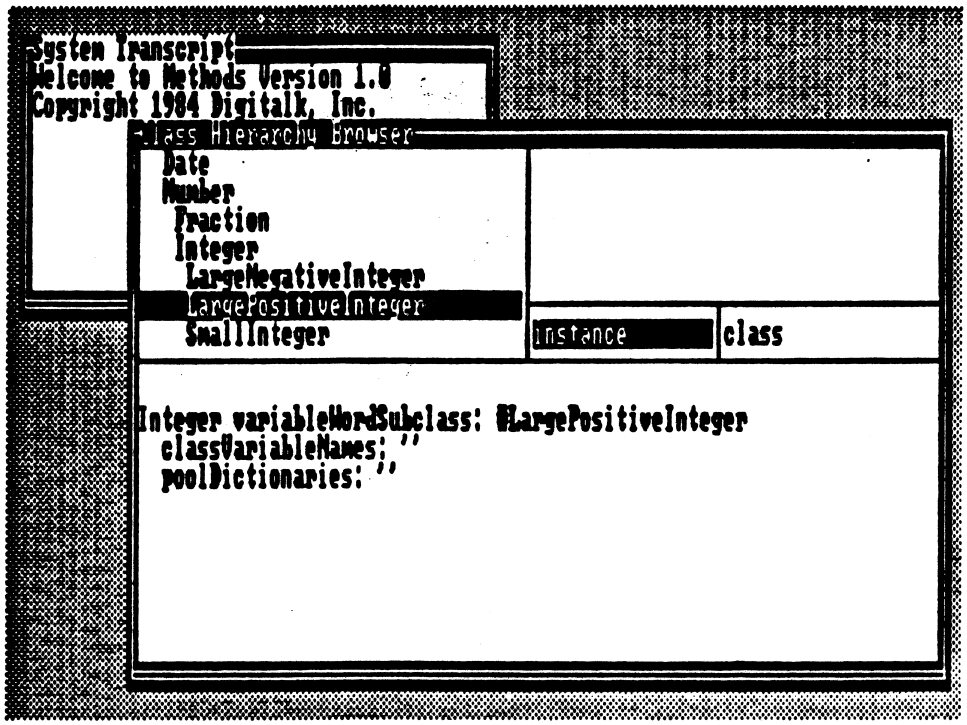
The fourth message defines classes whose member objects contain bytes. Objects with bytes contain only indexed instance variables, so there is no instance variable name string argument.



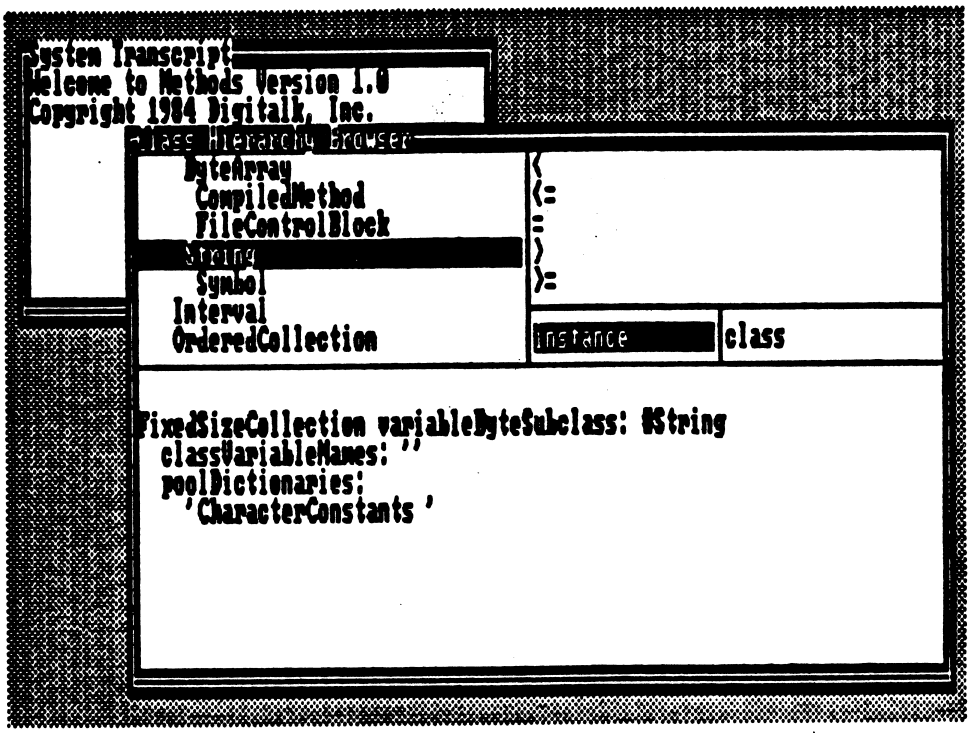
The picture of the display above shows a Class Hierarchy Browser with the bottom pane containing the definition of class Point. Class Point has two named instance variables, x and y. There are no class variables or pool dictionaries.



