

Digitalk License Statement

This book and the accompanying software are copyrighted and are therefore protected by the Copyright Laws of the United States and copyright provisions of various international treaties. The effect of such laws and treaties is that you may not, without a license from DIGITALK, copy or distribute the book or software. The software and book may be used by any number of people and moved to different locations provided there is no possibility of either of them being used simultaneously at two or more locations or being used by two or more people at the same time.

DIGITALK hereby grants to you the right to make archival copies of the enclosed software solely for the purpose of protecting yourself from loss or damage of such enclosed software.

Warranty

DIGITALK warrants the enclosed diskettes and documentation to be free from defects in materials and workmanship for a period of 60 days from the date of purchase. DIGITALK will replace any defective diskette or documentation returned to DIGITALK during such warranty period. Replacement is the exclusive remedy for any such defects, and DIGITALK shall have no liability for any other damage.

IN NO EVENT SHALL DIGITALK, INC., BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGE, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL OR OTHER DAMAGES. DIGITALK, INC., SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, RELATED TO DEFECTS IN THE DISKETTE AND DOCUMENTATION.

Governing Law

This statement shall be governed and construed under the laws of the state of California and subject to the exclusive jurisdiction of the courts therein.

Smalltalk/V 286

Tutorial and Programming Handbook

digitalk inc.

The programming language Smalltalk and many of the concepts of modern user interfaces were developed in research projects at Xerox Palo Alto Research Center (PARC) over a period of several years, and culminated in Smalltalk-80. We should like to express our appreciation to the researchers in the Learning Research Group under Alan Kay and the System Concepts Laboratory under Adele Goldberg. We recognize the debt that Smalltalk/V owes to their creative efforts.

Copyright 1988 by Digitalk Inc., all rights reserved
First printing May 1988

Copying or duplicating this manual or any part thereof is a violation of the law. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including but not limited to photocopying, without written permission from Digitalk Inc.

Digitalk Inc.
9841 Airport Boulevard
Los Angeles, California 90045

IBM is a trademark of International Business Machines Corporation; Unix is a trademark of AT&T; Smalltalk-80 is a trademark of Xerox Corporation.

TABLE OF CONTENTS

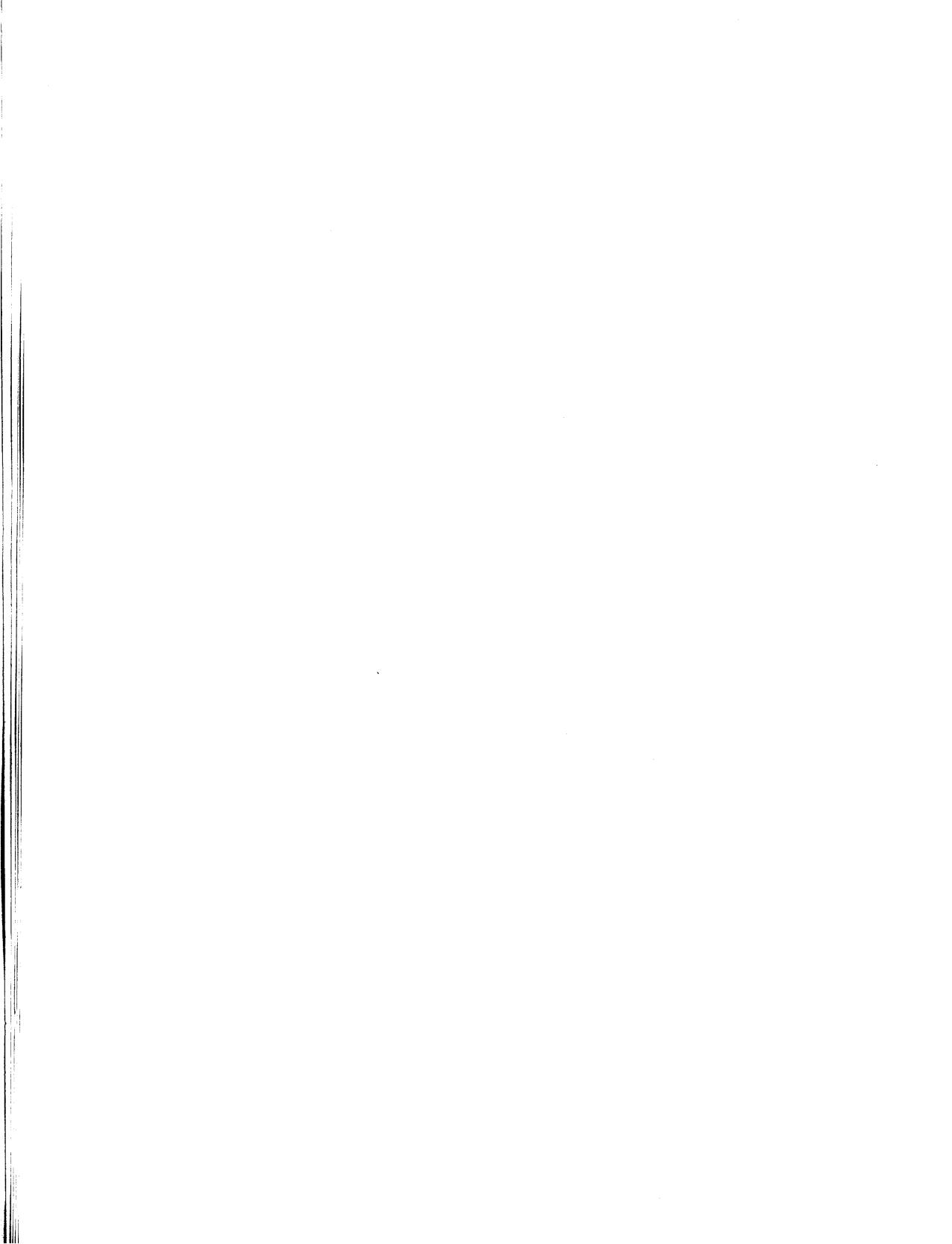
INTRODUCTION	1
System Requirements	2
Before Starting	2
PART 1: OVERVIEW	5
Chapter 1 INTRODUCTION TO THE SMALLTALK LANGUAGE	5
Smalltalk's Big Ideas	5
Smalltalk vs Conventional Languages.....	6
The World According to Objects.....	11
Ideas Into Action.....	18
PART 2: SMALLTALK/V TUTORIALS	21
Chapter 2 INTRODUCTION TO THE SMALLTALK/V ENVIRONMENT	21
Installing Smalltalk/V	21
Starting Up Smalltalk/V	23
Exiting Smalltalk/V	24
Getting Around	25
Working with Your Mouse.....	25
Windows and Menus	26
Windows.....	26
Menus.....	28
Working with Windows	32
Quick Tour—Windows and Menus	33
Inside the Window Pane	37
Starting Out	40
Tutorial Files	44
Chapter 3 OBJECTS AND MESSAGES	45
Simple Objects	45
Simple Messages	46
Unary Messages	47

Keyword Messages	47
Arithmetic Messages	48
Binary Messages	48
Messages Inside of Messages	49
Expression Series	49
Cascaded Messages	50
Simple Loops	50
Objects and Messages Are Safe	51
Temporary Variables	51
Assignment Expressions	52
Return Expressions	52
Global Variables	52
Putting It All Together	53
 Chapter 4 CONTROL STRUCTURES	 57
Comparing Objects	57
Testing Objects	57
Conditional Execution	58
Boolean Expressions	59
Looping Messages	60
Simple Iterators	61
Block Arguments	61
Generalized Iterators	62
Concluding Example	64
 Chapter 5 CLASSES AND METHODS	 67
Classes	67
Methods	68
The Class Hierarchy Browser	69
The Special Variable "self"	70
Creating New Objects and the Special Object "nil"	70
Instance Variables	71
Recursion	71
Pattern Matching	72
Adding a Method to a Graphics Program	73
Class Variables	75
Inspectors	75
 Chapter 6 INHERITANCE	 79
The Class Hierarchy	79
Inheritance	80
Inheritance of Instance Variables	81

The Methods of the Animal Classes	82
Inheritance of Methods	83
The Special Variable "super"	84
Creating Animal Objects	84
Polymorphism	85
More General Pattern Matching	86
Processing Recursive Data Structures	87
A New Class: MonitoredArray	88
Class Methods	89
Chapter 7 STREAMS AND COLLECTIONS	91
Streams	91
Printer Stream	92
Collections	94
Generic Code	96
Blocks as Objects	97
Patterns	97
Computing Letter Pair Frequencies	98
Animals Revisited	99
A Network of Nodes.....	102
Chapter 8 DEBUGGING	107
A Document Retrieval System	107
How Class WordIndex Works	109
Debugging Class WordIndex.....	110
Hop, Skip and Jump.....	115
Chapter 9 GRAPHICS	117
Some Basic Concepts	117
The Basic Class of Graphics: BitBlt	121
Extension of BitBlt	130
Chapter 10 WINDOWS	139
The Prompter	139
Single Pane Window	140
Single Pane Window with More Interaction	141
Multi-Pane Windows	145
Chapter 11 OBJECT-ORIENTED DEVELOPMENT	155
The Smalltalk/V Application Development Cycle.....	155
Knowing When to Stop.....	160

