

# Smalltalk Kernel Language Manual

BY Larry Tesler

SEPTEMBER 1977

Smalltalk is a programming language that deals with *objects*. Every object has its own data. Objects communicate by sending and receiving *messages* according to simple *protocols*.

Objects are grouped into *classes*. The objects of a class are called its *instances*. Each class specifies the protocols its instances will know as well as the *methods* they will use to respond to messages they receive.

The standard facilities of the Smalltalk system include i/o protocols for disk, display, keyboard, mouse, and Ethernet; basic data structures such as numbers, strings, arrays, and dictionaries; basic control structures such as loops and processes; program editing, compiling, and debugging; text editing, illustration, music, animation, and information storage and retrieval.

Smalltalk applications include education, personal computing, and systems research.

This interim reference manual covers the semantics and the present syntax of the Smalltalk kernel language. It is assumed that the reader has seen Smalltalk in operation and has programmed computers in Algol-like languages. Familiarity with LISP, Simula, and/or Smalltalk-72 should be helpful but is not necessary. System operation is described in the memo, *Browsing and Error Analysis*, available under separate cover from the Learning Research Group, Systems Science Laboratory.

**XEROX**

PALO ALTO RESEARCH CENTER

3333 Coyote Hill Road / Palo Alto / California 94304

## Disclaimer

The Smalltalk kernel system is not yet ready for general release. We ask that you do not copy and distribute either the manual or the system at this time.

An interim version of the kernel is available for limited release. Potential users of the limited release should apply to the Learning Research Group for additional documentation and for a copy of the system.

All interactive programming systems have an inner and an outer face. Our general strategy for building a programming system has been to concentrate on making the foundations as strong as possible, and then to build a communications facility that allows the user to browse, control, and extend the foundation.

The foundation of the newest Smalltalk system includes a large virtual memory that can support tens of thousands of Smalltalk objects, the equivalent of two million bytes of storage. For compatibility with the rest of the Alto world, the kernel supports a file system external to the virtual memory.

Built into the limited release are the following facilities:

- a microcoded interpreter for compact code syllables that efficiently implement inter-object message communication;
- a scheduling facility for both real-time and relative control of communication between objects;
- graphics creation including text, line drawing, painting, and font editing;
- text editing facilities with multiple typefaces and multiple documents;
- interactive programming and debugging, with on-line access to system source programs in the Smalltalk language.

Also implemented, but not yet in a useful form for release are:

- an Ethernet communication system;
- a multiple-strategy information storage and retrieval kernel.

The present document-handling facilities are not completely unified. A more unified version is currently under development. In the coming months, most of the user-communication facilities will be replaced by a simpler and more comprehensive design.

The syntax described in this manual is an interim offering. Compared with the syntax of Smalltalk-72, it is unambiguous and therefore easy to compile into compact code syllables. Most programmers have less difficulty with syntax errors than in previous versions. However, there are still weaknesses that must be corrected, especially if it is to be taught to children or adults as a first programming language. When the new syntax is released, old programs will be translated to it mechanically.

The present compiler is also an interim offering. For most purposes, it provides satisfactory service: users have experienced edit-compile-test-debug-edit cycles of at most a few minutes. However, for certain purposes its performance is unsatisfactory, so it will soon be replaced by one that is even faster and provides more helpful error indications.

## Table of Contents

<b>Introduction</b>		
	I History	i
<b>1. Semantics</b>		
	1.1 Classes	1
	1.2 An archetypal class	4
	1.3 Instances	5
	1.4 Messages	5
	1.5 Methods	6
	1.6 Subclasses	7
<b>2. Expressions</b>		
	2.1 Literals	9
	2.2 Special constants	10
	2.3 Variables	11
	2.4 Self-references	11
	2.5 Expressions	12
<b>3. Messages</b>		
	3.1 Message forms	13
	3.2 Selectors	13
	3.3 Message dictionaries	14
	3.4 Computed selectors	14
	3.5 Precedence	15
	3.6 Remote evaluation	15
<b>4. Statements</b>		
	4.1 Blocks	17
	4.2 Cascading	17
	4.3 Conditionals	18
	4.4 Loops	20
<b>5. Definitions</b>		
	5.1 Method definitions	23
	5.2 Class definitions	25
	5.3 Instance creation and initialization	26
<b>6. Contexts</b>		
	6.1 Contexts	27
	6.2 Processes	27
<b>Appendices</b>		
	I. Predefined classes	29
	II. Syntax summary	47
	III. Character Set	49

## A Short History of Smalltalk

### *Influences.*

The Dynabook project and its language, Smalltalk, have a heritage that stretches back into the 1960's. The first really self-contained personal computer was the LINC of Wes Clark. The FLEX machine of Alan Kay and Ed Cheadle was the first design of a simulation-oriented higher-level-language personal computer.

The design of the FLEX language was most strongly influenced by the *class* and *instance* ideas of Simula I, and by SketchPad and the Burroughs B5000. In addition, FLEX used ideas found in the FSL compiler-compiler, JOSS, Euler, and LEAP. A subsequent design of the FLEX system reflected additional inspiration from the IMP extensible language, LISP, the Project Genie 940 operating system, and the GRAIL system.

In 1970-71, Alan Kay produced two unpublished language designs, Novala and Smalltalk-71. Novala was an attempt to find a common and simple semantics for both FLEX and the PLANNER language. Smalltalk-71 was also a goal-oriented language, but it was designed for children. It sought to combine the simplicity of LOGO with the generality of pattern-based systems.

Smalltalk-71 and the pattern-based system, LISP70, were designed concurrently and had considerable influence on each other. Eventually, it was decided that the pattern-directed approach was too "syntactic" and required more control of programming style than most novice programmers are able to muster.

### *Smalltalk-72.*

The next system, Smalltalk-72, was designed by Alan Kay and implemented by Dan Ingalls. The language was a "return to the well", to the object-oriented semantics of Simula and the communication ideas of the FLEX machine. By this time, Simula-67 had appeared in this country. It was a more comprehensive design than its predecessor. Although Simula-67 introduced the important new idea of *subclassing*, its Algol heritage enforced strong typing of variables and a rigid calling discipline. The FLEX scheme of message-oriented communication between processes liberated some of the Simula ideas, and added to them a simple framework for user-extensible syntax, interactive programming, and an integrated user interface. The extensible control structure ideas of Dave Fisher's CDL language design were a major influence on Smalltalk-72.

The communications basis of Smalltalk-72 has endured through many subsequent designs and through the use of the system by hundreds of children and adults. The system itself has been continually redesigned and rebuilt by the Learning Research Group.

### *The Smalltalk kernel.*

By 1976, considerable systems had been built in Smalltalk. A style of programming had been established and how the system was used by a variety of people had been studied. Ideas for a qualitatively different successor to Smalltalk-72 were germinating. However, many projects had been "backburnered" because of system limitations, especially virtual capacity and performance. In that year, we decided to begin the design and construction of

a quantitatively improved follow-on to Smalltalk-72. It was to be strong, fast, and capacious enough to handle the backburnered projects, as well as to support experiments leading to a qualitative successor. This system is now known as the Smalltalk *kernel*.

The ideas, designs, and implementation of the kernel are due to the entire group that worked on it. Every person made major contributions. The absence of any of them would have been detrimental to the success of the system.

Those who had significant influence on the design of the Smalltalk kernel were (in alphabetical order):

Alan Borning, Adele Goldberg, Laura Gould, Chris Jeffers, Ted Kaehler, Alan Kay, Diana Merry, Dave Robson, John Shoch, Larry Tesler, and Steve Weyer, *Dan Ingalls*

The implementation of the kernel was due to Dan Ingalls, Ted Kaehler, Diana Merry, and Dave Robson. Since its completion in June, it has undergone a number of improvements, and has seen contributions from a number of PARC regular and temporary employees.

### References.

#### Wes Clark's LINC (ca. 1963-4)

W. Clark and C. Molnar, *A Description of the LINC*, Computers in Biomedical Research II, Academic Press, New York, (1965)

#### FLEX machine design (1967-69) Alan Kay and Ed Cheadle

Alan Kay, *FLEX*, *A FLEXible EXtensible Language*, Univ. of Utah Dept. of Comp. Sci., Tech. Rep. 4-7 (May 1968)

Alan Kay, *The Reactive Engine*, PhD Thesis, Univ. of Utah Dept. of Comp. Sci., (Sept. 1969)

#### Simula I (1965).

O. Dahl, K. Nygaard, *SIMULA -- an ALGOL-Based Simulation Language*, CACM, (Sept. 1966)

#### SketchPad (1962).

I. Sutherland, *Sketchpad: A Man-Machine Graphical Communication System*, MIT Linc. Lab., Tech. Rep. 296, (Jan. 1963). Excerpts in Proc. SJCC (1963)

#### The B5000 (1961) Burroughs Corporation.

R. S. Barton, *A New Approach to the Functional Design of a Digital Computer*, Proc. Western Joint Comp. Conf., (May 1961)

W. Lonergan, P. King, *Design of the B5000 System*, Datamation, (May 1961)

Burroughs Corporation, *Burroughs B5500 Information Processing System Reference Manual*, (1964)

#### FSL compiler-compiler (1964).

Jerome Feldman, *A Formal Semantics for Computer Languages and Its Application in a Compiler-Compiler*, Carnegie PhD Thesis, (1964). Excerpts in CACM (Jan. 1966)

JOSS (1962-3) The RAND Corporation.

Cliff Shaw, *Joss: A Designer's View of an Experimental On-Line Computing System*, AFIPS Conf. Proc., XXXVI, 1 (Fall, 1964).

Euler (1965)

N. Wirth, H. Weber, *Euler -- A Generalization of ALGOL, and its Formal Definition: Part I, Part II*, CACM, (Jan., Feb. 1966)

LEAP (1966) MIT Lincoln Laboratory.

Jerome Feldman, *Aspects of Associative Processing*, Tech. Note 1965-13, MIT Linc. Labs, (April 1965)

J. Feldman, P. Rovner, *The LEAP Language and Data Structure*, Tech. Note DS-5436, MIT Linc. Labs., (Oct. 1967)

IMP Extensible Language (1965-7).

E. T. Irons, *Experience With an Extensible Language*, CACM, (Jan. 1970)

LISP (1959) MIT.