The picture of the display above shows a Class Hierarchy Browser with the bottom pane containing the definition of class OrderedCollection. Class OrderedCollection has two named instance variables, startPosition and endPosition, and additional indexed instance variables. There are no class variables or pool dictionaries.



The picture of the display above shows a Class Hierarchy Browser with the bottom pane containing the definition of class LargePositiveInteger. Class LargePositiveInteger has only indexed instance variables. There are no class variables or pool dictionaries.



The picture of the display above shows a Class Hierarchy Browser with the bottom pane containing the definition of class String. Class String has only indexed instance variables. Class String has no class variables. Class String uses pool dictionary CharacterConstants.

Chapter 8 - Streams

(To be completed)

Chapter 9 - Collections

(To be completed)

Chapter 10 - Magnitudes

(To be completed)

Chapter 11 - The Trilogy: Writing Interactive Applications

(To be completed)

Chapter 12 - A Complete Smalltalk Program

The Problem

Appendix 1 contains a summary of the EBNF syntax specification of Smalltalk and a cross-reference to the syntax which shows where each identifier is defined and referenced in the syntax specification. This chapter presents the Smalltalk program which produced Appendix 1, a new class called CrossReference.

CrossReference accepts the input syntax specification from an instance of class ReadStream or one of its subclasses. The input contains a series of lines. A line is terminated by either carriage-return, line-feed (MS-DOS convention) or line-feed (Unix convention). Each line contains words and non-words. A word begins with a letter and contains letters and digits. A word beginning in column one of a line is assumed to be defined on the line. Words beginning in other columns are references to a definition appearing elsewhere. Non-words are the characters between words on a line. They are ignored.

CrossReference produces its listing on an instance of class WriteStream or one of its subclasses. First the input syntax is listed with each line preceded by its line number. Next the sorted cross-reference appears. Each line contains a word and a list of numbers of the lines which define or refer to the word. A definition is flagged with a minus sign.

The CrossReference Class Specification

The message which creates class CrossReference is as follows:

```
Object subclass: #CrossReference
    instanceVariableNames:
        'references source listing line lineCount'
    classVariableNames: ''
    poolDictionaries: 'CharacterConstants'
```

There are five instance variables and the pool dictionary CharacterConstants. The instance variables are used as follows.

references	A dictionary of word references. The key of each entry is the string for a word. The value of each entry is an OrderedCollection of integers representing the line numbers on which the associated word is defined or referenced.
source	The input stream containing the syntax specification.
listing	The output stream containing the syntax and cross-reference listings.
line	A string representing the line being processed.
lineCount	The integer count of the number of lines processed.