Chapter 12 APPLICATION DEVELOPMENT: CASE STUDY	161
The Case Study: A State-Transition Perspective	161
The Case Study Problem as a Smalltalk/V Problem.....	162
A Window Model for the SalesCom Application.....	163
Menus Enrich the Window Model.....	164
Getting There in Half the Time: <i>Recycling Code</i>	165
Re-working the Network of Nodes	166
Raiding the Animal Habitat	171
Customers and Events: <i>A Matter of State</i>	176
Methods and Messages: Bringing the Prototype to Life	178
It's Getting Better All the Time: Evolutionary Development	181
Where to Go from Here	182
 PART 3: SMALLTALK/V 286 REFERENCE	 187
Chapter 13 THE SMALLTALK LANGUAGE	187
Objects	187
Classes	189
Messages and Methods	194
 Chapter 14 SMALLTALK/V 286 CLASSES	 203
Magnitudes	203
Streams	211
Interface to DOS File System	216
Terminal Input and Output	219
Collections	222
Window Classes	230
Graphic Classes	241
Multiprocessing Classes.....	256
 Chapter 15 SMALLTALK/V 286 ENVIRONMENT	 263
The Keypad	263
Active Window	264
Cycling	265
Using Menus	265
Manipulating Windows	266
Panes	268
Text Editor	271
Saving the Image	274
Exiting Smalltalk/V	275
Evaluating Smalltalk Expressions	275

The System Dictionary	277
Maintaining Smalltalk/V	281
DOS Shell	288
Font and Cursor Shapes	289
Chapter 16 SMALLTALK/V 286 STANDARD WINDOWS	293
Disk Browser	293
Class Hierarchy Browser	297
Class Browser	301
The Inspector	303
Debugger Windows	304
Method Browser	308
PART 4: ENCYCLOPEDIA OF CLASSES	311
APPENDICES	491
Appendix 1 SMALLTALK SYNTAX SUMMARY	491
How Syntax is Specified	491
Smalltalk Syntax	492
Appendix 2 PRIMITIVE METHODS	495
How Primitive Methods Work	495
Primitive Number Assignments	495
User Defined Primitive Methods	499
Appendix 3 CONFIGURING SMALLTALK/V	505
Memory Configuration	505
Hardware and BIOS Configuration	507
Speed vs Space	508
Appendix 4 METHOD INDEX	509
Index	547



INTRODUCTION

Welcome to **Smalltalk/V** and the world of object-oriented programming systems or, more often, OOPS for short. You've joined the world's largest community of Smalltalk users. Owners of Digitalk's **Smalltalk/V** are people, like you, who want to squeeze maximum power and performance out of their MS-DOS, OS/2 and Macintosh computers.

You're in good company. **Smalltalk/V** is found widely in academic and research laboratories, R&D and product development departments of Fortune 1000 corporations, systems development agencies in government as well as on home PCs for recreational and entrepreneurial pursuits. **Smalltalk/V** applications have been developed in the areas of simulation, expert systems, intelligent tutoring computer-based instruction, database query interfaces, computerized typesetting and integrated programming environments.

Smalltalk/V is selected by so many for such diverse applications because Smalltalk is both a *powerful language*—you can get a lot of activity out of a few lines of code—and a *powerful program development environment*—software utilities help you to reuse as many lines of pre-written code as possible and, once copied, to quickly edit and correct errors in such code for your own program.

To encourage an exploratory "design-prototype-refine" approach to application development, **Smalltalk/V** lets you edit and install small code modules without lengthy compile and link sessions, building a program piece by piece and seeing the results immediately. You experiment with bits and pieces of a program long before it is complete, exploring ideas, structures and algorithms as the application takes form.

Except for a small kernel in machine language, **Smalltalk/V** is written in **Smalltalk/V**. Commented source code for virtually the entire system is supplied in digestible chunks of source code which you can reuse and modify in your applications.

Smalltalk/V features pure object-oriented programming, a revolutionary approach to data abstraction, providing a new dimension in which to organize the elements of a software system. For you, this means highly reusable software, truly generic code and the opportunity to use a prototyping style of software development.

This book is intended for both people who have never used Smalltalk as well as experienced Smalltalk programmers. It's organized into five parts:

- **Part 1, Overview**, introduces object-oriented programming through a discussion of Smalltalk's big ideas and concludes with a comparison of Smalltalk and Pascal versions of an example program.

- **Part 2, The Smalltalk/V Tutorials**, is a series of tutorials that teach the Smalltalk language through examples you run in the **Smalltalk/V** environment.
- **Part 3, The Smalltalk/V 286 Reference**, is a complete specification of **Smalltalk/V 286**. You'll find summaries of Smalltalk's syntax and semantics, descriptions of windows and menus that make up the environment, and a rundown of the major building blocks (classes) included in the system.
- **Part 4, The Encyclopedia of Classes**, is a comprehensive, structured description of the classes and methods in **Smalltalk/V 286**.
- **Appendices** cover advanced features such as writing your own Smalltalk primitives and extensions in other languages and conclude with a detailed cross-referencing Method Index and Index.

System Requirements

Smalltalk/V286 requires an IBM-PC, PS/2 or compatible, with an 80286 or 80386 processor and the following equipment:

- 1 Megabyte of RAM
- Hard disk and one diskette drive
- Monochrome or color monitor
- Graphics controller (either CGA, MCGA, EGA, VGA, Hercules, Toshiba, or AT&T)
- PC-DOS or MS-DOS, Version 2.0 or later

The following items are optional:

- Expansion to 16 Megabytes of RAM (extended memory only)
- Mouse (highly recommended, Microsoft compatible)
- Floating point co-processor (80287 or 80387)

Before Starting

Before proceeding, please take a moment to make sure that you have the complete **Smalltalk/V 286** package:

- Two diskettes labeled **Image** and **Source**
- This book
- Registration Card

The diskettes are not copy-protected. Using DOS disk copying utilities, you can make one or several backup copies, as long as they are for archival purposes so you can protect your investment.

The **Smalltalk/V** community is growing daily. Digitalk's user newsletter *SCOOP*, keeps registered **Smalltalk/V** users informed of programming hints, product upgrade information, bug reports, available Goodies packs, and special licensing and pricing information. To make the most of your **Smalltalk/V** investment, and to enable us to serve you more quickly when you need support, return the enclosed Registration Card and join the **Smalltalk/V** community to stay well-informed.

Sign the Registration Card and mail it to:

Digitalk Inc.
9841 Airport Boulevard
Los Angeles, California 90045

Part 1

Overview

1 INTRODUCTION TO THE SMALLTALK LANGUAGE

You do not have to read this chapter to get going with *Smalltalk/V*. One legitimate school of Smalltalk thought suggests that the best introduction to object-oriented programming is simply to jump right in—to learn Smalltalk by experience. If this notion appeals to you, proceed directly to Chapter 2. You may want to return here to supplement your experience. But there is nothing in this chapter that you have to know to understand and make effective use of *Smalltalk/V*.

If this were a book on driving a car, this Overview describes a bit of the "physics" behind the car's engine, drive chain and suspension—hardly prerequisite knowledge to the act of driving. But racing drivers will tell you that the more you know about how your car works, the better you can drive it—knowing how to pull maximum performance from the potential of the car's interacting component parts. If this notion appeals to you, proceed.

Smalltalk's Big Ideas

Smalltalk grew from a few powerful ideas.

- The most important component in a computing system is the individual human user.
- Programming should be a natural extension of thinking.
- Programming should be a dynamic, evolutionary process consistent with the model of human learning activity.
- A computing environment is both a language and a productivity enhancing interface of programmer/user "power tools"—utilities to express yourself in that language and to organize and flexibly use both procedural and factual knowledge.

Smalltalk embodies these ideas in a framework for human/computer communication. At the simplest level, Smalltalk is yet another programming language like Basic, C, Pascal or Lisp. You will see in this chapter how you can write Smalltalk programs that have the "look and feel" of conventional Pascal or other familiar programming languages.

You will also see how some thirty lines of Pascal, or less than twenty lines of "Pascalese" Smalltalk, can be reduced to five lines of Smalltalk the way it is meant to be written. And that's not five lines of dense, cryptic syntaxes like C or APL allows, coding shortcuts that come back to haunt you in application maintenance and enhancement costs.

If we try to build an ideal machine that lives up to the promise of the big ideas above, we would want a computing environment that is both very hardy and forgiving. If programming is to be a natural extension of thinking and learning, the system has to take programming errors in stride—a simple coding error can't crash the system or you'd lose all incentive to use an exploratory prototyping style of application development.

Smalltalk promotes the development of **safe** systems. Smalltalk "errors" are merely objects telling you they do not understand how to do what you are asking them to do—hardly events which blow up the system. And Smalltalk's encapsulation of digestible chunks of program code with their own local data in independently active objects promotes a **"divide and conquer"** approach to programming problem solving. Smalltalk objects are easily **inspected**, **duplicated**, modified and, perhaps most importantly, **re-used**. Smalltalk lets you get on to the business of solving your problem, not writing the same code over and over.

The Tutorials will introduce you to the range of programming "power tools" standard in **Smalltalk/V** that help you use, re-use and modify the storehouse of Smalltalk source code which is part of the basic system. But first, it can be helpful to understand that Smalltalk is both very much like and, at the same time, very much unlike conventional programming languages.

We'll then introduce you to some of the special terminology and exciting ideas that energize object-oriented programming in Smalltalk. From there it's on to the introductory tutorial which gets you up and running and writing your first **Smalltalk/V** code.

Smalltalk vs. Conventional Languages

This section presents an overview of **Smalltalk/V** by comparing examples of code in both Smalltalk and Pascal to help you learn **Smalltalk/V** more quickly. You don't have to be a Pascal programmer to benefit from the comparison as thorough explanations accompany each example.