John McCarthy, *Recursive Functions of Symbolic Expressions and Their Computation By Machine*, CACM, (April 1960)

John McCarthy, *The LISP 1.5 Programmer's Manual*, MIT Press, (Feb. 1965)

Project Genie 940 Operating System (ca. 1965-6) Univ. of Cal. (Berkeley)  
Lampson, Lichtenburger, Pirtle, Deutsch, Barnes.

B. W. Lampson, W. W. Lichtenburger, and M. W. Pirtle, *A User Machine in a Time-Sharing System*, Proc. IEEE, (Dec 1966)

GRAIL (ca. 1966-68) The RAND Corporation.

T. Ellis, J. Heafner, W. Sibley, *The GRAIL Project: An Experiment in Man-Machine Communications*, RAND, RM-5999-ARPA, (Sept. 1969)

PLANNER (ca. 1968).

C. Hewitt., *PLANNER -- A Language For Proving Theorems in Robots*, Proc. IJCAI, (Sept. 1969)

G. Sussman, T. Winograd, E. Charniak, *Micro-Planner Reference Manual*, MIT AI Memo 203, (July 1970)

LOGO (ca. 1967) MIT and Bolt Beranek and Newman, Inc.

Feurzig, W., et. al., *Programming Languages as a Conceptual Framework for Teaching Mathematics*, Final Report on BBN Logo Project, (June 1971)

LISP70 (1971) Smith, Enea, Tesler. Stanford A.I. Project.

L. Tesler, H. Enea, D. Smith., *The LISP-70 Pattern Matcher*, Proc. IJCAI, (Sept. 1973)

**SIMULA-67 (1967)**

O. Dahl, B. Myhrhaug, K. Nygaard, *SIMULA--Common Base Language*, Norwegian Computing Center, Oslo, Norway (1970)

**CDL (1969)**

David Fisher, *Control Structures for Programming Languages*, PhD Thesis, Carnegie-Mellon University, (May 1970)

**The Dynabook and Smalltalk-72 (1971-2).**

Alan Kay, *A Personal Computer For Children of All Ages*, Proc. ACM Nat'l Conf., (Aug. 1972)

Alan Kay, *A Dynamic Medium for Creative Thought*, Proc. NCTE, (Nov. 1972)

Alan Kay, Adele Goldberg (Ed.), *Personal Dynamic Media*, Learning Research Group, Xerox Palo Alto Research Center, (Mar. 1975)

Alan Kay, *Personal Computing*, Conf. on 20 Yrs. Comp. Sci., Univ. of Pisa, Pisa, Italy, (June 1975)

A. Goldberg, A. Kay, *The Smalltalk-72 Instructional Manual*, Learning Research Group, Xerox Palo Alto Research Center, (June 1976)

A. Kay, A. Goldberg, *Personal Dynamic Media*, IEEE Computer, (March 1977)

Alan Kay, *Microelectronics and the Personal Computer*, Scientific American, (Sept. 1977)

## Chapter 1: Semantics

The Smalltalk programming language deals with *objects*. An *object* has both *methods* (procedures) and *state* (data). Objects can communicate with each other by sending and receiving *messages* according to definable *protocols*. An object is an independent activity that maintains its own state and communicates on its own terms with other objects. Therefore, it has the properties of a complete computing machine.

The notion of objects pervades Smalltalk. An array is an object. A number is an object. The operator's screen is an object, and so is each window displayed thereon. If the operator is creating a document, the document is an object, as are its component parts such as paragraphs and illustrations. If the operator is composing music, the score, each of its voices, and each of their notes are objects, as are the instruments and their timbres.

### 1.1 Classes.

A single Smalltalk environment contains tens of thousands of objects. The computer representations of all these objects requires several hundred thousand words of storage. To make it practical for the programmer and the system to deal with such a large space of objects, objects that share certain common properties are grouped into families. As in the language Simula, a family of objects is called a *class*. The objects of a class are called its *instances*.

By convention, the name of a class begins with a capital letter both in programs and when it is used as a proper noun, but not when it is used as a common noun. Thus, it would be proper to assert that "the integer 35 is an instance of class Integer". There are other spelling conventions in the system, but unfortunately, they are not presently followed in a consistent manner.

Over fifty classes are predefined in the standard Smalltalk system. Among the classes you should know about are:

#### Integer

The instances of class Integer are whole numbers between -32768 and +32767 that also can be treated as 16-bit values or (if small) as character codes.

#### Float

The instances of class Float are exponent/mantissa representations of numbers. The precision is about 9 decimal places, and the exponent range is +16000.

#### LongInteger

The instances of class LongInteger are whole numbers of any reasonable precision.

#### Point

A point is an x-y pair of numbers. It often (but not necessarily) represents a location on the display screen. Arithmetic can be performed on two points; a new point is returned whose x and y are the result of the specified arithmetic performed on the x and y of the original points.

**Rectangle**

A rectangle is a pair of points that often (but not necessarily) represent a rectangle on the display screen whose upper left corner (called the "origin") and lower right corner (called the "corner") are the two points. Arithmetic can be performed on two rectangles; a new rectangle is returned whose origin and corner are the result of the specified arithmetic performed on the origin and corner of the original rectangles. If the rectangle corresponds to an area of the display screen, then the rectangle can operate on that image in various ways.

**String**

A string is a sequence of zero or more eight-bit bytes, i.e., integers between 0 and 255 that can be used to represent text characters.

**Substring**

A substring is a reference to a list (a "map") of positions in some ("mapped") array (often, but *not necessarily*, a string).

**Interval**

An interval is an arithmetic progression defined by a start-number, a step-number, and a stop-number, e.g., *1 to: 5 by: 2* represents the progression 1, 3, 5.

**Paragraph**

A paragraph is an array of characters each of which is assigned a typeface. The paragraph as a whole has a specified *alignment*: flush left, flush right, centered, or justified. Characters can be inserted or deleted from the paragraph. Typefaces can be changed for any sequence of characters. The alignment can be changed. The paragraph can be converted to and from Bravo representation.

**Vector**

A vector is a one-dimensional array of references to objects, some of which may themselves be vectors.

**Stream**

A stream is a reference to some position in some array, such as a string or a vector. It can quickly advance or retreat through the array, getting, putting, or skipping the elements passed.

**File**

This class and the next are included to provide compatibility with the rest of the Alto world. A file is an object that accesses an Alto disk file on one disk or the other. Access is normally sequential, but random and multi-page access are provided as well.

**Directory**

A directory is an object that accesses an Alto disk file directory (e.g., 'sysdir' on dp0).

### UserView

There is currently a single instance of class UserView, accessed through the global variable, *user*. That instance can interact with the user's terminal in various ways.

### Turtle

A turtle is a drawing pen filled with black, white, or complementing ink, with a penpoint from 1 to 8 dots thick that can draw on the display screen. It "crawls" around the screen under the direction of messages sent to it. (It is sometimes said that Smalltalk is "like LOGO"; actually, the main similarity between the two educational languages is the availability of turtle graphics. In Smalltalk, there may be of course many independent instances of class Turtle in existence at a time.)

### Textframe

A text frame allows paragraphs to be displayed on the screen, typeset to a specified width and clipped on a specified boundary.

### Dispframe

A display frame is a window on the display screen through which a teletype-style dialogue can be carried on. There is typically a single display frame, managed by *user* (the only instance of class UserView). It can prompt for user input, read the input (e.g., 3+4), evaluate it, and print the last value computed (e.g., 7).

### FontWindow

A font window is a window on the display screen in which one character at a time out of a font (typeface) can be edited. The character is displayed blown up so that each dot in its dot-matrix representation can be pointed at by the user and set to black or white. Through a menu one can change the width of the character.

### BitRect

A bit rectangle is a rectangular matrix of black and white dots that can be displayed on the screen and in which the user can paint ("edit").

### Cursor

A cursor is a 16 by 16 matrix of black and white dots suitable for display as the Alto screen cursor. It can install itself as the current screen cursor during the execution of a given program and can remember and restore the previous cursor when the program is done.

### Menu

A menu is a short list of short one-line text entries that can be displayed in a temporary frame on the screen so that the user can select one of the entries and invoke an associated action.

## Class

Each Smalltalk class is itself an instance of the class, `Class`. Each class "manages" its own instances and messages, that is, it can create new ones and can enumerate the ones that exist.

## ClassOrganizer

As an aid in dealing with the large number of classes in Smalltalk and the variety of messages which they recognize, a facility is provided for *organizing* them into *categories*. The predefined classes are organized into the following categories: Kernel classes; Numbers; Basic data structures; Sets and Dictionaries; Graphical objects; Text objects; Browser and Debugger; Files and Compiler; Primitive access. The messages of a class are, by convention, organized into such categories as Initialization, Access to Parts, and Private. "Private" messages are intended to be invoked only by methods of the same class.

## Context

To respond to a message, an object executes a *method*. To do so, it needs to have space in which to work, needs to keep track of its progress through the method, and needs to remember who sent the message so it can reply when it is done. All these needs are serviced by an instance of class `Context`, one of which is created almost every time any object receives a message. A context also provides help in debugging programs and in constructing control structures such as loops, processes, and coroutines.

### 1.2 An archetypal class.

For the sake of discussion in this manual, reference will often be made to class `Dictionary`. You should know about it:

## Dictionary

A dictionary associates with each of a set of objects (usually all of the same class) a corresponding value. It can tell what value is associated with a given object and can insert, delete, and change associations.

A particular instance of class `Dictionary` might contain the following five nonsensical associations:

<u>objects</u>	<u>values</u>
'six'	6
'twelve'	12
'+'	'plus'
'Fastalk'	'slow'
'Smalltalk'	'fast'

Let us examine this instance from two viewpoints, the *internal* viewpoint of its representation, and the *external* viewpoint of its protocol (i.e., the messages it recognizes). The two viewpoints are quite different. We will then examine *methods*, the link between the two.

### 1.3 Instances.

From the internal viewpoint, an instance of class Dictionary is composed of two *fields* called *objects* and *values*. Both fields reference vectors whose lengths are the same power of 2. If each vector has a length of, say, 8, and the dictionary has 5 associations, then 3 of the locations in each vector will be vacant. Vacant locations are represented by a reference to the special constant *nil*.

Storage in the Smalltalk system is managed with the aid of *reference counts*. For any object to thrive and occupy space in a Smalltalk environment, one or more *references* to it must exist in *variables* of other thriving objects. If the variable named *dict* references a Dictionary instance, then that instance and the two vectors it references will thrive. However, if the dictionary ceases to be referenced by *dict*, then (assuming no other variable references it) it will be *deallocated*, that is, it will disappear and its storage space will be reclaimed for other uses. If the two vectors referenced by the dictionary are referenced nowhere else, they will be deallocated as well.