The CrossReference Methods

We now present the five methods of class CrossReference in top-down order. The message source:listing: is sent to an instance of CrossReference to request a cross-reference listing. First the source and listing instance variables are assigned the argument streams, and the references instance variable is assigned an empty dictionary. Then the message listSource is sent to self (the CrossReference object) to list the source. The message reset is sent to the source stream in order to position it at the beginning. Then the messages collectReferences and listReferences are sent to self. Finally, the listing stream is sent the message close which, for a FileStream, closes the file.

```
source: sourceStream listing: listStream
        "Initialize a CrossReference instance, list the source
         with line numbers followed by the cross references"
    source := sourceStream.
    listing := listStream.
    references := Dictionary new.
    self listSource.
    source reset.
    self
        collectReferences;
        listReferences.
    listing close
```

The method listSource produces a listing of the source stream with line numbers. A whileFalse: loop is used to process all the lines of the source stream. For each line, the line number is printed followed by the line itself. Lines are obtained from the source stream with the message nextLine, which returns a string. The shared variable Space is defined in the CharacterConstants pool dictionary.

```
listSource
        "Copy the lines of the source stream to
         the listing stream preceding each line
         with its line number"
    lineCount := 1.
    [source atEnd]
        whileFalse: [
            listing
                nextPutAll: lineCount printString;
                next: 4 - lineCount printString size put: Space;
                nextPutAll: source nextLine;
                cr.
            lineCount := lineCount + 1]
```

The method collectReferences scans the source stream and builds the references dictionary. A whileFalse: loop iterates until the end of the source stream is reached. The message nextLine is sent to the source stream to obtain a line as a string which is assigned to instance variable line. The message scanLine is sent to self to do the major work of scanning the line and building dictionary entries.

```
collectReferences
        "Read the lines of the source stream and
         build the references dictionary"
    lineCount := 1.
    [source atEnd]
        whileFalse: [
            line := source nextLine.
            self scanLine.
            lineCount := lineCount + 1]
```

The method scanLine processes the string in instance variable line to build the references dictionary. First a temporary ReadStream is created on line in order to use the nextWord message already defined in class Stream. Then a whilefalse: message iterates until all the words in the line are processed. Words are obtained by sending the message nextWord to the ReadStream. Each word is looked up in the references dictionary using the message includesKey:. If the word is not present, it is added with an associated empty OrderedCollection as value. Finally, the entry for the word has the current value of lineCount added to the associated OrderedCollection. If the first word of a line begins in column one, the negative of lineCount is used to indicate a defining occurrence of the word.

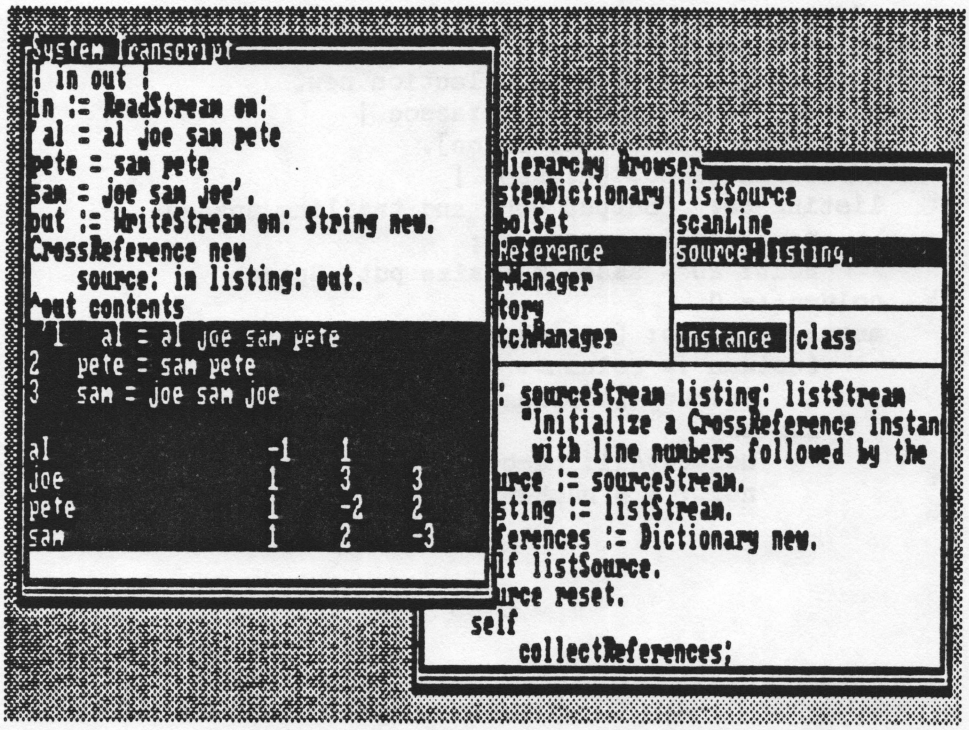
```
scanLine
        "Scan the words on the current line and add
         to references dictionary"
    | lineStream word isFirstWord |
    lineStream := ReadStream on: line.
    isFirstWord := true.
    [(word := lineStream nextWord) == nil]
        whileFalse: [
            word size = 1 "ignore 1-letter words"
                ifFalse: [
                    (references includesKey: word)
                        ifFalse: [
                            references at: word
                                    put: OrderedCollection new].
                    (references at: word) add:
                        ((isFirstWord and: [(line at: 1) isLetter])
                            ifTrue: [lineCount negated] "definition"
                            ifFalse: [lineCount])].     "reference"
            isFirstWord := false]
```