The step-by-step code examples are followed by a complete program written in both languages which solves the same problem. We conclude by rewriting the Smalltalk version of the algorithm, taking advantage of object-oriented features to significantly reduce the amount of code required to do the same procedure.

The examples which follow present a series of statements in Pascal and **Smalltalk/V**. The left column shows program fragments in Pascal, while the right column shows equivalent code fragments in **Smalltalk/V**.

Assignment to a Scalar Variable

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

These statements look the same in both Pascal and Smalltalk. The assignment operator is `:=`. Variable names have the same syntax in both languages. In the example statements, the contents of variable `b` are added to the contents of variable `c` and stored in variable `a`. In Pascal, the computed value is stored. In Smalltalk, assignment statements always store pointers to objects which contain the values.

A Series of Statements/Expressions

<code>x := 0;</code>	<code>x := 0.</code>
<code>y := 'answer';</code>	<code>y := 'answer'.</code>
<code>z := w</code>	<code>z := w</code>

The statement separator is semicolon in Pascal and period in Smalltalk. Note that in both languages, the statement separator character is not used after the last statement in the series. The first statement assigns the constant zero to the variable `x`. The second statement assigns a literal string to the variable `y`. In both languages, a string is an array of characters. The third statement assigns the contents of variable `w` to variable `z`.

A Function Call with One Argument

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

The function `size` is called with argument `array` and the value returned is stored in the variable `a`. In Smalltalk, calling a function is known as *sending a message*. In this case, the message `size` is sent to the contents of variable `array`.

Function Calls with Two Arguments

<code>x := max(x1, x2);</code>	<code>x := x1 max: x2.</code>
<code>y := sum(p, q)</code>	<code>y := p + q</code>

In Pascal, the arguments to the function call are enclosed in parentheses. In Smalltalk, for a two-argument message, the arguments precede and follow the message name. Note that in Smalltalk, the standard arithmetic operations are performed via messages. In the first example, the message `max:` is sent to the contents of variable `x1` (the first argument), with the contents of `x2` as the second argument. The result returned is assigned to the variable `x`. In the second example, the message `+` is sent to the contents of variable `p` with the contents of variable `q` as the second argument, and the result returned is assigned to the variable `y`.

A Function Call with Three Arguments

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

When a message has three or more arguments in Smalltalk, the name of the message is split into **pieces**, and a **piece of the message name appears preceding each of the arguments after the first**. This distribution of the message name helps to describe the message arguments. In the example, the **message name is between:and:** and the arguments are variables **x**, **x1**, and **x2**. This example could be used to test whether the value of **x** is between the values of **x1** and **x2**, and assign the Boolean result (true or false) to the variable **b**.

Subscripted Variable Access

x := a[i];	x := a at: i.
a[i + 1] := y;	a at: i + 1 put: y.
a[i + 1] := a[i]	a at: i + 1 put: (a at: i)

Pascal uses square brackets to specify subscripting, whereas Smalltalk uses **at:** and **at:put:** messages. In the first example, the value of variable **i** is used to index the array identified by variable **a**, and the value obtained is stored in variable **x**.

The second example shows replacing an element of an array with a new value. Note that a Pascal assignment may store into an array element, whereas in Smalltalk only scalar variables appear to the left of an assignment statement, so an **at:put:** message is used.

The third example shows accessing and changing array elements. Parentheses are used in the Smalltalk example to specify evaluation order.

If Statements

if a < b then	a < b
a := a + 1;	ifTrue: [a := a + 1].
if atEnd(stream) then	stream atEnd
reset(stream)	ifTrue: [stream reset]
else	ifFalse: [c := stream next]
c := next(stream)	

Pascal and Smalltalk provide similar capabilities for the conditional execution of a series of statements based on the result of evaluating a boolean expression. In Smalltalk, the conditional statements are enclosed in square brackets. In the first example above, the variable **a** will be incremented by one if the value of variable **a** is less than the value of variable **b**.

The second example illustrates conditionally executed code during file access. The file being accessed is identified by the variable `stream`. If the file is positioned at the end; the `reset` message is sent to reposition it at the beginning. Otherwise, the variable `c` is assigned the next character in the file.

Iterative Statements

<pre>while i < 10 do begin sum := sum + a[i]; i := i + 1 end;</pre>	$\begin{aligned} & [i < 10] \\ & \text{whileTrue: [} \\ & \quad \text{sum := sum + (a at: i).} \\ & \quad \text{i := i + 1}. \end{aligned}$
<pre>for i := 1 to 10 do a[i] := 0</pre>	$\begin{aligned} & 1 \text{ to: } 10 \text{ do: [:i } \\ & \quad \text{a at: i put: 0}] \end{aligned}$

Pascal and Smalltalk provide similar capabilities for repeated execution of a series of statements. In the first example, the two statements in the loop will be executed as long as the value of the variable `i` is less than 10. In the second example, the single statement in the loop will be executed with the variable `i` taking on the values 1 through 10 in succession.

Returning Function Results

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

Pascal and Smalltalk both provide for specifying the result of function (or in Smalltalk, method) evaluation. In Pascal, the function result expression is assigned to the function name, which serves as a variable for containing the result. In Smalltalk the caret (`^`) appears before an expression that is the method result. This causes method execution to cease and the value of the expression to be returned as the method result. In the example, the value of the variable `answer` is the function (and method) result.

Storage Allocation and De-allocation

<pre>new(p) dispose(p)</pre>	$p := \text{Array new: } 5$
------------------------------	-----------------------------

Pascal and Smalltalk both provide for the dynamic allocation of variables (in Smalltalk terminology, objects). In the first line of the example above, both languages assign to the variable `p` a pointer to the newly allocated object. In Pascal, however, it is necessary to explicitly de-allocate objects when they are no longer needed in order to reclaim their space. This is done via the `dispose` function call. In Smalltalk, space reclamation (garbage

collection) is automatic and consequently, there are no language facilities for specifying object de-allocation. This simplifies programming by eliminating a potential source of error, de-allocating at the wrong time.

A Complete Program

What follows is a complete program with Pascal code on the left, Smalltalk on the right.

program frequency;

```

const
size = 80;
var
s: string [size];           | s c f k |
i: integer;
c: char;
f: array[1..26]             f := Array new: 26.
    of integer;
k: integer;
begin
writeln('enter line');      s := Prompter
readln(s);                  prompt: 'enter line'
                                default: "".
for i := 1 to 26 do          1 to: 26 do: [ :i |
    f[i] := 0;                f at: i put: 0].
for i := 1 to size do        1 to: s size do: [ :i |
    begin
        c :=                   c := (s at: i) asLowerCase.
        asLowerCase(s[i]);
        if isLetter(c) then       c isLetter
            begin                 ifTrue: [
                k := ord(c)          k := c asciiValue
                - ord('a')          - $a asciiValue
                + 1;                + 1.
                f[k] := f[k] + 1      f at: k put: (f at: k) + 1
            end                     ]
        end;                   ].
for i := 1 to 26 do          ^f
    write(f[i], ' ')
end.

```

The programs above ask the user to enter a line of text from the keyboard. It then computes the frequency of occurrence of each alphabetic character in the input text. All characters are treated as lower-case letters.

The example emphasizes the similarities of Pascal and Smalltalk syntax. The algorithm used is identical in both cases. The input characters are examined one at a time and if they are characters, the frequency counter for that letter is incremented.

None of the powerful built-in building blocks of Smalltalk were used in the above example. The example below shows the same program written using some of these built-in building blocks.

```
| s f |
s := Prompter prompt: 'enter line' default: ''.
f := Bag new.
s do: [ :c | c isLetter ifTrue: [ f add: c asLowerCase ]].
^f
```

A **Prompter** is used to get the input string from the user. A **Prompter** is a special type of window. An empty **Bag** is then created to hold the character frequencies. Bags are a type of collection that count occurrences of objects. The input string is then iterated over, and each character is examined. If the character is a letter, its lower case equivalent is added to the **Bag**. The resultant **Bag** is then returned.

Already, Smalltalk is revealing its expressive power. The considerably shorter rewrite uses a few of Smalltalk's pre-defined objects, each comes with its own highly developed behavior. In the hundred forty or so classes of **Smalltalk/V** object types, there are over two thousand methods you can call upon. Each new object and its methods, which you create, will be added on equal footing with the generic objects which come in **Smalltalk/V**.

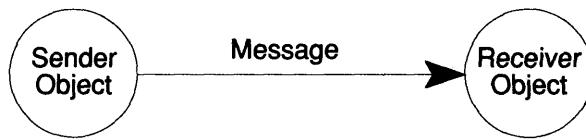
Objects obviously have something to offer—a tremendous source of Smalltalk programmer productivity. A greater appreciation of what objects are and how they behave is in order.

The World According to Objects

Smalltalk is built on the simple yet powerful model of "communicating objects" as shown in Figure 1.1. What could be more natural. We experience our world largely as a vast collection of discrete objects, acting and reacting in a shared environment.

* s := Prompter prompt: 'enter line'.
 |> f := Bag new.
 |> s do: [:c | c isLetter ifTrue: [f add: c asLowerCase, yourself]].
 |> ^f.