### 1.4 Messages.

From the external viewpoint of another object wanting to access a dictionary, its fields can be reached only indirectly, by sending messages to the instance. For example, if *dict* is a variable that references our sample dictionary, then one can send a look-up message to the dictionary with the Smalltalk expression:

```
dict lookup: 'twelve'
```

and the value returned will be *12*. Note that *dict*, the recipient of the message, is written first, followed by the message itself. The details of syntax of the Smalltalk language are presented in Chapters 2 through 5.

From the external viewpoint, it is not essential to know that a dictionary is represented by a pair of vectors and that their length is a power of 2. If class Dictionary is redefined to use a different representation, and if its methods are modified appropriately, then all other methods that use dictionaries will continue to function.

It is possible that instances of several classes are able to respond to the same message. To the extent that two or more classes share that ability, it is said that they *implement the same protocol*. A given class may have part of its protocol in common with one class and part in common with another.

To send a certain message to an object, the object must be of the "right type"; that is, the object's class must be able to respond to that message. Although the notion of *class* is formalized in Smalltalk, the notion of *type* is not. Types are not declared or even named.

In Smalltalk, as in LISP and APL, a given variable may reference an object of any class and of any type. Therefore, if a programming error causes an unintended object to be assigned to a variable, the error is not detected when the program is compiled, nor even when the assignment occurs, but somewhat later when an uncomprehended message is sent to the object.

The absence of type qualification has its benefits. It permits the notation to be more concise. It enables old programs to operate on instances of new classes, to the extent that those classes follow old protocols. It facilitates the availability of a small, reliable, and easily extended resident compiler, thereby permitting direct execution of expressions both for debugging programs and for invoking them.

### 1.5 Methods.

Now that we have examined an instance from both the internal (concrete) viewpoint of instance fields and the external (abstract) viewpoint of message protocols, we come to the link between these two viewpoints, the *method*. A class only understands a message for which a method has been specified. For example, class Dictionary employs the following method (explained below) to respond to *lookup:* messages:

```
lookup: name | x "the value corresponding to name, if any, else false"
           [x ← self find: name⇒ [↑values*x] ↑false]
```

Smalltalk programs are easier to discuss if the special characters are pronounced as follows:

	with temporary	"	comment
[	begin	←	gets
⇒	then only	↑	return
.	sub	]	end

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 a *temporary variable* called *x* that is not an argument but is used within the method to assist its execution.

The body of a method is a *block* of Smalltalk *statements* that provides a procedural implementation of the method. 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 *true alternative* enclosed in *[* and *]*, and a *false alternative*. In the present example, its three parts are:

```
x ← self find: name
```

An object can send itself a message by addressing it to the pseudo-variable *self*. In this statement, the message *find: name* is sent to the dictionary itself. That message is a predefined one that returns the location of *name* within *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 the conditional statement is considered false if the value of the assignment was *false*, and is considered true otherwise.

```
↑values*x
```

This statement is executed if and only if the condition was true. It sends the message *\*x* to the vector *values*. That message is a predefined one that returns the *x*th element of the vector. The *↑* symbol causes the result to be returned as the value of the current method.

```
↑false
```

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

## 1.6 Subclasses.

When several classes implement the same protocol (i.e., respond to the same messages), they may or may not employ the same methods to do so. If it happens that the same methods are employed, it would be awkward to have to repeat their definitions in each class. Therefore, a limited facility for sharing method definitions among classes is provided. A class may have *subclasses*. Take as an example the standard Smalltalk class, `HashSet`:

### HashSet

A hash set is a set of objects that are usually of the same class. No object appears more than once in the set. A hash set can tell whether a given object is present and can insert and delete objects.

Class `Dictionary` (discussed earlier) is actually a subclass of class `HashSet` which associates a value with every object. A hash set has only *objects*; it has no *values*. Therefore, there is no *lookup: name* message. However, there is a *find: name* message that returns the location of *name* in *objects*. Therefore, class `Dictionary` need not define a method for the *find:* message; it inherits one from `HashSet`. It also inherits the field *objects*, so its class definition need only mention the field *values*. The definitions of the two classes are:

```
Class new title: 'HashSet';
    fields: 'objects'

Class new title: 'Dictionary';
    subclassof: HashSet;
    fields: 'values'
```

The syntax of definitions is covered in chapter 5. However, it should be explained here that newly defined names appear in quotes (e.g., *'values'*), while previously defined names do not (e.g., *HashSet* in the definition of class `Dictionary`).

The following are additional predefined Smalltalk classes that you should know about. Some of them are subclasses or superclasses of the ones listed earlier. A couple of them are *abstract*, that is, they have messages shared by their subclasses, but they do not have any instances of their own.

### Object

Every class is ultimately a subclass of class `Object`. Thus, every instance of every class can inherit messages from class `Object`, such as the ability to create a copy of itself and the ability to create a window through which the user can inspect its state.

### VariableLengthClass

Classes like `Vector` and `String` are unusual in that their instances have numbered elements instead of named fields. Such classes are themselves instances of class `VariableLengthClass`, a subclass of class `Class`. Each variable length class manages its own instances and messages.

### Number

Class `Number` is an abstract superclass of classes `Integer`, `Float`, and `LongInteger`. The messages of the abstract class `Number` do things that don't require knowing the number representation, e.g., it can return the larger of itself and another number after calling upon a representation-dependent message to compare its own magnitude with that of the other number.

### UniqueString

Class UniqueString is a subclass of class String. A unique string is an entry in a certain global set in which no two entries are equal to each other. It is similar to an "atom" in the language LISP.

### Array

Class Array is an abstract superclass of Vector, Interval, String, Substring, and UniqueString. Every array is an ordered set of elements that can be accessed by positive integer subscripts. The messages of the abstract class Array do things that don't require knowing the array representation, e.g., it can produce a copy of itself with an insertion, deletion, or replacement by calling upon representation-dependent messages to calculate its length, to create a new instance, and to access individual elements.

### MessageDict

Class MessageDict is a subclass of class HashSet. Every class has a message dictionary in which to hold the messages it can understand and the methods it uses to respond to those messages. The objects of a message dictionary are the names ("selectors") of the messages. With each message are associated the source code and object code of its method.

### ObjectReference

An object reference has only one field, its *value* field, which, like all fields, contains a reference to an object. An instance of class ObjectReference has no other function but to hold that value, so it may be thought of as an "indirect reference". It can have its contents examined or reassigned much as if it were a variable, except that a message must be sent to it to make it do so.

### SymbolTable

Class SymbolTable is a subclass of class Dictionary. The values it associates with its objects are instances of class ObjectReference. In effect, it associates a "variable name" with a "variable".

### Window

A window is a rectangular area of the display screen with which is associated a user interface to a system facility. Each such facility is implemented as a subclass of class Window. The messages of class Window poll user input devices and report events to the subclass.

### ParagraphEditor

A paragraph editor handles edits to a paragraph through a window. It implements the concept of a *selection* encompassing any sequence of characters within the paragraph, and operations to edit that selection, such as *typing*, *cut*, *paste*, and *copy*.

Appendix I contains a complete list of the predefined classes in the current version of Smalltalk, and discusses in English what they can do. The Smalltalk-language versions are available both on-line (through the *browsing* facility) and in hard copy form.

## Chapter 2: Expressions

Appendix II presents the full syntax of the Smalltalk language in a formal notation. The Smalltalk character set is presented in Appendix III. The present chapter and the next three discuss the syntax informally with the aid of examples.

### 2.1 Literals.

A literal is a symbol whose value is a constant implied by the spelling. Literals only exist for a few classes: Integer, Float, String, UniqueString, and Vector.

#### 2.1.1 Integer literals

Examples:

```
0  ^940  32767  ^32766  0377  0177777  0100000
```

Incorrect:

```
0940
```

A literal of class Integer is written as an unbroken sequence of digits; if negative, it is preceded by a "high minus" sign: `^`. If the first digit is `0`, the rest are in octal radix.

#### 2.1.2 Float literals

Examples:

```
0.0  ^6.2238  31.415927e^1
```

Incorrect:

```
0.  ^6.  .31415927e1
```

A literal of class Float is written as a decimal-radix number constant immediately followed by a decimal point (a period) and one or more decimal digits. After it may be an exponent of the form `e` followed by a decimal-radix integer constant.

#### 2.1.3 String literals

Examples:

```
" 'a' 'Hi' 'They said, "Yes!"' ""
```

Incorrect:

```
"" 'They said, 'Yes!'"
```

A literal of class String is written as an arbitrary sequence of characters enclosed in apostrophes. To include an apostrophe in the string, write two in a row.

### 2.1.4 *UniqueString* literals

Examples:

`+` `↑` `,` `Help` `printon`:

Incorrect:

`)` `12`

A literal of class `UniqueString` is either a sequence of letters, digits, and colons not starting with a digit, or any other single character except a parenthesis. It is preceded by a `⌘` unless it is embedded in a vector literal (see below). The difference between a unique string and a string is that no two instances of class `UniqueString` contain the same characters. Unique strings are similar to *atoms* in the language LISP.

### 2.1.5 *Vector* literals

Examples:

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

Incorrect:

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

A literal of class `Vector` is written as an arbitrary sequence of literals enclosed in parentheses. It is preceded by a `⌘` unless it is embedded in another vector literal.

## 2.2 *Special constants.*

The following constants of class `Object` have reserved names:

- `nil`      The default initial value for a variable.
- `false`    Anything but *false* is effectively "true".
- `true`     Useful to force a "true" condition.

### 2.3 Variables.

Examples:

```
x x1 theOldVersion sm1977a
```

A variable name is a sequence of letters and digits of which the first character is a letter. Upper and lower case letters are different, e.g., *hello* is not the same variable as *Hello*.

There are four kinds of variables in Smalltalk: *shared variables*, *class variables*, *instance variables*, and *method variables*. The distinction is one of scope and lifetime.

*Shared variables* reside in *pools* which may be shared by many classes. Both the name and the storage for the value of the variable are shared by all those classes. Some pool variable names are spelled with an initial capital letter, but, by convention, names of more local variables never begin with a capital letter. A pool is an instance of class *SymbolTable*.