The method listReferences produces the cross-reference listing using the references dictionary. First an empty SortedCollection is created and the entries of references are added to it with the associationsDo: loop. Then the sorted associations are iterated over in order with the do: loop. Each word (the association key) is copied to the listing followed by its collection of reference line numbers (the association value). If there are too many references to fit on one line, then succeeding lines with blank word field are used.

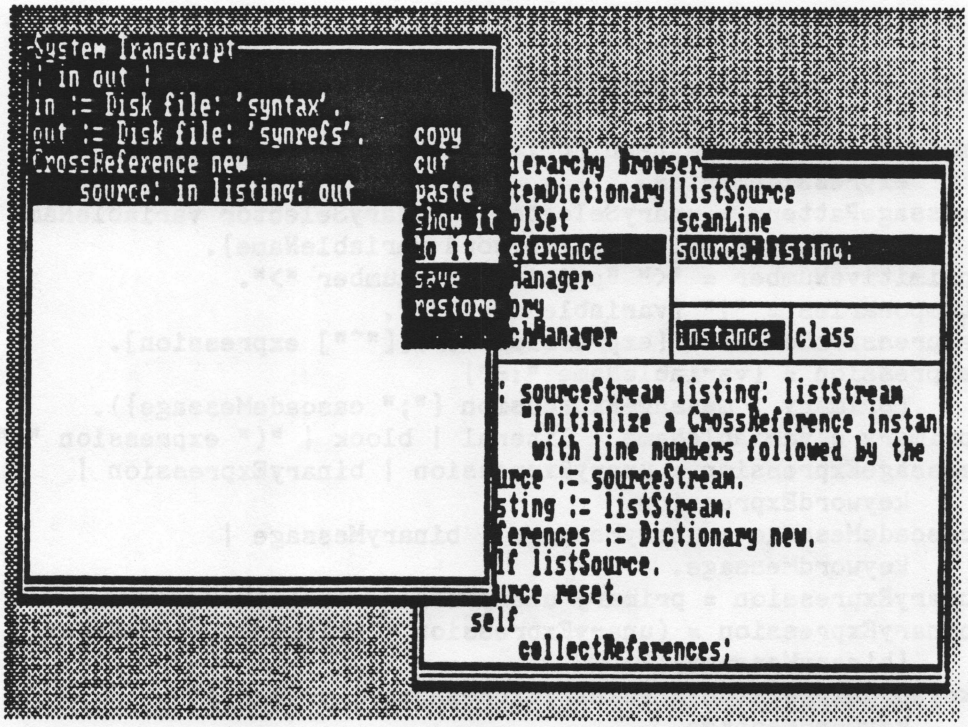
```
listReferences
        "Produce a sorted list of names with reference
         line numbers on the listing stream"
    | sortedReferences column |
    sortedReferences := SortedCollection new.
    references associationsDo: [ :assoc |
        sortedReferences add: assoc].
    sortedReferences do: [ :assoc |
        listing cr;  "output word and trailing spaces"
            nextPutAll: assoc key;
            next: 20 - assoc key size put: Space.
        column := 0.
        assoc value do: [ :number | "output references"
            (column := column + 1) \\ 9 = 0
                ifTrue: [listing cr; next: 20 put: Space].
            listing
                nextPutAll: number printString;
                next: 6 - number printString size put: Space]]
```