Figure 1.1
Communicating Objects



At the human social level we are a society of doctors, lawyers, beggars and thieves, etc. Although we are a population of unique individuals, we cluster in occupational groups based on the behavioral skills and knowledge we each develop and exhibit as seen below:

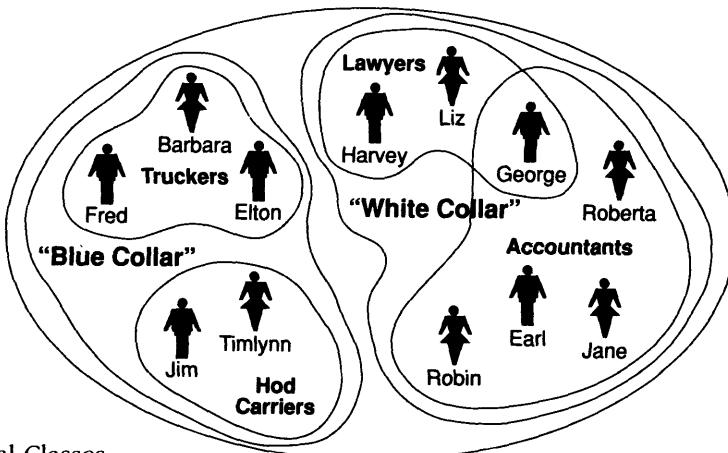


Figure 1.2
Human Occupational Classes

Break a leg, call in a doctor and tell him or her about your condition. You trust the doctor's special knowledge and skills to help make you better. Self communicates with Doctor Black Box.

Want to become a lawyer? You learn the law and how to behave like a lawyer. Then as corporate counsel in response to the MegaCorp CEO's question, "What's our exposure on this new project?", your answer is couched in legal considerations while the chief financial officer reflects on fiscal impacts.

In Smalltalk's object-oriented terms, occupational abstractions like doctor, lawyer, programmer, etc., are *classes* of which we individuals are *instances*. To become a lawyer, we learn legal *methods*. Communications between individuals are comparable to Smalltalk *messages*, their content equivalent to Smalltalk *selectors* as shown in Figure 1.3. Correspondence between our perception of the world and its representation in machine terms through Smalltalk gets at the heart of Smalltalk's power.

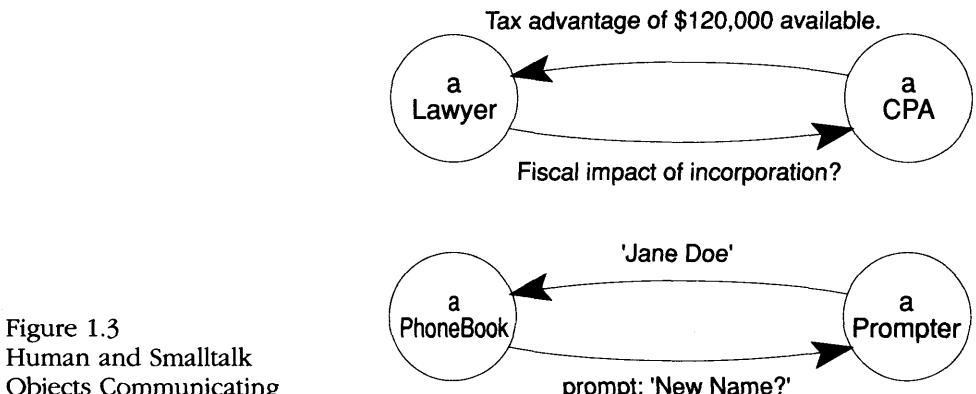


Figure 1.3
Human and Smalltalk
Objects Communicating

What are objects?

A Smalltalk object is simply related pieces of code and data. The pieces of code are Smalltalk *methods*—a library of self-contained subroutines unique to each class giving each class of object its specific behaviors. An object's data structure is described by its collection of *instance variables*.

When you create a specific *instance* of a class, the initial values of the object's instance variables are assigned. The object's methods are its know-how. If we were to create a Smalltalk "car driver object", it would likely include "brake", "steer", "watch for traffic" and "shift gears" methods. Instance variables of such a car driving object would include "reaction time", "temperament" and "visual acuity" of the driver.

Related data and program pieces are encapsulated within a Smalltalk object, a *communicating black box*. The black box can send and receive certain messages. *Message passing* is the only means of importing data for local manipulation within the black box. And if an object needs something done that it does not know how to do within its own set of methods, it sends a message to another object, in effect, *asking for assistance* in the completion of a task.

In Smalltalk, objects communicate to objects just as lawyers talk to accountants in our occupational analogy in Figure 1.3. A professional's know-how is comparable to a Smalltalk object's collection of methods. People communicate using their know-how.

Know-how does not communicate to know-how. The lawyer's knowledge used to prepare a client's will does not include a "direct memory access" to the accountant's ability to compute financial implications of the settlement of an estate. Similarly, a Smalltalk object's methods do not call other objects' methods directly. Rather, the lawyer's methods include knowing when to send a message requesting financial services, just as the CPA knows when and how to ask for legal services.

In OOPS terms, *information hiding*—as this encapsulation of code and data is known in computer science—makes for highly portable, easily modifiable and safe software. Large applications may be easily maintained since objects may be updated, recompiled, tested and called immediately back into service with their new behavioral capabilities on line.

What kinds of objects can be described?

Like their physical counterparts, Smalltalk objects have attributes and exhibit behaviors. Since everything in Smalltalk is an object—including the Smalltalk environment itself—then what you can do with the language becomes a question of what objects can be described and manipulated.

If the encapsulation of information hiding provides the means for creating objects, then a language's *data abstraction* capabilities determine what objects can be described. Marco Polo called upon his powers of data abstraction daily as he traveled to parts unknown. Things which could not be understood or named within his current world view required invention, *new words for new objects*.

You need the same powers of an extendible language capable of describing arbitrary data structures if you are to tailor the generic Smalltalk environment to your purposes. Smalltalk lets you create arbitrary new data structures, compound objects which can be thought of as an array whose elements can be any combination of numbers, symbols or character strings as well as another array making nested data structures possible. Where it is generally an exception or nuisance in conventional languages, creating new data structures is done routinely when you define a new class or subclass of objects in Smalltalk.

How do objects communicate and behave?

Smalltalk objects take responsibility for their own actions, responding individually to every message. Your application may have occasion to print an integer, a floating point number, an ASCII character or a string of symbols. Since each of these elementary data types is defined as a Smalltalk class, instances of these classes come with a bundle of behavioral features built-in, its methods. Each of the elementary data types knows how to perform generally required behaviors such as print, duplicate and comparison operations.

So when it comes time to print, your Smalltalk application simply sends the near universal message **printString** to each of the variety of data type objects to be included in a report:

```
'This is a string' printString.  
423 printString.  
#(123) printString.  
$A printString.  
#('array of' 3 'strings and' 2 'numbers') printString.
```

Integers, arrays and characters take care of getting themselves represented on paper. This Smalltalk characteristic of having different objects responding uniquely to the same message is known as *polymorphism*. It means you won't have to memorize a unique vocabulary for each class used in building your applications.

Because Smalltalk objects take responsibility for their own behavior, you won't have to litter your application with conditional checks through case statements to see that the proper type of function is called to operate on a piece of data. This feature saves much time and significantly reduces software maintenance costs since only affected objects need be edited and re-compiled to enhance a Smalltalk application.

Among the many reasons objects communicate, a frequent objective is to **change the state** of the receiver object. Sending a message to store a new value in a variable (a Smalltalk variable is an object that stores other objects) is an obvious example of state changing messaging. A bit more subtle is a user request to resize a window which results in a mouse-based interaction which sends new screen coordinates to the window's instance variables which store its size and location on the screen. The window's state changes upon receipt of the **resize** message.

Smalltalk guarantees that there will be a response by a message recipient. If an object determines that it does not know how to perform a requested behavior, it will at least answer with a "Message not understood" response message. The method which sends this response also kicks in Smalltalk's debugging utilities to help you determine and correct the **failure to communicate**.

So even program errors are detected and resolved within the object-oriented messaging framework of Smalltalk. This makes for a very exploratory environment in which to develop application software. **A working prototype can be constructed quickly and enhancements integrated easily into the evolving system.**

How does Smalltalk organize objects and their methods?

Smalltalk organizes its classes into a hierarchy of classes and subclasses, a portion of which is shown in Figure 1.4. For example, the **Integer** class is a subclass of the **Number** class which is a subclass of **Magnitude** which is a subclass of class **Object**, the most general Smalltalk class and parent of all other classes.