All classes share the *global* pool named *Smalltalk*. In that pool are found variables such as *user* (the only instance of class *UIView*), *mem* (an integer array to address main memory), *Top* (the top-level priority scheduler), and the names of all the predefined classes.

*Class variables* are variables shared by all instances of a class. They are actually pool variables in a pool that can not be shared by any other class. Note: the current compiler does not allow subclasses to share class variables.

The name of an *instance variable* is shared by all existing instances of one class, but each instance maintains its own storage for the value. Instance variables are also known as *fields*.

The name of a *method variable* is local to a method in a class. New storage for the value of the variable is allocated each time its method is invoked and is deallocated when the method is completed. There are two kinds of method variables: arguments to the message, and all others; the latter are sometimes known as *temporary variables*.

When a variable name occurs in a method, it is first looked up among method, instance, and class variables; if it is not found among them, it is sought in the pools shared by the class (in the order of declaration). If it is not found anywhere, then (in the current version of the compiler) it is inserted automatically into the special pool named *Undeclared*, with a value of *nil*.

### 2.4 Self-references.

Some standard pseudo-variables are provided automatically to every method. They can not be assigned values by the method. The most important one is *self*, the instance whose method is being performed. For example, a dictionary can tell itself to find the location in its *objects* vector of the value *name* by evaluating *self find: name*.

The default return value of a method is *self*. This default is generally employed by messages performed for effect. To override the default in messages performed for value, use the *return* statement ( $\uparrow$ ), as in the following method of class *Integer*:

```
|arg "the largest multiple of arg not greater than me"  
  [ $\uparrow$ (self/arg)*arg]
```

Other variables provided automatically are *super* (Section 3.3) and *thisContext* (Section 6.1).

### 2.5 Expressions.

The constructs discussed so far in this chapter have all been syntactic *primaries*, out of which may be constructed larger *expressions*. As in other programming languages, an expression can be *evaluated* according to certain rules, in order to yield a *value*. The value of every expression is an object (or more precisely, a reference to an object). An expression of any size can be made to serve as a syntactic primary by enclosing it in parentheses, e.g.:

(dict lookup: 'twelve')

There are three expression forms in the Smalltalk syntax:

a variable followed by an assignment arrow and an expression, e.g.:

$x \leftarrow y + 16$

just a primary, e.g.:

16

a primary followed by one or more messages, e.g.:

16 + vec length

In the first form, the expression after the arrow is evaluated, and the value is assigned to the variable before the arrow. The value of the expression also becomes the value of the assignment as a whole. Thus,  $a \leftarrow b \leftarrow 4$  will assign 4 to both  $a$  and  $b$ . An assignment to a variable often stands alone as a statement, in which case it is called an *assignment statement*.

In the second form, the primary is evaluated, and its value becomes the value of the expression.

The third form of expression sends messages to objects in order to yield a value. When an object sends a message, it waits for a response before continuing to perform its method. If asynchronous computation is desired, *processes* may be created and scheduled (see Section 6.2).

The syntax of messages is discussed in the next chapter.

## Chapter 3: Messages

### 3.1 Message forms.

There are three principal forms of message in the Smalltalk syntax: *unary*, *binary*, and *keyword*. Each form has a variation called an *assignment* form. The variety of forms are represented by the following examples.

An expression sending a no-argument "unary" message *next* to a stream *input* is:

```
input next
```

An expression sending a one-argument "binary" message *+2* to an integer *j* is:

```
j+2
```

An expression sending a one-argument binary message *\*x* to a vector *values* is:

```
values*x
```

An expression sending a one-argument "keyword" message *lookup:'six'* to a dictionary *dict* is:

```
dict lookup: 'six'
```

An expression sending a two-argument keyword message *insert: 'six' with: 6* to *dict* is:

```
dict insert: 'six' with: 6
```

An expression sending a one-argument "unary assignment" message *next ← char* to *input* is:

```
input next ← char
```

An expression sending a two-argument "binary assignment" message *\*x ← value* to *values* is:

```
values*x ← value
```

An expression sending a two-argument "keyword assignment" message *instfield: 4 ← value* to an arbitrary object *obj* is:

```
obj instfield: 4 ← value
```

Messages with more than two arguments are formed by using additional keywords. Each keyword ends with a colon. Note that a unary message has no colon. A binary message is introduced by a single non-alphanumeric character chosen from a limited set that includes the characters:

```
+ - * / | \ · < > ≤ ≥ ≠ =
```

### 3.2 Selectors.

Every message has a name that is an instance of class *UniqueString*. The name is called a *selector*. It is derived by stripping out the arguments and all delimiters (spaces, tabs, carriage returns, and comments) from the message.

Examples:

```
next + · lookup: insert:with: next← ·← instfield:←
```

Keyword selectors can *not* be written consecutively, or else they would concatenate into a selector with a longer name. Thus,  $(a \text{ min: } b) \text{ max: } c$  and  $a \text{ min: } (b \text{ max: } c)$  are quite different from  $a \text{ min: } b \text{ max: } c$ . The first two expressions would invoke the keyword selectors *min:* and *max:*, while the latter would invoke the single keyword-selector *min: max:*.

### 3.3 Message dictionaries.

When a message is passed to an object, Smalltalk selects a method to execute as follows. It looks in the *message dictionary* of the object's class for the message selector. If the selector is found there, the corresponding method is executed. Otherwise, the superclass of that class is determined, and its message dictionary is probed. This process is repeated as many times as necessary, up to and including class *Object*, the superclass of all classes. If no superclass has a definition for the selector, then the program stops execution and the user is notified that the message was "not understood".

A message to *self* is not exceptional; it simply starts the message dictionary search at the class of *self*. The self-reference *super* means the same as *self* except that it starts the search at the superclass of the class in which the sending method is defined. For example, class *Window* responds to the message *show* by painting the interior of its screen image white and the outline black, and by displaying a title above the top edge. Class *PanedWindow* is a subclass of class *Window* that responds to the message *show* by calling *super show* and then sending a message to each pane telling it to outline its image. Thus, if the variable *p* contains a reference to a *PanedWindow*, *p show* will execute *PanedWindow's show* method, which in turn will execute *Window's show* method. But if *w* contains a reference to an ordinary *Window*, *w show* will only execute *Window's show* method.

### 3.4 Computed selectors.

Occasionally, the message to pass to an object must be computed during execution. For example, it may be desired to evaluate  $a+b$ ,  $a-b$ ,  $a*b$ , or  $a/b$  depending on whether the value of a certain variable *op* is  $\mathcal{E}+$ ,  $\mathcal{E}-$ ,  $\mathcal{E}*$ , or  $\mathcal{E}/$ . To force the value of *op* to be treated as a selector, use the form:

a perform: op with: b

For messages of more than one argument, append additional *with:* phrases. For messages of no arguments, omit the *with:* phrase. For example, it may be desired to evaluate *file creationDate*, *file writeDate*, *file length*, or *file extension* depending on whether the value of *sortKey* is  $\mathcal{E}creationDate$ ,  $\mathcal{E}writeDate$ ,  $\mathcal{E}length$ , or  $\mathcal{E}extension$ . To force the value of *sortKey* to be treated as a selector, use the form:

file perform: sortKey

If the value of *op* in the first example is not a one-argument selector, or if the value of *sortKey* in the second example is not a zero-argument selector, then Smalltalk will report an error to the user.

### 3.5 Precedence.

There are three levels of precedence in the Smalltalk expression syntax: unary selectors bind tightest; binary selectors come next; keyword selectors are weakest. Exception: the  $\leftarrow$  in an assignment message has the weakest right precedence. In case of equal precedence, grouping of unary and binary selectors is left to right. Example:

`u length - v length * w length max: x length + q vec length`

is evaluated as if it had been parenthesized as follows:

`((u length) - (v length)) * (w length) max: ((x length) + ((q vec) length))`

Although precedence rules are well-defined, it is advisable to make no assumptions about the *order of evaluation* of primaries in an expression. It is unlikely that you can guess the behavior of:

`p*(i+i+1) ← q*(i+i+1) max: r*(i+i+1).`

If  $i$  was initially zero, does  $q*1$ ,  $q*2$ , or  $q*3$  get evaluated? It may vary from implementation to implementation. However, one can be certain that the recipient and all arguments are fully evaluated before the recipient's method begins execution, except in the remote evaluation case described below.

### 3.6 Remote evaluation.

When an open colon ( $\textcircled{8}$ ) is employed instead of a closed colon ( $\textcircled{:}$ ) after a keyword, the argument that follows is *not evaluated* before the message is passed. The recipient may evaluate such an argument after receipt, by passing that argument the message *eval*. The evaluation may be done more than once if desired, or not at all; it will always take place in the context of the sender (cf. Algol call-by-name). For example, the expression:

`user time $\textcircled{8}$  [dict lookup: 'twelve']`

tells the Smalltalk user interface to return the time (in 39 millisecond units) that it takes to evaluate the expression in brackets. The method employed to respond to *time $\textcircled{8}$*  is:

`time $\textcircled{8}$  expr | t "the time in 39ms units to evaluate expr"  
[t ← mem*280. expr eval. ↑mem*280-t]`

The first statement in the block assigns to the temporary variable  $t$  the current value of the real-time clock in Alto memory location 280. The second statement evaluates the expression in question; this act is called *remote evaluation*. The third statement returns the difference between the new clock reading and the old.

There is a special case of remotely evaluated arguments in which the argument is a variable. In that case, it is supposed to be possible for the recipient to assign a value to that variable. However, this *remote assignment* facility is not supported by the current compiler except in the *for* statement (see Section 4.4).



## Chapter 4: Statements

### 4.1 Blocks.

Evaluation in Smalltalk is sequenced by use of *statements* and *blocks*. Any expression can serve as a statement; in addition, the language has a number of other statement forms. A block is a sequence of statements separated by periods and enclosed in a pair of square brackets.