Testing Class CrossReference

Cross-references are most likely to be used with instances of class FileStream for input and output. However, we can simplify our initial testing by taking advantage of the fact that other classes obey the same protocol as does class FileStream. In our case, we use an instance of ReadStream on a String for the input syntax and a WriteStream on a String for listing output. In this way we immediately view the input and output during testing.



The figure above shows a sample screen with two windows: the system transcript and a class hierarchy browser. The transcript contains the messages which create the test data stream and an instance of class CrossReference. The output of the test run is shown reversed in the transcript. This was produced by selecting the text above the output and executing it with the show it menu command.



The picture of the display above shows Smalltalk expressions in the transcript which obtain a cross-reference with the source and listing both instances of class FileStream. The files are both in the current directory Disk, and have pathnames syntax and synrefs respectively. The expression text is selected and the menu is positioned to execute the text with the do it command.

Appendix 1 - Smalltalk Syntax Summary and Cross-Reference

```
1   method = messagePattern [primitiveNumber] [temporaries]
2       expressionSeries.
3   messagePattern = unarySelector | binarySelector variableName |
4       keyword variableName {keyword variableName}.
5   primitiveNumber = "<" "primitive:" number ">".
6   temporaries = "|" {variableName} "|".
7   expressionSeries = {expression "."}[["^"] expression].
8   expression = {variableName ":="}
9       (primary | messageExpression {";" cascadeMessage}).
10  primary = variableName | literal | block | "(" expression ")".
11  messageExpression = unaryExpression | binaryExpression |
12      keywordExpression.
13  cascadeMessage = unaryMessage | binaryMessage |
14      keywordMessage.
15  unaryExpression = primary unaryMessage {unaryMessage}.
16  binaryExpression = (unaryExpression | primary) binaryMessage
17      {binaryMessage}.
18  keywordExpression = (binaryExpression | primary)
19      keywordMessage.
20  unaryMessage = unarySelector.
21  binaryMessage = binarySelector (unaryExpression | primary).
22  keywordMessage = keyword (binaryExpression | primary)
23      {keyword (binaryExpression | primary)}.
24  block = "[" [{":" variableName} "|"] expressionSeries "]".
25  keyword = identifier ":".
26  binarySelector = "-" | selectorCharacter [selectorCharacter].
27  unarySelector = identifier.
28  literal = number | string | characterConstant |
29      symbolConstant | arrayConstant.
30  arrayConstant = "#" array.
31  array = "(" {number | string | symbol | array |
32      characterConstant} ")".
33  number = [digits "r"] ["-"] bigDigits ["." bigDigits]
34      ["e" ["-"] bigDigits].
35  string = "'" {character | "''" | '"'} "'".
36  characterConstant = "$" character | "$" "'" | "$" `"`.
37  symbolConstant = "#" symbol.
38  symbol = unarySelector | binarySelector | keyword {keyword}.
39  identifier = letter {letter | digit}.
40  character = selectorCharacter | letter | digit |
41      "[" | "]" | "{" | "}" | "(" | ")" | "^" | ";" | "$" |
42      "#" | ":".
43  selectorCharacter =   "," | "+" | "/" | "\" | "*" | "~" |
```

```
44      "<" | ">" | "=" | "@" | "%" | "|" | "&" | "?" | "!".
45  letter = capitalLetter |
46      "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
47      "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
48      "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z".
49  capitalLetter =
50      "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
51      "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
52      "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z".
53  digits = digit [digit].
54  digit =   "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
55      "8" | "9".
56  bigDigits = bigDigit {bigDigit}.
57  bigDigit = digit | capitalLetter.
58  comment = `"` {character | "'"} `"`.
```

```
array                30    -31   31
arrayConstant        29    -30
bigDigit             56    56    -57
bigDigits            33    33    34    -56
binaryExpression     11    -16   18    22    23
binaryMessage        13    16    17    -21
binarySelector       3     21    -26   38
block                10    -24
capitalLetter        45    -49   57
cascadeMessage       9     -13
character            35    36    -40   58
characterConstant    28    32    -36
comment              -58
digit                39    40    53    53    -54   57
digits               33    -53
expression           7     7     -8    10
expressionSeries     2     -7    24
identifier           25    27    -39
keyword              4     4     22    23    -25   38    38
keywordExpression    12    -18
keywordMessage       14    19    -22
letter               39    39    40    -45
literal              10    -28
messageExpression    9     -11
messagePattern       1     -3
method               -1
number               5     28    31    -33
primary              9     -10   15    16    18    21    22    23
primitiveNumber      1     -5
```