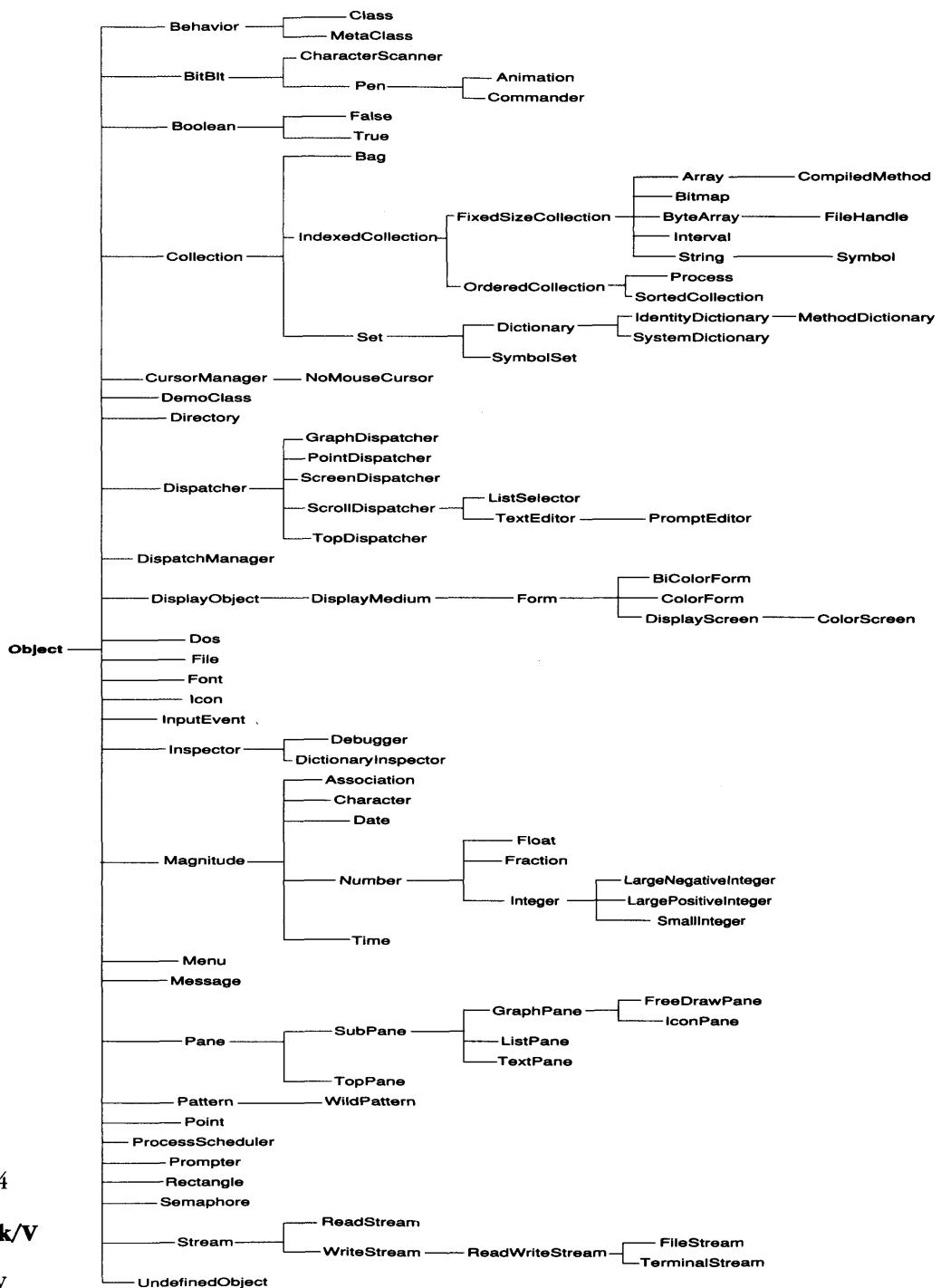


Figure 1.4
Partial
Smalltalk/V
Class
Hierarchy

Inheritance provides a mechanism for both organizing and maintaining the collection of Smalltalk object classes. Inheritance recognizes similarities among objects, capitalizing on the fact that similar objects often behave similarly.

A subclass inherits all the methods known to the parent. If you define a new customer database class to be a type of Smalltalk **SortedCollection**, your application will automatically know how to add, copy, edit, remove, print and sort customer records. You need only create methods which describe the new behaviors the customer database must exhibit beyond those of the generic **SortedCollection**.

If a selected parent's behavior is inappropriate, you simply define a method by the same name to override the parental way of doing things. For instance, to protect confidential customer information, the print method of the **CustomerRecord** class could be written to perform a password checking operation before printing. The customer record would then enforce security when asked to print itself while a less rigorously defined object simply prints itself upon receipt of the same print message.

Smalltalk's inheritance features encourage "programming to exception and modification". Without inheritance, you would have to spend a good bit of your time telling new classes of objects how to do elementary things like print. And if you later revamp the procedures for printing specific data types, you would have potentially hundreds of individual print methods to update and re-compile if it were not for inheritance.

Only slight reflection is needed to appreciate the programmer productivity potential of Smalltalk's inheritance mechanism. A little more reflection will confirm that the hierarchy of Smalltalk classes is consistent with the model of classification systems we routinely lay over our perceptions of the world.

How do you maintain a Smalltalk world of objects?

Each Smalltalk object is an encapsulated program operating on its own local data, a little self-contained computer. The user extendable **Smalltalk/V** system comes with over two thousand pre-defined methods in over one hundred classes. With time, your personalized **Smalltalk/V** environment will contain hundreds, maybe thousands, of new objects, classes of objects and their associated methods.

If you had to explicitly manage program and data file handling for each Smalltalk object as well as act as traffic cop to the messaging activity, there would be no incentive to adopt object-oriented programming. The power of Smalltalk's natural human thinking model at the design level would be lost at code writing. **Smalltalk/V**, therefore, is designed to manage the "dirty work" of *object storage* and *memory management* for you.

Imagine the task for your hardware and operating system software if your **Smalltalk/V** directory were strewn with:

obj1.exe
obj1.prg
obj1.dat
obj2.exe
obj2.prg
obj2.dat
obj3.exe
obj3.prg
obj3.dat

. . . and so on for hundreds or thousands of objects if it were possible.

That's what it would take if Smalltalk's objects managed themselves independently under the conventional model of computing. Smalltalk solves this problem with a global solution, an object-oriented model for storage.

When you install **Smalltalk/V** at the beginning of **The Tutorial**, you will note two large files. These are the **Smalltalk/V** source and image files. The source file contains an ASCII representation of the Smalltalk source code. The image file is a computer readable snapshot of the current state of your **Smalltalk/V** environment, a kind of group photo preserving the state of each object.

Saving the image to permanent disk storage allows you to end a Smalltalk session, saving your objects in "suspended animation" to be revived exactly as you left them when you restart Smalltalk. In this global and efficient manner, Smalltalk takes care of storing objects.

Having an efficient approach to storage management might only serve to preserve chaos if RAM memory were to become cluttered with persistent though unused and unwanted objects, therefore taking up valuable real estate in RAM if no longer needed. **Smalltalk/V** lets you focus on programming while automatic memory management, sometimes unglamorously referred to as "garbage collection", maximizes RAM available for the creation of new objects and keeps accumulated "clutter" from crashing the system. Automated object storage and memory management further insure a safe environment for Smalltalk application development.

Ideas into Action

Smalltalk/V is a prescription for an exciting programming experience ... the rare combination of an open and safe system inviting you to shape its capabilities to your needs. The first several chapters of **The Tutorial** introduce you to the **Smalltalk/V** programming environment and to the range of objects and their behavior in the **Smalltalk/V** system.

But this is only the beginning. Smalltalk really shows its stuff in real world application development. The expanding collection of re-usable Smalltalk objects and their methods when used in an evolutionary, prototyping development cycle result in quick and efficient development of complex applications. The Tutorial concludes with an application development example which showcases the "design-test-improve" cycle of development encouraged by the fully object-oriented character of Smalltalk/V.

Part 2

Smalltalk/V 286 Tutorials

2 INTRODUCTION TO THE SMALLTALK/V 286 ENVIRONMENT

In this chapter, you will learn how to install Smalltalk/V, start up its environment, and edit and evaluate some of its most basic expressions. You'll learn how to move the cursor, use windows and enter and edit text. By the end of this chapter, you'll be familiar enough with Smalltalk/V to run the tutorials, which make up the remainder of Part 2 of this book.

If you have worked in a "windows" environment before, you may already be familiar with some ideas in this chapter. After **Installing** and **Starting Up Smalltalk/V**, you may want to take the **Quick Tour** and read the **Starting Out** sections of this chapter, then scan the rest.

If the Smalltalk/V operating environment is new to you, be sure that you understand the ideas in this chapter before moving on. You can learn more about the Smalltalk/V operating environment in **Part 3: Smalltalk/V 286 Reference**. Chapters 15 and 16 provide an in-depth discussion of many of the topics introduced in this chapter.

Installing Smalltalk/V

Before you begin installing Smalltalk/V, make sure that you have made backup copies of the **Source** and **Image** diskettes included in your Smalltalk/V package.

Copying Smalltalk/V to a Hard Disk

Copying Smalltalk/V to the hard disk involves two steps: 1) creating a subdirectory on your hard disk and 2) copying the Smalltalk/V files.

Creating a Subdirectory

Smalltalk/V must have its own subdirectory on your hard disk. You can create a Smalltalk/V subdirectory as a child of any directory already residing on your disk. Here we are going to create a subdirectory called **smalltalk**, but you can name your subdirectory any name you chose.

To create the subdirectory, enter the following command at the DOS prompt then hit the **return** or **enter** key:

C> mkdir \smalstalk

This creates a subdirectory called **smalstalk**, with the root directory as its parent on Drive "C". You may create a subdirectory from any directory on your disk. Refer to your DOS manual for instructions.

Copying Smalltalk/V Files

To copy **Smalltalk/V** files to your newly created subdirectory, insert the **Source** diskette in Drive A. Enter the following command at the DOS prompt and hit the **return** or **enter** key:

C> copy a:.* \smalstalk

This copies all files on the Drive A **Source** diskette to the hard disk subdirectory called **smalstalk**. After each file is copied to the hard disk, DOS reports back with a screen message. When the message "**xx files copied**" appears, DOS is finished copying the diskette.

Remove the **Source** diskette from Drive A, and replace it with the **Image** diskette. Repeat the procedure.

With the **Smalltalk/V** files copied to the hard disk, you are ready to run the installation program.

Running the Installation Program

To install **Smalltalk/V**, you need only know the type of video adapter board and monitor your system is using.