A block may serve as a syntactic primary. Its value is determined as follows. If its last statement is not followed by a period, then that statement's value becomes the value of the block. Example:

```
[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, *nil* becomes its value. Example:

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

in which the statements are executed for effect, and the value of the whole block is just *nil*.

If the last thing executed in a block is a *return statement* introduced by a  $\uparrow$  symbol, then the method in which it is embedded terminates and returns the value of the expression that follows the  $\uparrow$ . Example:

```
[v ← dict lookup: 'twelve'. w ← dict lookup: 'six'.  $\uparrow$ v+w]
```

in which 18 is returned from the surrounding method, even if other blocks intervene.

The value returned by a method that does not execute a return statement is always *self*, regardless of the last statement in the outermost block. Section 5.1 discusses methods in more detail.

### 4.2 Cascading.

A number of messages may be sent to the same recipient in sequence without naming the recipient each time. Separate the messages by semicolons. The last object that was sent a message before the first semicolon will be sent all the messages. Example:

```
self pen go: dist+10; turn: 90; penup; go: 20; pendn
```

According to the rules of precedence, the last message sent before the first semicolon was *go: dist+10*. The object to which it was sent was the value of *self pen*. Therefore, that same object is sent the messages *turn: 90*, *penup*, *go: 20*, and *pendn*, in that order. Each message is sent before the arguments of the next message are evaluated.

The value of a cascade statement is the response from its last message. Thus, it is possible (but not very readable) with a single mention of *dict* to both insert an entry into that dictionary and look up an entry in it:

Acceptable:

```
v ← [dict insert: 'twelve' with: 12; lookup: 6]
```

Incorrect:

```
v ← dict insert: 'twelve' with: 12; lookup: 6
```

To the latter form, the compiler will complain "improper cascading."

### 4.3 Conditionals.

The value of an expression or of a cascade may be treated as a true-or-false condition to choose between two alternative paths of execution. The condition is followed by a  $\Rightarrow$  ("then only") symbol and two alternatives. If the condition is *false* then the second alternative is executed. Otherwise, the first alternative is executed. The entire construct, including the condition and both alternatives, is called a *conditional*. Example:

```
x < y  $\Rightarrow$  [3] 4
```

in which the condition is  $x < y$ , the first (true) alternative is [3], and the second (false) alternative is 4.

Syntactically, the first alternative of a conditional is a bracketed block, and the second alternative is (careful!) everything following that up to the end of the block in which the conditional appears. It follows that a conditional must always be the last statement in a block. The value of the executed alternative becomes the value of that block. Example:

```
z ← [x ← dict lookup: 'six'. x < y  $\Rightarrow$  [3] sink next ← x. 4]
```

which first looks up 'six' in *dict*, assigning the result to the variable  $x$ , and which then compares that result with  $y$ . If  $x$  is less than  $y$ , it executes the first alternative, a block with a single statement that simply evaluates 3. Otherwise, it executes the second alternative, consisting of two statements, the first of which is *sink next ← x* and the second of which simply evaluates 4. The value assigned to  $z$  is either 3 or 4.

An alternative is often itself a conditional, for example:

```
highest ← [x > y  $\Rightarrow$  [x > z  $\Rightarrow$  [x] z] y > z  $\Rightarrow$  [y] z]
```

which assigns to *highest* the largest value among  $x$ ,  $y$ , and  $z$ . Both alternatives of the condition  $x > y$  are themselves conditionals. In Algol-60, the statement would have been written:

```
highest ← if x > y then (if x > z then x else z) else if y > z then y else z
```

The first alternative of a conditional may be an empty pair of square brackets. The second alternative may be omitted altogether. The value of the missing alternative in either case is *nil*. Example:

```
[user keyset=4⇒ [] user kbd=015⇒ [text scrollby: 2]]
```

in which nothing happens if the value of *user keyset* is 4; otherwise, a scroll message is sent to *text* if the value of *user kbd* is 015; and otherwise nothing happens at all. In Algol-60, the statement would have been written:

```
if keyset(user)=4 then begin end else if kbd(user)=015 then scrollby(text, 2)
```

A condition may include conjunctions, disjunctions, and negations, all of which are simply messages recognized by class Object. There are two forms of conjunction:

```
p and: q
p ando q
```

The condition is true iff both *p* and *q* are true (not *false*). However, in the first form, both *p* and *q* are evaluated, while in the second form, *q* is not evaluated when *p* is *false*.

There are two forms of disjunction:

```
p or: q
p oro q
```

The condition is true iff either *p* or *q* is true (not *false*). However, in the first form, both *p* and *q* are evaluated, while in the second form, *q* is not evaluated unless *p* is *false*.

There are two forms of negation, both of which employ the binary message, *same as*, symbolized  $\equiv$ :

```
p ≡ false
false ≡ p
```

The condition is true iff *p* is *false*.

The second alternative of a conditional may begin with a cascade. In that case, the recipient of the messages in the cascade is that object which was last sent a message in the condition. Thus, in:

```
ans ← [x+2<v length⇒ [1]; <50⇒ [2]; <90⇒ [3] 4]
```

which could be written vertically as:

```
ans ← [x+2 <v length⇒ [1];
        <50⇒ [2];
        ≤90⇒ [3]
        4]
```

the recipient of all three messages, *<v length*, *<50*, and *≤90*, is the value of *x+2*; the first condition is whether the recipient is *<v length*; if that is true, *ans* becomes 1; otherwise, the second condition is whether the recipient is *<50*; if that is true, *ans* becomes 2; the third condition is whether the recipient is *≤90*; if that is true, *ans* becomes 3; otherwise, *ans* becomes 4. This construct provides a kind of "case statement".

The Smalltalk conditional often causes trouble for beginners with experience in other languages. A common error is to omit the brackets around the block whose value it supplies.

#### 4.4 Loops.

Loop statements and other control statements may be written as messages without explicit recipients. Implicitly, they are sent to an object called *thisContext* (see Section 6.1). The current compiler supports only the control statements described in this section, all of which are loop statements.

The currently supported loop statements each take a remotely evaluated argument (see Section 3.6) after the keyword `do`. The current compiler requires that the argument of `do` be a block.

A loop statement always has the value *nil*, which is generally of no interest. There are three major kinds: *until* statements, *while* statements, and *for* statements.

The *until* statement repeats a block of statements until a condition is true (not *false*). The test of the condition is made before each performance of the block:

```
until user keyset=2 do [text scrollBy: 1]
```

The *while* statement repeats a block of statements until a condition is false. The test is made before each performance of the block:

```
while user keyset=2 do [text scrollBy: 1]
```

Both of the above statements tell *text* to scroll one line at a time as long as the value of *user keyset* is 2.

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 assigned to an *iteration* variable. The iteration variable is a remotely assigned argument after the keyword `for`. This is the only use of remotely evaluated variables supported by the current compiler.

There are two forms of *for* statement. They obtain the set of values for the iteration variable in different ways. One form assigns to the iteration variable the successive integers between 1 and a stop-value. Example:

```
for i to: 10 do [a*i + b*i * 2]
```

in which the first ten elements of the array *a* are assigned double the values of the corresponding elements of *b*. The other form assigns to the iteration variable the successive elements of an array or stream. Example:

```
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 means "modulo"), and the first divisor which works (if any) is returned from the surrounding method without further testing.

The argument of *from:* need not actually be an array or stream, as long as it behaves like one in the following way. It must be able to respond to the message *asStream* by returning an object that responds to the message *next*. Instances of class *Stream*, class *File*, class *HashSet*, every subclass of class *Array*, and several other predefined classes are able to do so.

To generate a progression of numbers other than "1 to n in steps of 1", use the second form of *for* statement, making the array be an instance of class *Interval*. Example:

```
for8 i from: (100 to: 10 by: -5) do8 [myFile next + a*i].
```

in which the elements numbered 100, 95, 90, ..., 10 in the array *a* are written on *myFile*.

Caution: the second form of *for* statement will stop supplying values either when the array or stream is exhausted, or when an element is encountered in the array or stream that has the value *false*, whichever comes first. Therefore, in:

```
for8 num from: inStream do8 [num<100⇒ [outStream next + num]].
```

*num<100* is evaluated a number of times, with *num* assigned each successive item from *inStream*, until the end of the stream is encountered or the first occurrence of *false*, whichever comes first.



## Chapter 5: Definitions

### 5.1 Method Definitions.

In the following method definition of class Dictionary (see Section 1.5):

```
lookup: name | x  "the value corresponding to name, if any, else false"
                [x ← self find: name ⇒ [↑values`x] ↑false]
```

the parts of the definition are known as:

The pattern

```
lookup: name
```

The temporary variables

```
x
```

The prologue

```
"the value corresponding to name, if any, else false"
```

The body

```
[x ← self find: name ⇒ [↑values`x] ↑false]
```

Defining a method makes or updates an entry in the message dictionary of the class, associating the compiled program with the selector derived from the pattern. A pattern looks exactly like a message except that instead of arguments, there are the names of variables to which arguments shall be assigned when a message is received. If a particular argument variable is neither a class variable nor an instance variable of the recipient, then it is automatically declared to be a method variable.

Additional method variables may be declared using the | construct. If there are several such variables, they are separated from each other by delimiters (spaces, carriage returns, tabs, and comments):

```
avgOfNext3 | x y z  "the average of my next three items"
             [x ← self next. y ← self next. z ← self next. ↑(x+y+z)/3]
```

Every method except the most trivial should be accompanied by a prologue, i.e., a comment that explains the role of the method. The ideal prologue does not discuss either the algorithm or the representation used, only what the method does from the viewpoint of the sender.

There are several special forms of method recognized by the current compiler. If the method has no arguments and does nothing except to return either *self* or an instance variable, faster object code is compiled. The program block may be omitted altogether if the method does nothing after the pattern but return *self*. Such a method is useful to implement (non-automatic) *coercions*. For example, several classes respond to *asStream* by creating and returning a Stream, but classes Stream and File simply return *self*.

The body of a predefined method sometimes is followed by a *primitive* clause. An example in class `Integer` is:

```
+ arg "the integer sum of me and arg"
  [↑self+arg asInteger] primitive: 6
```

Smalltalk attempts to execute the method as the 6th standard machine language or microcode subroutine in its repertoire. If the primitive fails (as when the argument of `+` is not an integer), then the regular method body is executed instead. In the example above, the body tells the argument to return an integer equivalent of itself; then it tries again.

As a consequence of the above definition in class `Integer`, it is the case that `5 + 2.3` evaluates to 7, since `2.3 asInteger` is 2. However, `2.3 + 5` evaluates to 7.3, because class `Float` defines:

```
+ arg "the sum of me and arg"
  [↑self+arg asFloat] primitive: 67
```

and `5 asFloat` is 5.0.

It is possible to define such messages so they are more symmetric. For example, one could change the above definition in class `Integer` to:

```
+ arg "the sum of me and arg"
  [↑arg+self] primitive: 6
```

whereupon `5 + 2.3` would invoke `2.3 + 5` and thus evaluate to 7.3.