selectorCharacter	26	26	40	-43			
string	28	31	-35				
symbol	31	37	-38				
symbolConstant	29	-37					
temporaries	1	-6					
unaryExpression	11	-15	16	21			
unaryMessage	13	15	15	-20			
unarySelector	3	20	-27	38			
variableName	3	4	4	6	8	10	24

Appendix 2 - Primitive Methods

How Primitive Methods Work

Computing is done in a Smalltalk system by the population of objects sending messages to each other. But in this continuing flow of messages, where is the useful work performed? The answer is, in primitive methods. Primitive methods perform low-level functions such as arithmetic operations, indexed instance variable access and device access. They are also used for higher-level but performance critical methods such as stream access and block transfers.

Primitive methods are identified with an integer primitive number enclosed in angle brackets following the message pattern. For example, the implementation of the subscripting method at: in class Object is as follows:

```
at: index
    <primitive: 60>
    ^self primitiveFailed
```

Primitive methods have two parts: (1) an assembly language part and (2) a Smalltalk part. The assembly language part is identified by the number following "primitive:" in angle brackets. The Smalltalk part follows the angle brackets.

The assembly language part of a primitive is executed first. It concludes by either succeeding (returning an object that is the method result) or failing. If the assembly language part fails, the Smalltalk part is executed to return the method result. This split responsibility is often used as follows. The assembly language code handles the most common but simple cases for efficiency. Because Smalltalk is much easier to write than assembly language, the Smalltalk code handles the infrequent but complex cases.

User Defined Primitive Methods

A complete example of a user defined primitive appears at the end of this appendix.

1. Software Interrupts

User defined primitives are implemented using software interrupts. Primitive numbers 90 through 97 are available for user definition. These are mapped onto DOS software interrupts 60 through 67, respectively. The program containing the user primitives is linked separately from Methods. The two programs are executed in series. First the program with user primitives is executed to install the software interrupts. It places the 20-bit addresses of the interrupt routines into the associated interrupt vectors (using INT 21H). It then terminates with a DOS "exit-and-stay-resident" interrupt (INT 27H). Then Methods (file "methods.exe") is executed.

2. Object Pointers

Methods uses 16-bit object pointers. Even-valued pointers identify objects of class SmallInteger, where the integer value is the pointer value divided by two. These represent integers in the range -16,384 to 16,383. Odd-valued pointers address other objects. The object pointer is used to index an object table which contains the 20-bit address of the object. An assembly language macro is provided for converting object pointers to object addresses.

Methods uses a tree-walking asynchronous garbage collector. When an object pointer is stored, the tree being walked is changed. It is necessary to notify the garbage collector of such changes. An assembly language macro is provided for this purpose.