Supported Video Adapters

At time of publication, **Smalltalk/V 286** supports the following:

<i>Video Adapter Type</i>	<i>Monitor Type</i>
AT&T (640 x 400)	color/mono
CGA (640 x 200)	color/mono
MCGA (640 x 480)	color/mono
CGA Color Graphics (640 x 350)	color
EGA Color Low Resolution (640 x 200)	color
EGA Monochrome Graphics (640 x 350)	mono
VGA (640 x 480)	color/mono
Hercules Monochrome (720 x 348)	mono
Toshiba T3100 (640 x 400)	mono
IBM 3270 (720 x 350)	mono

The installation program will present you with an up-to-date list of video adapters supported by the version of the software you have.

The Installation Program

To install **Smalltalk/V**, make sure that the directory currently logged on the screen is your directory containing the **Smalltalk/V** files.

Now enter the following command:

install

When you hit **return** or **enter**, the installation program displays a list of supported video adapters. Enter the number corresponding to your hardware configuration followed by the **return** or **enter** key.

The installation program performs two functions. First it records your hardware configuration in the file **go** in the current directory. Second, it decompresses all of the **Smalltalk/V** files from the distribution archives. Then the archive files are deleted.

If your hardware configuration changes, you can run the installation program again. Since the archive files have been deleted, only the first function is performed.

Starting Up Smalltalk/V

Do not try to run **Smalltalk/V** unless you have already installed the **Smalltalk/V** program.

To start up **Smalltalk/V**, make sure that the current directory displayed on the screen is the directory holding **Smalltalk/V** programs.

Now enter the following command and hit **return** or **enter**:

v

Smalltalk/V then loads. When Smalltalk/V is ready to run, it will display the start-up screen below:

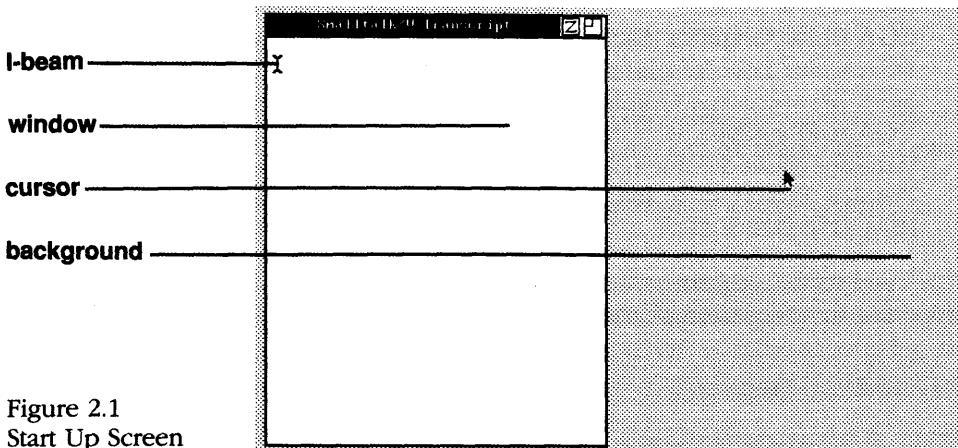


Figure 2.1
Start Up Screen

Exiting Smalltalk/V

To exit Smalltalk/V and return to DOS, move the cursor to the background and click the right mouse button to bring up the system menu. Select the **exit smalltalk** item by moving the cursor down the menu and clicking the left mouse button when the cursor is over your choice on the menu. The menu shown below will pop up on the screen:

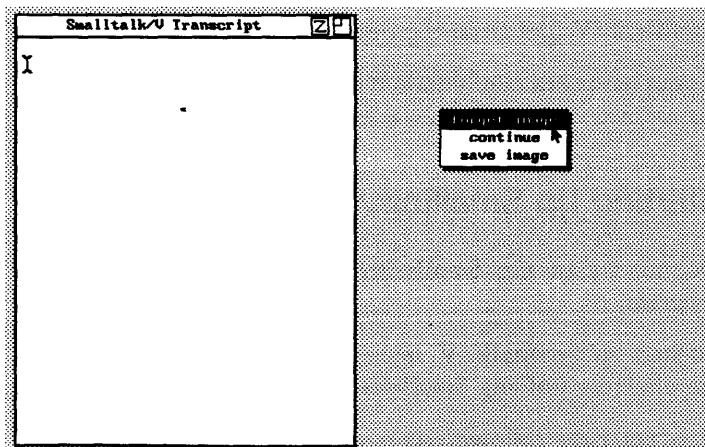


Figure 2.2
Exit Menu

This menu asks you whether or not to save your changes. For now, choose **forget image**; since you haven't done any programming yet, you don't have anything you need to save. Later, when you are actually programming in **Smalltalk/V**, you'll want to save your changes after each **Smalltalk/V** session.

Getting Around

To get around the **Smalltalk/V** environment, you must open up and close windows, make selections from popped-up menus and move the cursor using a mouse or your keyboard.

The Cursor

The cursor is your pointer on the screen. It tells **Smalltalk/V** where you are going to do something like pop-up a menu. You can move the cursor by dragging your mouse in the direction you want it to move or by using the cursor keys on the numeric keypad. When using the keypad, you can move the cursor in larger amounts by holding down the **shift** key while pressing down on a cursor key.

The I-beam

The insertion point or I-beam is a special text marker that is used when editing text strings. It appears in a text window or pane and marks the spot where new text will be inserted or deleted.

Working with Your Mouse

Using a mouse is the easiest way to get around the **Smalltalk/V** environment. With your mouse you can quickly move between windows and background, text and menus by dragging the mouse and clicking the correct button.

Mouse Buttons

Your mouse has two buttons. The **right** button "administrates" your way around the environment. Use the right button to bring up menus for selection and for scrolling within a window pane. The left button "selects" things for **Smalltalk/V** to execute. Use the **left** button to choose specific menu items, text or text lines, and objects from a list.

Mouse-key Equivalents

If you do not want to use a mouse, you can use your keyboard keys to get around the Smalltalk/V environment. Throughout this chapter you will find specific instructions for using the keyboard. For more keyboard instructions, refer to Chapter 15.

Windows and Menus

Smalltalk/V is a menu-driven system. You give commands to the environment by selecting things from a menu of possible things to do. There are several different windows and menus available in the Smalltalk/V environment.

This section of the chapter introduces the windows and menus of the Smalltalk/V standard window set. You will first learn some general information about windows and menus and then you will take a **Quick Tour** of the main Smalltalk/V windows and menus. For more detailed reading, see Chapters 15 and 16 in **Part 3** of this manual.

Windows

A window is an object with a border, label bar, window buttons and one or more panes and pop-up menus as shown in Figure 2.3. A window can be active or non-active. Windows can be opened, closed, collapsed, resized and moved around on the screen.

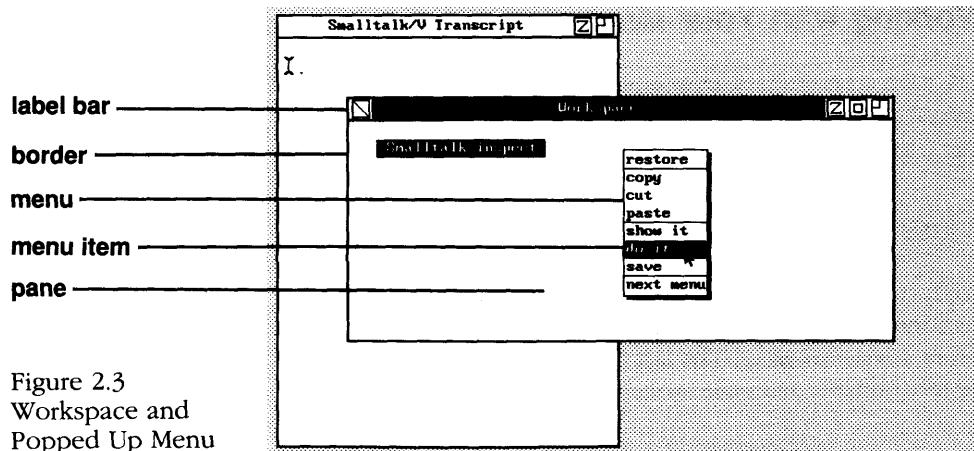


Figure 2.3
Workspace and
Popped Up Menu

Label Bar

Each window has its own label bar and menu. We will say more about menus a little further on. The window title is displayed on the label bar along with one or more small buttons, depending on which Smalltalk/V window is open. The buttons provide quick access to specific window activities.

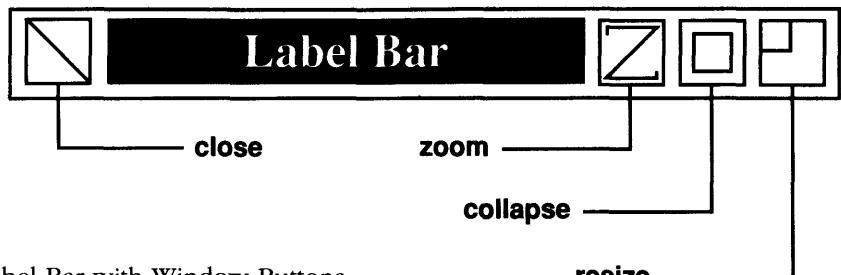


Figure 2.4
Window Label Bar with Window Buttons

Close Button: When selected, the window closes and disappears from the screen. To close the window, place the cursor onto the close button and click the left mouse button or use the numeric keypad + key.