There are several dozen primitive operations in the system that deal with arithmetic, input-output, streaming, interrupts, and so forth. New primitives can be added only by arrangement with the Smalltalk implementers.

Innumerable examples of method definitions may be found in the Smalltalk predefined classes, which are available both in hard copy form and on-line through the Smalltalk user interface.

## 5.2 Class Definitions.

An example of a class definition is:

```
Class new title: 'BitRectEditor';
    subclassof: Window;
    fields: 'tools picture dirty';
    declare: 'actionPic actionButs picframe toolpic windowmenu tools'
```

in which *BitRectEditor* is the name of the class, *Window* is its superclass, *tools*, *picture*, and *dirty* are its instance variables, and *actionPic*, *actionButs*, and so forth are its class variables. Note the string quotes around each argument except the superclass. The 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 (counting those of its superclass) have changed, then in the current Smalltalk, all old instances will be *obsolete*, i.e., they will fail to respond to messages. Furthermore, all the methods defined for the class may become undefined until they are each updated and recompiled. When class redefinition is attempted, the user is warned of any such impending trauma and given a chance to withdraw the redefinition.

If a class is to share pool variables with other classes, an additional clause (preceded by a semicolon) should be added to the class definition for each such pool. Example:

```
sharing: Transcendentals
```

where *Transcendentals* is the pool. A pool is simply a symbol table that associates variable names (of class *UniqueString*) with variables (of class *ObjectReference*).

To initialize class or pool variables, define a message with the special name, *classInit*. For example, class *BitRectEditor* has a message called *classInit* that assigns values to its class variables. The statement *BitRectEditor classInit* will automatically invoke that message and thus initialize *actionPic*, *actionButs*, etc.

Classes like *String* and *Vector* that have numbered elements instead of named fields belong to a subclass of class *Class* called *VariableLengthClass*. To create such a class for the first time requires a slightly different form of definition than the usual (although redefining is done in the usual way):

```
VariableLengthClass new title: 'String';
    subclassof: Array;
    bytesize: 8
```

where 8 is the size in bits of each element. If the *bytesize:* clause is omitted, the elements are references to Smalltalk objects (as in class *Vector*); if it is included, then the elements are restricted to be integers (as in class *String*). Currently, the only supported byte sizes are 8 and 16.

A large number of sample class definitions may be found among the Smalltalk language versions of the predefined classes.

### 5.3 Instance creation and initialization.

A new object is created by passing its class the unary message *new*. The class allocates storage for the new object, initializes the object's instance variables to the constant *nil*, and returns the new object. Usually, the new object is then passed additional messages to initialize it fully. For example, a dictionary may be created by the statement:

```
dict ← Dictionary new init: 16
```

Instances of variable-length classes are created differently, using the keyword message *new: length*. The elements of vector-like objects are initialized to *nil* and those of string-like objects are initialized to all one-bits. An example is in class Dictionary's method for *init:*, as follows:

```
init: size
    [values ← Vector new: size. super init: size]
```

This method initializes the *values* field of the new instance itself, and calls upon its superclass (HashSet) to initialize the *objects* field.

If an instance can be initialized by the unary message *init* or by the unary message *default*, then the word *new* may be omitted when it is created and initialized. For example, to create an output Stream of characters, simply use:

```
out ← Stream default
```

This is exactly equivalent to *Stream new default*. The reason that *classInit*, *init*, and *default* are special is that class Class responds to the message *default* as follows (and similarly to *classInit* and *init*):

```
default "a new instance of me, initialized by my default method"
    [↑self new default]
```

No facilities are provided in Smalltalk for explicit deallocation. An object is destroyed automatically when no reference to it exists. Thus, the programmer is generally freed from concern with deallocation. However, if an instance points at itself, or if it is part of a ring, then Smalltalk will never realize that it can be deallocated. Therefore, if a data structure includes cycles or back pointers, then when it is no longer needed, be sure to remove cyclic pointers by changing them to *nil*.

## Chapter 6: Contexts

This chapter is written for experienced systems programmers.

### 6.1 Contexts.

An instance of class `Context` is created for every execution of a method body. A context is not created when the method is executed as a primitive or when it does nothing but return either *self* or an instance variable.

The fields of a context are:

<code>sender</code>	the context from which the message was sent;
<code>receiver</code>	the object to which the message was sent (i.e., <i>self</i> );
<code>mclass</code>	the class in whose message dictionary the method was found;
<code>method</code>	the object code (a string interpreted by microcode);
<code>tempframe</code>	a vector to hold method variables and a stack;
<code>pc</code>	the zero-origin subscript in <i>method</i> of the next instruction byte to be executed;
<code>stackptr</code>	the zero-origin subscript in <i>tempframe</i> of the top of the stack.

It is possible to send a message to a context. The remote evaluation message, *eval*, invokes a method of class `Context`, because a remotely evaluated argument is simply a context that shares the *tempframe* of the sender.

To send a message to the context currently executing, address it to *thisContext*. Note that when a message is sent to a context, another context is created in which to execute the method; the *receiver* of that context is the original context.

During debugging of a program, it is useful to be able to examine the variables of a method in progress, as well as the variables of its sender, its sender's sender, and so on down the "stack". The debugging facilities of the Smalltalk user interface send messages to contexts in order to display the stack to the user.

Although most programs never mention class `Context` or its instances explicitly, it is indeed a very important class.

### 6.2 Processes.

It is possible to make a method return to other than the sender of the message by changing the *sender* field of *thisContext* before returning. It is also possible to assign the value of *thisContext* to another variable. From these crude facilities can be derived a way to suspend execution of a context and then to resume it later. From that capability can be built control structures involving processes, coroutines, and so forth. For examples, see the predefined class `PriorityScheduler`.



## Appendix I: Predefined classes

### Kernel classes

#### Class

Each Smalltalk class is itself an instance of the class, Class. An instance of class Class (i.e., a class) can:

- be told its fields, i.e., the representation to use for its instances;
- create a new instance of itself;
- create a copy of an old instance;
- be told a new method of responding to a message;
- be told to forget a method;
- produce a standard printed representation of an instance;
- produce a standard printed representation of its methods.

#### VariableLengthClass

Classes like Vector and String are unusual in that their instances have numbered elements instead of named fields. Such classes are themselves instances of class VariableLengthClass, a subclass of class Class. An instance of class VariableLengthClass (e.g., class Vector or class String) can:

- create a new instance of itself with a specified number of elements;
- do anything defined by its superclass, Class.

#### ClassOrganizer

As an aid in dealing with the large number of classes in Smalltalk and the variety of messages which they recognize, a facility is provided for *organizing* them into *categories*. A class organizer also contains a comment describing the class and the roles of its fields. It can:

- classify a name under a specified category heading;
- return a vector of all the names under a specified heading;
- look up a name and return its category;
- produce a printable editable representation of the categorization;
- redefine the organization by reading back its edited representation.

### Context

To respond to a message, an object executes a method. To do so, it needs to have space in which to work, needs to keep track of its progress through the method, and needs to remember who sent the message so it can reply when it is done. All these needs are serviced by an instance of class Context, one of which is created for every execution of a method body. It also provides help in debugging programs, for it can:

- determine the context that sent it the message;
- single step through the method;
- report the current values of variables employed by the method.

Processes, coroutines, and other control structures can be constructed by manipulating instances of class Context.

### Object

Every class is ultimately a subclass of class Object. Thus, every instance of every Class can inherit messages from class Object. Every object in the system can do all the following things, unless its own class has overridden class Object's methods:

- determine whether a given value is the same as itself;
- tell what class it is in;
- produce a printed representation of itself;
- produce a copy of itself;
- create a window through which the user can inspect its state.

### UserView

There is only one instance of class UserView, accessed through the global variable, *user*. That instance can:

- give control of the user interface to one window at a time;
- read the keyboard, mouse, and keyset;
- converse with the user through a simple dialog window;
- access on-line system documentation;
- report errors, warning, and breakpoints to the user;
- measure the time it takes to execute a program;
- change the proportions of the usable display area;
- exit the Smalltalk system and overlay a different system.

## Numbers

### Number

Class Number is a superclass of Integer, Float, and LongInteger. An instance of any of those classes can:

- determine the larger or smaller of itself and another number;
- tell its sign (-1, 0, or +1);
- create an instance of class Point, with itself as x and either itself or another number as y;
- create an instance of class Interval from itself to some other number.

### Integer

An instance of class Integer is a whole number in the range -32768 to +32767. It can:

- compare its magnitude with another whole number;
- do arithmetic on itself and another whole number;
- do 16-bit logical operations and shifts;
- return a Float or LongInteger instance of its own magnitude;
- print itself in decimal, octal, or binary radix;
- treat itself as a character code and convert between upper and lower case, tell whether it is a letter or a digit, and so forth;
- do anything defined by its superclass, Number.

### Float

An instance of class Float is an exponent/mantissa representation of a rational number. Its precision is about 9 decimal places, and its exponent range is  $\pm 16000$ . It can:

- compare its magnitude with another Float or Integer instance;
- do arithmetic on itself and another Float or Integer instance;
- return its integer part or its fractional part;
- return itself "as an Integer", if in the range -32768 to +32767;
- compute its own sine, cosine, or square root;
- treat itself as an angle in degrees and return that angle in radians;
- print itself using decimal notation in the form  $\pm 99999.9999$ , or if more places are required, using scientific notation, e.g., 6.23e24;
- do anything defined by its superclass, Number.

### LongInteger

An instance of class LongInteger is a whole number of any reasonable magnitude. It can:

- compare its magnitude with another LongInteger or Integer instance;
- do arithmetic on itself and another LongInteger or Integer instance (exception: division is not implemented as yet);
- print itself;
- do anything defined by its superclass, Number.

### Date

A instance of class Date represents a specified day of a specified year. It can:

- compute a date a specified number of days earlier or later;
- produce a printed representation of itself in various formats;
- read back that printed representation.

### Time

An instance of class Time represents a specified second of an unspecified day. It can:

- produce a printed representation of itself in various formats.

## Basic data structures

### Array

Class Array is a superclass of Vector, Interval, String, Substring, and UniqueString. An instance of any of those classes can:

- take an array of subscripts and either return an array of, or reassign, the specified elements;
- tell how many elements it contains (its "length");
- compare itself element by element with another array;
- return the concatenation of itself and another array;
- copy itself exactly, with an insertion, with a deletion, with a replacement, or with unoccupied room on its end;
- return or reassign its last element;
- assign the same value to all its elements;
- interchange two of its elements;
- find the first element equal to or unequal to a given value;
- sort itself, or return a permutation that would sort itself;
- treat itself as a priority list and promote an element to the front;
- return itself in reverse order;
- return a Stream that can scan its successive elements.