Certain object pointer values are permanently fixed so that primitives may test for specific objects or classes. The fixed pointers are defined in file "fixedptrs.usr" on the Methods image diskette. The objects defined in this file are the following:

(1) The objects true, false and nil

(2) The classes Array, Character, FCB, LargeNegativeInteger, LargePositiveInteger, Point, SmallInteger, String,

Symbol and UndefinedObject.

(3) The system dictionary, Smalltalk.

(4) The instances of class Character, with ASCII values 0 through 255.

3. Accessing Objects Within Primitives

Smalltalk methods cannot corrupt object memory because an object is not allowed to access outside itself. Primitive methods can access all of 8088/8086 memory and therefore have the opportunity to corrupt object memory. The implementor of a primitive has the responsibility to guarantee that only the instance variables of the receiver object are changed. If you have a primitive which stores outside its receiver object into another object in the Methods image, discard the image.

The first 16-bit word of each object contains the object size rounded up to be even. This is followed by the object's instance variables. Instance variables are 16 bits if they contain pointers or words, or 8 bits if they contain bytes.

4. Macros

Assembly-language macros are provided to simplify the writing of user primitives. These handle all the details of interfacing with the Smalltalk interpreter. The macros are contained in file "access.usr" on the Methods image diskette. The following macros are provided.

(1) enterPrimitive
This must be the first instruction in a user primitive. It saves key registers on the stack and sets register BP to address the stack. After enterPrimitive:

- register DI contains the receiver object pointer
- [BP+16] contains the last argument object pointer
- [BP+18] contains the next to last argument object ptr
- etc ...

(2) exitWithSuccess numberOfArguments
This macro is used to exit the primitive when it has

successfully computed an object pointer as method result. The macro argument is the number of arguments to the Smalltalk method implemented by the primitive. A unary method has no arguments, a binary method 1, etc. Register BX must contain the object pointer of the method result.

(3) exitWithFailure numberOfArguments
This macro is used to exit the primitive when it cannot compute the method result. An example is, a floating point arithmetic primitive, where the argument is not the same class as the receiver. The macro argument is the number of arguments to the Smalltalk method implemented by the primitive.

(4) getObjectAddress objectPtrReg,offsetReg,segmentReg
This macro returns the 20-bit address of an object, given its 16-bit object pointer. The registers objectPtrReg and offsetReg must be index registers, whereas segmentReg must be a segment registers. The object pointer is input to the macro in objectPtrReg. The object address is returned as segmentReg:offsetReg.

(5) getClass objectPtrReg,classPtrReg,workSegmentReg
This macro, given an object pointer in a register, returns the class pointer in a register for the class of the object. A work segment register is identified with workSegmentReg.

(6) markPtr objectPtrReg,workSegmentReg
This macro is used for communication with the asynchronous garbage collector whenever an object pointer is stored. The object pointer stored is passed to the macro in objectPtrReg. A work segment register must be identified with workSegmentReg.

(7) isSizeEven objectPtrReg,workReg,workSegmentReg
Objects are allocated in an even number of bytes, even though their contents may be an odd number of bytes. The first word of an object contains its (even) allocated byte size. This macro tests the actual size of an object. It sets the condition code zero if the size is even and non-zero if the size is odd.