Zoom Button: When selected, Smalltalk/V zooms in on the text pane so that it fills the whole screen. To select the zoom button, place the cursor onto the button and click the left mouse button or use the numeric keypad + key. To unzoom the text pane, click on the label bar and the window redraws to its original configuration. You can also use the F8 function key to zoom a text pane.

Collapse Button: When selected, the window collapses to show only the label bar. If the window is already collapsed, selecting this button expands the window to its original size and position on the screen. To select the collapse button, place the cursor onto the button and click the left mouse button or use the numeric keypad + key.

Resize Button: Select this button and the system responds with a rectangle outline for resizing the window. You can resize the window using either the mouse or keyboard.

With a Mouse: Be sure that the cursor is on the resize button. Press and hold down the left mouse button while you move the mouse to drag the cursor and resize the rectangle outline. Release the mouse button and the window redraws to its new size.

With the Keypad: With the cursor on the resize button, press and release the numeric keypad + key. Use the keyboard cursor keys to move the cursor and resize the rectangle outline. Once again, press and release the numeric keypad + key and the window redraws to its new size.

Pane

Each window has one or more panes, depending upon which Smalltalk/V window is open. Each pane also has a menu. The pane is the workspace for programming in Smalltalk/V. Here you select items from lists using the left mouse button, insert and edit text strings using the I-beam, and select and evaluate Smalltalk/V programming code.

Each pane also has a scroll bar and cursor available for scrolling through all the text contained in the pane as shown in Figure 2.5. You can bring up the scroll bar by pointing the cursor onto the pane and holding down the right mouse button. We will discuss scrolling in greater detail a little further on.

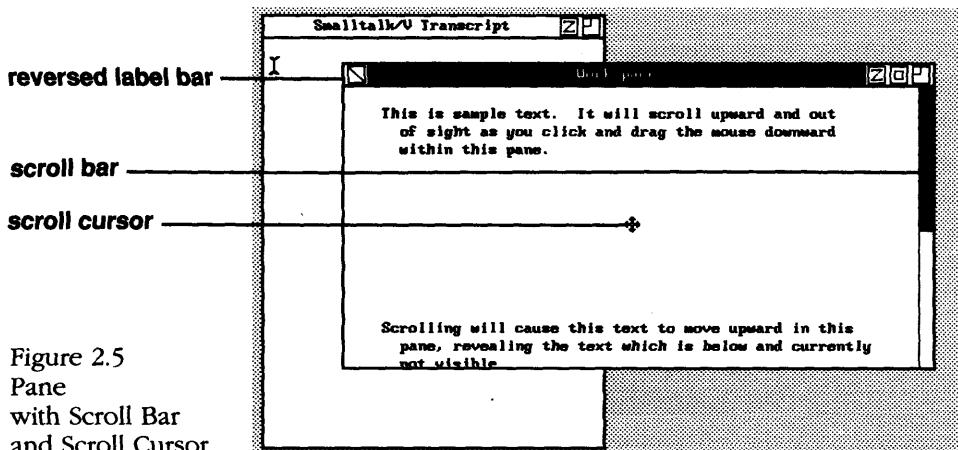


Figure 2.5
Pane
with Scroll Bar
and Scroll Cursor

Menus

There are many menus in the Smalltalk/V environment. A menu is an object containing a list of choices relevant to a particular window, pane or other object. Menus are hidden behind windows, panes and the system screen. You can make a menu visible by popping up the menu using the mouse or keypad.

Popping up Menus

To pop up a menu, position the cursor over the object hiding the menu. Then, click the right mouse button or press the **Del** key on the keyboard. The menu pops up.

Selecting from a Menu

To select an item from a menu, move the cursor over the item you wish to select. Click the left mouse button or press the **+** key on the numeric keypad.

Hiding a Menu

If you are using a mouse, move the cursor outside of the menu and click either mouse button. The menu disappears. If you are using a keypad, use the cursor keys to move the cursor out of the menu and the menu automatically disappears.

Finding Different Menus

Smalltalk/V has many different kinds of menus some of which are shown in Figure 2.6. What kind of menu you get depends on where the cursor is when you pop it up. If the cursor is outside of every window, the system menu is popped up. When the cursor is on the window label bar, a window menu pops up. When the cursor is on a pane, a pane menu specific to that pane pops up.

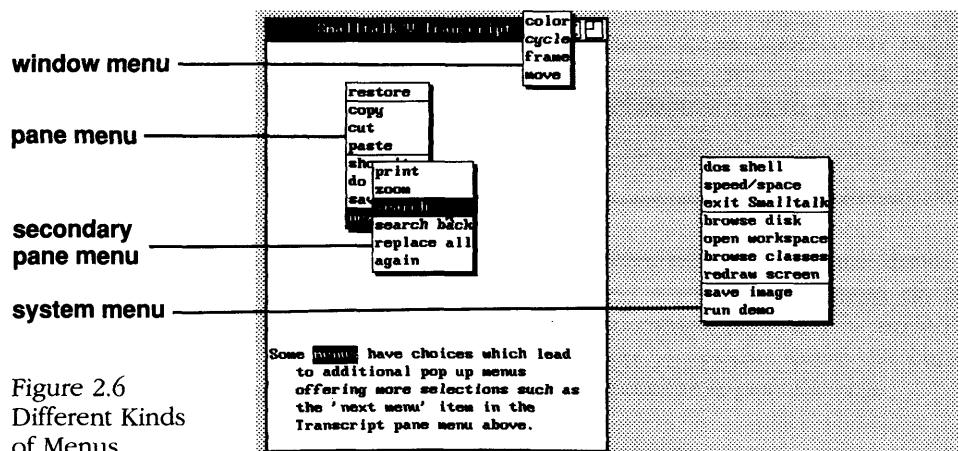


Figure 2.6
Different Kinds
of Menus

The System Menu

The system menu lets you open new windows and perform system level functions, such as exiting Smalltalk/V or redrawing the screen.

To find this menu, place the cursor outside any window onto the screen background. Then click the right mouse button or press the Del key.

The Window Menu

The window menu lets you manipulate the window as a whole. For example, you can resize the window by selecting frame from the menu. Or you can deactivate the active window and activate the window behind it by selecting cycle from the menu.

To quickly find this menu from any part of the window, press the Ins key. Or, you can place the cursor over the window label bar, and click the right mouse button or press the Del key.

The Pane Menu

The pane menu lets you manipulate the contents of the window pane. For example you can evaluate selected Smalltalk/V code by choosing show it or do it from the pane menu. Or, you can edit selected Smalltalk/V code by choosing copy, cut or paste.

To find this menu, place the cursor anywhere inside the window pane and click the right mouse button or press the Del key.

Using Menus: Running the Demonstration Program

To get you started, we've included a demonstration program, showing you how to use menus and move the cursor. To start the demonstration program, move the cursor outside of all of the windows, and pop up the menu. You should see the following screen:

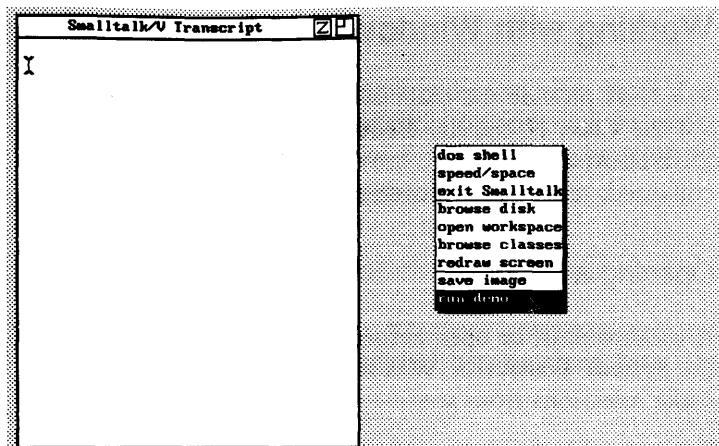


Figure 2.7
Starting the Demo

Now select the item **run demo** from the menu. (If **run demo** is not on the menu, you didn't have the cursor outside of all the windows when you popped up the menu. Try again.) When you select **run demo**, you'll see a special menu:

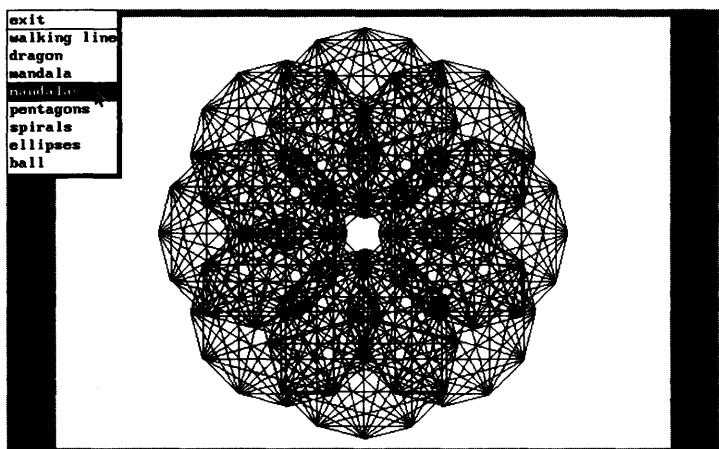


Figure 2.8
Running the Demo

When you select any item from this menu, the demonstration program draws a corresponding picture. For example, select **mandala** and a mandala is drawn on the screen. Select **ball** and you'll see a little greater program complexity. As you move the cursor, the ball moves differently. The ball moves faster when you move the cursor towards the bottom of the screen, and slower when the cursor is near the top.

To leave the demo, select the **exit** item from the demo menu.