### Vector

A vector is an array of references to objects. It can:

- return or reassign its nth element;
- do anything defined by its superclass, Array.

### Interval

An interval is an arithmetic progression defined by a start-number, a step-number, and a stop-number. It can:

- return its nth element;
- do anything defined by its superclass, Array, that does not alter its state.

**String**

A string is a sequence of zero or more eight-bit bytes, i.e., integers between 0 and 255. The bytes are often treated as ascii characters. It can:

- return or reassign its nth byte;
- return or reassign its nth two-byte "word";
- compare itself with another string, ignoring case discrepancies;
- return an upper case copy of itself;
- return a long integer whose digits are its characters;
- return a tokenized parse of itself;
- return a hash of itself;
- treat itself as an Alto file name matcher (with \* and # characters) and match itself against a string treated as a file name;
- replace all occurrences in itself of a specified character sequence by another specified character sequence;
- look itself up in the global set of unique strings and return the unique (possibly new) instance of class UniqueString that equals itself;
- do anything defined by its superclass, Array.

**Substring**

A substring is a reference to a list (a "map") of positions in some ("mapped") array (often, but *not necessarily*, a string). It can:

- take a subscript, subscript the map to determine a position in the mapped array, then return or reassign that position in the mapped array;
- do anything to the mapped array that is defined by its superclass, Array, except that interchanging and sorting only affect the map, not the mapped array;
- do anything defined by its superclass, Array.

**UniqueString**

Class UniqueString is a subclass of class String. A unique string is an entry in a certain global set ("UStable") in which no two entries are equal to each other. It can:

- help a string to perform its lookup in UStable;
- return a hash different from a string hash, and faster;
- do anything defined by its superclass, String.

**Stream**

A stream is a reference to some position in some array. It can:

- advance or retreat through the array, getting, putting, or skipping the elements passed;
- copy the array to a larger array if it passes the end while putting;
- treat itself as a character output stream and append a character, a standard delimiter, a string, or the printed representation of an arbitrary object;
- treat itself as a file buffer and tell the file when buffer reloading is required.

## Sets and Dictionaries

### HashSet

A hash set is a set of objects none of which appears more than once in the set. All the objects usually belong to a single class. The present implementation allows mixed classes only if every one of the classes can respond *false* to *=arg* where *arg* is an object in any of the other classes; future implementations will lift this restriction. A hash set can:

- tell whether a given object is present;
- insert or delete an object.

### Dictionary

Class Dictionary is a subclass of class HashSet. A dictionary associates a value with each object in it. It can:

- tell what value is associated with a given object;
- insert, delete, and change associations;
- do anything defined by its superclass, HashSet.

### MessageDict

Class MessageDict is a subclass of class HashSet. Every class has a message dictionary in which to hold the messages it can understand and the methods it uses to respond to those messages. The objects of a message dictionary are the names ("selectors") of the messages. With each message are associated the source code and object code of its method. A message dictionary can:

- look up a selector and return the corresponding source code or object code;
- associate a new method with a selector;
- do anything defined by its superclass, HashSet.

### ObjectReference

An object reference contains only one field, the *value* field. It is useful as an indirect reference to an object. An instance of class ObjectReference can have its contents examined or reassigned much as if it were a variable, except that a message must be sent to it to make it do so. An object reference can:

- make itself refer to a given object;
- return (a reference to) the object it references.

### SymbolTable

Class SymbolTable is a subclass of class Dictionary. The values it associates with its objects are instances of class ObjectReference. In effect, it associates a "variable name" with a "variable". In fact, a pool is implemented as a symbol table, and a pool variable as an object reference. However, method and instance variables are not implemented as instances of class ObjectReference. A symbol table can:

- declare a pool variable by inserting an association between the variable's name and a new object reference that will be the variable itself;
- find the variable of a specified name and read or write its value;
- do anything defined by its superclass, Dictionary.

## Graphical objects

### Point

A point is an x-y pair of numbers. It often (but not necessarily) represents a location on the display screen. A point can:

- be compared with another point;
- compute a new point by adding, subtracting, multiplying, or dividing its coordinates either by a number or by the corresponding coordinates of another point;
- compute a new point near itself but on a specified grid.

### Rectangle

A rectangle is a pair of points that often (but not necessarily) represent a rectangle on the display screen whose upper left corner ("origin") and lower right corner ("corner") are the two points. A rectangle can:

- compute a new rectangle by adding, subtracting, multiplying, or dividing its coordinates either by a number or by the corresponding coordinates of a point or of another rectangle;
- operate on the bitmap image displayed in the corresponding area of the display screen in any of the following ways:
  - fill the area with any *tone* that can be built from tiles all of which are the same 4x4 dot matrix specified by a 16-bit integer;
  - copy it to another area of the same size on the screen or in a hidden buffer (a string);
  - copy a hidden buffer of the same size into it;
  - in another area of the same size on the screen "brush" a dot pattern which has a black dot in those places in which both this rectangle *and* a specified tone have a black dot;
  - do any of the above operations in one of four modes: storing (opaque), oring (transparent), xoring (complement), or erasing;
- be told to change its coordinates;
- compute the intersection of itself with another rectangle, or the nonintersection, or the union.

### Turtle

A turtle is a drawing pen filled with black, white, or complementing ink, with a penpoint from 1 to 8 dots thick, at a particular location on the screen and facing at a given angle from the vertical. It can:

- crawl in a straight line to another specified screen location, drawing or not as it goes;
- print a string in any font starting at its current position on the screen and along the line it is facing;
- change its angle, ink, or penpoint width.

### Menu

A menu is a column of brief one-line text entries that can be displayed in a temporary frame on the screen. It allows the user to:

- display the menu on the screen under the cursor;
- select one or none of the entries with the cursor;
- let up the mouse button to remove the menu display, to restore the screen image that was there before, and to invoke the action selected (if any).

### Cursor

A cursor is a 16 by 16 matrix of black and white dots suitable for display as the Alto screen cursor. It can:

- install itself as the current screen cursor during the execution of a given program;
- remember the previous cursor;
- restore the previous cursor when the given program is done.

### BitRect

A bit rectangle is a rectangular matrix of black and white dots that can be displayed on the screen and in which the user can paint ("edit"). It can:

- show itself on the screen;
- update itself from its current image on the screen;
- do anything defined by its superclass, Rectangle.

### BitRectEditor

A bit rectangle editor is a window in which are displayed a painting and a menu of painting tools (a "toolbox"). When the user wakes up the window, the mouse can be used to select tools and to paint. A bit rectangle editor allows the user to:

- select one of six tools (pen, eraser, straightedge, block-of-gray-maker, paint brush, magnifier);
- use the selected tool;
- change properties of the selected tool (see class BitRectTool);
- do anything defined by its superclass, Window.

Caution: as of the release of this manual, there are a few bugs in the toolbox program which most often strike people who have not read its documentation. Full documentation is available on-line in the class organizations of this class and of the two classes below.

### BitRectTool

A bit rectangle tool is a tool that can change the contents of a bit rectangle. Its behavior is determined by a combination of properties: a grid, a tone of gray, a penpoint width, a mode, and an action. A bit rectangle tool can have each of its properties changed independently.

### RadioButtons

An instance of class RadioButtons is a row or column of square regions ("buttons") on the display screen, each of which has an undisplayed value. There is always exactly one button "pushed". The screen regions corresponding to all the unpushed buttons are grayed. RadioButtons are used in the BitRectEditor menu. An instance of RadioButtons can:

- be told to "push" the button whose region includes a given screen point (such as the cursor location);
- return the value associated with the currently pushed button;
- be asked whether a given point is within any button's region.

## Text objects

Note: Significant changes to this category are imminent.

### Paragraph

A paragraph is an array of characters each of which is assigned a typeface. The paragraph as a whole has a specified *alignment*: flush left, flush right, centered, or justified. A paragraph can:

- have any sequence of characters inserted, deleted, or replaced;
- have any sequence of characters change their typeface;
- have its alignment changed;
- be converted to or from Bravo representation;
- have anything done to its characters (ignoring typeface) defined by its superclass, Array.

### TextStyle

A text style specifies typesetting information that can be applied to a paragraph to display it on the screen. A text style can:

- associate with any font index a real dot-matrix font (there is no class Font, so the "strike" format font is stored in a string);
- be told a logical operation ("mode") by which the dots of characters should be put into the screen bit map;
- be told the size of a space character;
- be told the distance between tab stops;
- be told the height of displayed lines in two parts, the ascent and descent relative to the baseline (the letter "b" is said to ascend above the baseline, while the letter "p" both ascends and descends).

### Textframe

A text frame allows paragraphs to be displayed on the screen according to a specified text style, typeset to a specified width, and clipped on a specified boundary. It can:

- display a paragraph according to those constraints;
- report the screen position of any character in a paragraph that has been or could be displayed;
- report the index in the paragraph of the character displayed nearest to a specified screen position.

**Dispframe**

A display frame is a window on the display screen through which a teletype-style dialogue can be carried on. There is typically a single display frame, managed by *user* (the only instance of class *UserView*). It can:

- prompt for a user input;
- read a user input into a tokenized form;
- evaluate the tokenized input as a Smalltalk program and print the last value computed by that program;
- do anything defined by its superclass, *Stream*;
- show the recent dialogue, including anything appended by *Stream* operations.

**ParagraphEditor**

A paragraph editor is an intermediary between a paragraph and a window. It can:

- let the user select an inter-character position in the paragraph, or a sequence of characters;
- cut out the selected characters and put them in a global *Scrap*;
- replace the selection by the characters in the *Scrap*;
- let the user replace the selection by newly typed characters, or backspace over the characters preceding the selection;
- let the user scroll the image of the paragraph through a rectangular area on the display;
- display the paragraph in that area;
- do anything defined by its superclass, *Textframe*.

**FontWindow**

A font window is a window on the display screen in which one character at a time out of a font (typeface) can be edited. The character is displayed blown up so that each dot in its dot-matrix representation is easily distinguishable. The user can:

- point the cursor at any dot; set it black with red bug, white with yellow bug;
- depress blue bug to obtain a menu which allows a new width to be specified for the character, and which allows the whole window to be moved ("framed") elsewhere on the screen;
- type a character to be edited.