5. Example Primitive

The following is an assembly-language listing of an example of a user defined primitive for class String method at:. This also appears as file "example.prm" on the Methods image diskette.

```
PAGE 85,132
TITLE User Defined Primitive #90 (INT 60H)

;       This is an example of a possible user defined primitive.
;       The primitive below "primitive90" is an implementation
;       of the String at: primitive.
;
;       The first part of the code "installs" the primitive
;       "segment and offset" address into INT 60H vector.
;       The second part, "primitive90" defines the code that
;       will be executed when primitive number 90 from "METHODS"
;       is invoked.


PAGE
                INCLUDE fixdptrs.usr
IF1
                INCLUDE access.usr
ENDIF


PAGE
code            SEGMENT
                ASSUME CS:code,DS:code,ES:code
                ORG 100H

; This is primitive "installation code" with the exit and
; stay resident

                MOV     AX,CS                   ;set DS to address code segment
                MOV     DS,AX
                MOV     DX,OFFSET primitive90   ;load DX with primitive offset
                MOV     AX,2560H                ;use DOS to set address of INT 60H
                INT     21H
                LEA     DX,primitive90end       ;load DX with address beyond last
                INT     27H                     ;byte - exit and stay resident
```

```
PAGE
primitive90 PROC    FAR
            ASSUME CS:code,DS:nothing,ES:nothing

; This is code that will be executed everytime primitive
; number 90 is invoked from "METHODS"

            enterPrimitive                  ;enter primitive macro (see macro
                                            ;listings)

            MOV     CX,DI                   ;copy receiver in CX

;           getClass DI,DX,ES               ;put class of receiver in DX
;
;           CMP     DX,ClassStringPtr       ;if class of receiver is not
;           JNE     failure                 ;String then primitive fails
;                                           ;(This check is commnet since it is
;                                           ;not necessary but it is shown here
;                                           ;as an example on how to use the
;                                           ;getClass macro

            MOV     DI,[BP+16]              ;put index argument in DI

            SHR     DI,1                    ;shift right and
            MOV     AX,DI                   ;copy in AX
            JNC     smallInt                ;if no carry (even number) then
            RCL     DI,1                    ;its a small integer and jump else
                                            ;restore the index object ptr

            MOV     DX,SS                   ;set ES to address classes segment
            SHR     DX,1
            MOV     ES,DX

            CMP     WORD PTR ES:[DI],ClassLargePosInt ;if class of index object
            JNE     failure                 ;pointer is LargePositiveInteger
                                            ;then continue else primitive fails

            getObjectAddress DI,BX,ES       ;get address of index object ptr

            CMP     WORD PTR ES:[BX],4      ;size of this large positive
            JNE     failure                 ;integer must be 4 (2 bytes for
                                            ;size header + 2 bytes for the
                                            ;number itself) - not 4 suggests
                                            ;that number is more than 2 bytes
                                            ;long and primitive fails else

            MOV     AX,ES:[BX+2]            ;move 2 byte integer into AX and
            JMP     haveIndex               ;jump to haveIndex

smallInt:   JLE     failure                 ;if after shift sign of integer
                                            ;changes or its zero then
                                            ;primitive fails (index must be
                                            ;greater than zero)

haveIndex:  INC     AX                      ;index +1 to correct byte access
            JZ      failure                 ;later on; if value goes to 0 then
                                            ;index was too large and primitive
                                            ;fails
```

```
                MOV     BX,CX                   ;set BX to receiver object ptr and
                getObjectAddress BX,DI,ES       ;get address of receiver (string)

                isSizeEven BX,CX,DS             ;check even/odd length of string
                                                ;object (JZ for even length)

                MOV     BX,ES:[DI]              ;move size of string to BX

                JZ      even                    ;if isSizeEven result is zero
                DEC     BX                      ;condition flag then length
                                                ;of string is even (BX - 2 bytes
                                                ;for size header) else length of
                                                ;string is odd (BX - 3) so adjust
                                                ;size of string (BX) by -1

even:           CMP     AX,BX                   ;adjusted index must be less than
                JAE     failure                 ;BX (string size) else primitive
                                                ;fails

                                                ;now we must get the character
                                                ;from the string and turn it into
                                                ;an object pointer
                XOR     CH,CH                   ;set CH to zero
                MOV     BX,AX                   ;set BX to be adjusted index
                MOV     CL,ES:[DI+BX]           ;move to CL the char from string
                SHL     CX,1                    ;multiply CX by 2
                ADD     CX,FirstCharacterPtr    ;add to CX the base character ptr

                MOV     BX,CX                   ;set BX to result char pointer
                exitWithSuccess 1               ;and enter into successful exit
                                                ;macro with arg count 1 (the index)

failure:        exitWithFailure 1               ;failure macro with arg count 1

primitive90 ENDP

primitive90end LABEL BYTE                       ;label for exit and stay resident

code            ENDS
                END
```