**a note about fonts**

There is no *Font* class in the current implementation of Smalltalk. A display font is encoded in "strike format", a standard Alto format documented elsewhere. As an expedient, the font is stored as a series of bytes in an instance of class *String*. Printing a font string will yield nonsense characters.

## Browser and Debugger

Note: Extensive changes to this category are imminent.

### Window

A window is a rectangular area of the display screen with which is associated a user interface to a system facility. Each such facility defines a subclass of class Window. The messages of a window perform standard operations such as watching user input devices and sending itself messages corresponding to the events on those devices; it is expected that the subclass will intercept most of these messages and behave appropriately. A subclass of class Window should respond to the following messages:

<i>enter</i>	the user has just awakened this window;
<i>leave</i>	the user has just put this window to sleep;
<i>close</i>	the user has just closed this window;
<i>redbug</i>	the user has pressed the red mouse button;
<i>yellowbug</i>	the user has pressed the yellow mouse button;
<i>kbd</i>	the user has just struck a key on the keyboard;
<i>keyset</i>	the user is holding down keys on the keyset.

The messages of class Window have trivial default responses to all these messages. They also have non-trivial default responses to such conditions as *bluebug* (which presents a menu for reframing the window, closing it, or yielding control to a window it obscures); the subclasses may override these as well.

### ScrollBar

A scroll bar is a gray zone at the left edge of an awake window. When the cursor moves from the window into its scroll bar, the cursor shape changes to an up or down arrow. Redbug can then be used to scroll the contents of the window in the indicated direction. At each increment of scrolling, the text beside the cursor jumps to the middle of the window; therefore, the rate of scrolling is slowest when the cursor is near the middle.

### PanedWindow

A paned window is a window which has subwindows ("panes") that are awakened and resized in unison. It is common for the contents of some panes to be dependent on selections made in others. Two of the standard paned windows in the Smalltalk user interface are *browse windows* and *notify windows*.

### NotifyWindow

This class is a subclass of PanedWindow. A notify window is created whenever an error, warning message, or breakpoint is encountered in the execution of a program. It initially has one pane that shows the class and message of the currently executing method and which can display the "stack", i.e., the method that invoked the current one, the one that invoked that one, and so forth. It can be grown to six panes that display extensive information about the context of the event. It is possible to proceed from the point of the event (as from a breakpoint) or to close the window and start over.

### ListPane

A list pane is a pane in which is displayed a vertical list of one-line items. The list can be scrolled slow or fast, and any item can be selected. When an item is selected (or deselected), a *dependent* pane can be told to display appropriate material.

### CodePane

A code editor is a pane in which is displayed the source code for a single Smalltalk method. It can be edited (see class ParagraphEditor) and compiled.

### SystemPane, ClassPane, OrganizationPane, SelectorPane, CodePane

Each browse window has one instance of each of these classes. The first four are subclasses of ListPane and the last is described above.

### StackPane, VariablePane, CodePane

Each notify window has one instance of class StackPane, two of class VariablePane, and three of class CodePane. The first two are subclasses of ListPane and the last is described above.

## Files and Compiler

Note: Extensive changes to this category are imminent.

### File

A file is an object that accesses an Alto disk file on one disk or the other. It can:

- read or write a sequence of disk pages;
- extend or shorten the disk file;
- read the whole (reasonably sized) disk file into a string;
- read and compile Smalltalk programs from the disk file;
- write a printed representation of Smalltalk definitions onto the disk file;
- read or write a byte or word;
- skip a specified number of bytes or pages in either direction;
- do anything defined by its superclass, Stream.

### Directory

A directory is an object that accesses an Alto disk file directory (e.g., 'sysdir' on dp0). It can:

- allocate and deallocate disk pages;
- insert, delete, or rename file entries;
- look up a name and return information about the corresponding file;
- find all file names that match a pattern.

### Compiler

Class Compiler translates Smalltalk programs from source code to object code. It is invoked by the user interface commands `compile` and `doit`. The current compiler is not reentrant.

### Reader

A reader scans a character sequence and builds a vector of tokens such as numbers and strings. It can:

- turn an identifier or a one-character token into a unique string;
- turn a digit sequence with sign, decimal point, and exponent into a number;
- turn a parenthesized substring into a vector;
- turn a single-quoted substring into a string;
- skip comments, Bravo trailers, and separators.

## Primitive access

These classes allow access by knowledgeable programmers to the innards of the system.

### BitBlit

A bit block transferrer ("bit-blitter") allows access to the microcode that moves sub-bitmaps from one part of memory to another. User programs generally access that microcode not through this class but through class `Textframe` or class `Rectangle`.

### CoreLocs

An instance of class `CoreLocs` is an array of integers whose elements are the contents of successive locations in main memory. The most often used instance of this class is the value of the variable `mem`, which maps the entire Alto memory.

### FieldReference

An instance of class `FieldReference` allows access to a specified field of a specified instance. It will be used by future versions of the compiler to pass a method or instance variable by reference for remote assignment.

### PriorityInterrupt

A priority interrupt is an association between a priority scheduler and a priority level number. It can relay a message to its scheduler, after attaching the priority level number to the message.

### PriorityScheduler

A priority scheduler is a collection of contexts, some suspended and some not yet started, plus a collection of priority level numbers classified as to their status (awake, enabled, interrupted, or current). It can:

- execute a "critical" section of code with interrupts disabled;
- change the status of a priority level;
- switch control to the highest priority enabled context;
- become the "top" scheduler, which receives hardware interrupts.

## Appendix II: Syntax Summary

Chapters 2 through 5 describe each language construct in detail. Here, the complete syntax is presented in a formal notation derived from a suggestion by Wirth:

" "   surround a literal  
|       delimits alternatives  
{ }    mean 0 or more of the constituents inside  
< >   mean 1 or more of the constituents inside  
[ ]    mean 0 or 1 of the constituents inside  
~       means any character *except* those that follow  
...     stands for all characters between what precedes and what follows  
( )     group things together

The delimiters of Smalltalk include spaces, tabs, ends of lines, ends of pages, and comments.

comment       = `" " { ~ " " } " "`

Within the following constructs, delimiters may not occur.

digit         = `"0" | ... | "9"`

nonZeroDigit = `"1" | ... | "9"`

integerLiteral = `[ "-" ] nonZeroDigit { digit }`

floatLiteral   = `integerLiteral "." <digit> [ "e" integerLiteral ]`

octalLiteral   = `[ "-" ] "0" { "0" | ... | "7" }`

basicLiteral   = `integerLiteral | floatLiteral | octalLiteral | stringLiteral`

letter         = `"A" | ... | "Z" | "a" | ... | "z"`

constant       = `"nil" | "false" | "true"`

selfReference   = `"self" | "super" | "thisContext"`

variable       = `letter { letter | digit }`

unarySelector   = `letter { letter | digit }`

binarySelector  = `~(letter | digit | ":" | "@" | "(" | ")" | "[" | "]"  
| "<" | ">" | "-" | "+" | "&" | "&" | "&" | ";")`

keyword        = `letter { letter | digit } ( ":" | "@" )`

selector       = `unarySelector | binarySelector | <keyword>`

atomLiteral    = `(letter | ":" | "@") { letter | digit | ":" | "@" }  
| ~(letter | digit | ":" | "@" | "(" | ")")`

Within the following construct, all characters are significant, including delimiters.

stringLiteral = "" {~"" | "" ""} ""

Within the following constructs, delimiters may occur between constituents. The delimiters are ignored.

vectorLiteral = "(" {basicLiteral | vectorLiteral | atomLiteral} ")"

literal = basicLiteral | "☞" (vectorLiteral | atomLiteral)

primary = literal | constant | variable | selfReference | subExpression | block

subExpression = "(" expression ")"

expression = variable assignment | primary [messageChain]

assignment = "←" expression

messageChain = {unaryMessage} {binaryMessage} [keywordMessage]

keywordMessage = <keyword term> [assignment]

term = factor {binaryMessage}

binaryMessage = binarySelector factor [assignment]

factor = primary {unaryMessage}

unaryMessage = unarySelector [assignment]

block = "[" statements "]"

statements = {statement "."} [statement | returnStmt]

returnStmt = "↑" expression ["."]

statement = expression [cascade] | loopStmt

cascade = {";" messageChain} [alternatives]

alternatives = "⇒" block [cascade | statements]

loopStmt = keywordMessage

method = pattern [temporaries] [block [primitive]]

pattern = unaryPattern | binaryPattern | keywordPattern

keywordPattern = <keyword variable> [assignPattern]

binaryPattern = binarySelector [assignPattern]

unaryPattern = unarySelector [assignPattern]

assignPattern = "←" variable

temporaries = "]" {variable}

primitive = "primitive:" integerLiteral

## Appendix III: Character set

The current implementation of Smalltalk has a non-standard character set. Future versions will conform more closely to Ascii. The current set can be typed in either Bravo or the Smalltalk user interface. A couple of characters are currently printed differently on the display than on paper: Ⓢ is displayed as a rough circle with a central dot, and @ is displayed as ≡.

Bravo Stroke	Character	Smalltalk Stroke	Meaning
ats	≠	↑<	Relational
nts	#	↑=	
rts	≡	↑>	
fts	Ⓢ	↑f	Is same object as
cts	Ⓞ	↑:	Remote evaluation
? ! "	Ⓡ Ⓡ Ⓡ	? ! "	Then only Return Literal
uts	Ⓡ	↑-	Negative literal
ots	Ⓡ	↑"	Comment
gts	Ⓡ	↑g	Subscript
%	Ⓡ	%	Peek
sts	Ⓡ	↑s	Compile
@	Ⓢ	@	Point
↑ts	Ⓢ	LF	Do It
		↑w	Backspace word
		DEL	Cut
		↑b	Toggle bold face
		↑i	Toggle italic face
		↑~	Toggle underline
		↑d	Proceed
		↑c	Interrupt
		↑shift-ESC	Restart
\$	Ⓢ	\$	Misc. symbols
ets	Ⓢ	↑e	
kts	Ⓢ	↑k	
pts	Ⓢ	↑(	
tts	Ⓢ	↑t	
yts	Ⓢ	↑y	
]ts	Ⓢ	↑]	
qts	!	↑l	English punctuation
vts	%	↑5	
[ts	\$	↑4	
~	?	~	