Working with Windows

Windows provide the main interface between Smalltalk/V and you. For example, you use the **Class Hierarchy Browser** window to enter programs into the system and the **Disk Browser** window to browse and manipulate files. A window can be opened, closed, collapsed, resized and moved about on the screen.

Opening a Window

A window must be open before it can be activated. When you first start up Smalltalk/V, the **System Transcript Window** is opened. To open other Smalltalk/V windows you make selections from specific menus.

For example when you select **browse disk** from the **System Menu** the **Disk Browser Window** is opened. Or select **browse classes** from the same menu and the **Class Hierarchy Browser** window opens at a default size appropriate to the window type. If you want to change the size of a window, use the resize button on the label bar.

Activating a Window

Although there may be many windows visible on the screen at one time, only one of them can be active. To use a window, you need to activate it.

To activate a window, move the cursor over some portion of the window and click the left mouse button or press the **+** key on the numeric keypad. Smalltalk/V reverses the color of the label bar to show that the window is active and ready for use.

Deactivating a Window

You can deactivate a window by selecting another window on the screen. Click on the inactive window and its label bar reverses color to show it is active as the old window becomes deactivated.

If there are no other windows visible on the screen, move the cursor outside the window. Then click either the left mouse button or press the **+** key on the numeric keypad. This deactivates the window on the screen.

Resizing a Window

An active window can be resized to your own taste or to allow you to view more than one window at a time on the screen. You can resize a window by selecting the **resize button** from the label bar and dragging the cursor to resize the window, or by selecting **frame** from the window menu and following the same procedure.

Closing a Window

When you are finished with a window, you may close it. To close a window you can either select the **close window button** on the window label bar, or **close** from the window menu.

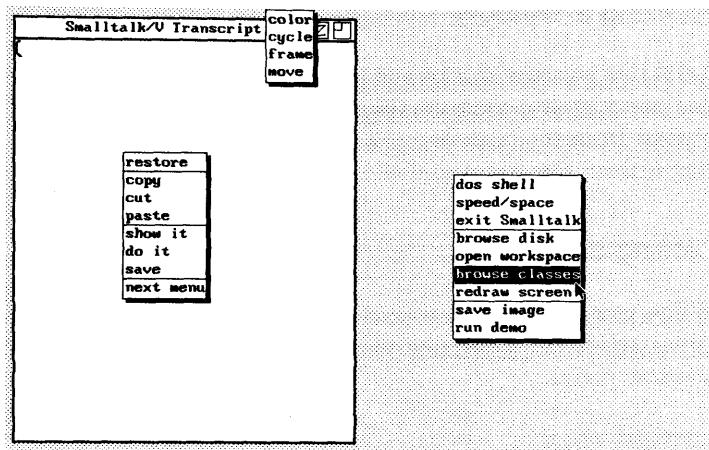
Moving a Window

You can move windows. This is useful when you have collapsed or resized windows and you want to reposition them on the screen.

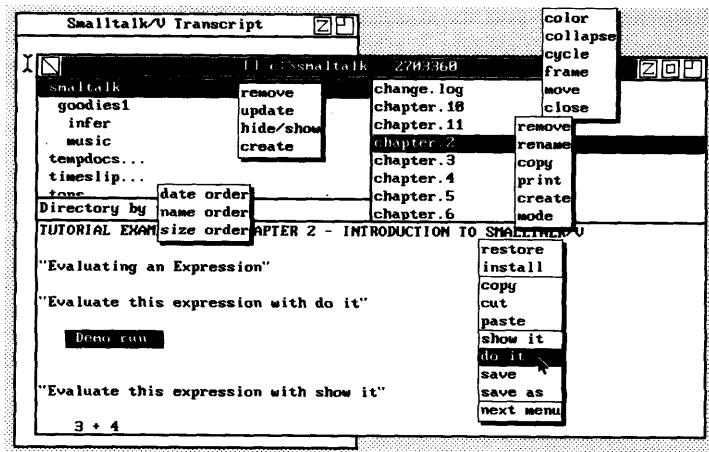
You can move a window in three ways. 1) Select **move** from the window menu. Then drag the cursor and its attached window outline, to its new position and click the left mouse button. The window is redrawn in its new location. 2) Place the cursor on the window label bar then press and hold the left mouse button. Drag the mouse and relocate the window on the screen. 3) Select **move** from the window menu using the **+** key on the numeric keypad. After the cursor grabs the window frame, use the cursor keys to move and reposition the window. Press the **+** key again and the window is redrawn in its new location.

Quick-Tour—Windows and Menus

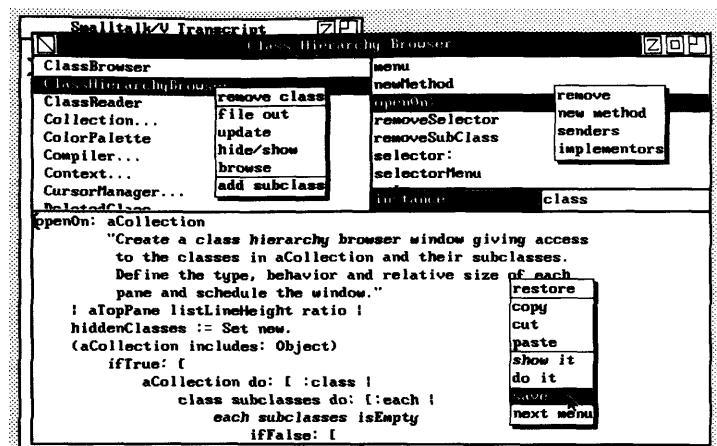
Part of the enjoyment of programming in Smalltalk/V is that you have lots of freedom to experiment. To whet your appetite, a Quick Tour of the main windows and menus of the Smalltalk/V environment can be found on the following pages. You can use the Quick Tour to preview the environment or as a roadmap in case you get lost. Detailed information on the Smalltalk/V windows and menus is found in Chapter 15 in Part 3 of this manual.



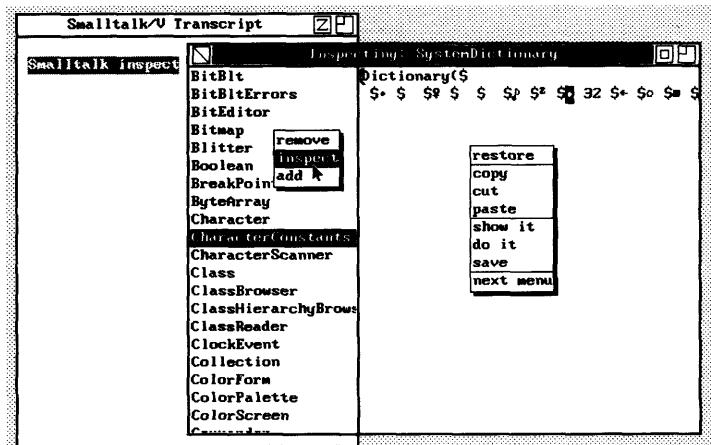
The System Menu and Transcript Window. The Transcript Window appears when you first run Smalltalk/V. By selecting from the System Menu, you can open other windows or perform system level functions.



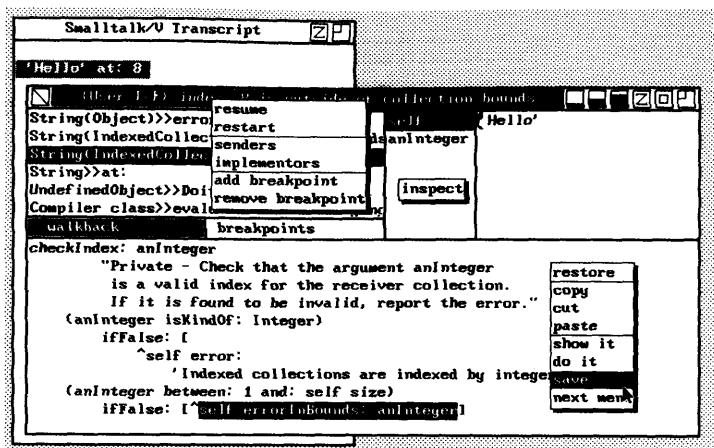
Disk Browser. The Disk Browser displays and lets you edit the files on a given directory and their contents.



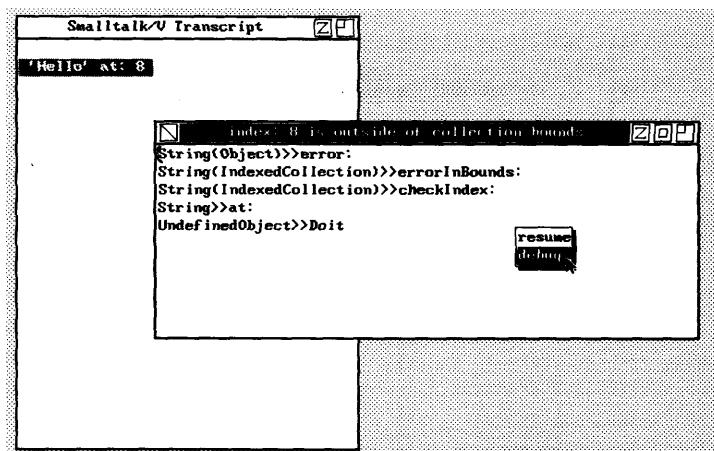
The Class Hierarchy Browser. The Class Hierarchy Browser shows you the interrelationship of the classes within Smalltalk/V, and lets you edit the code for each class.



Inspector. Inspectors let you examine and edit objects. They serve as a low-level debugging aid.



Walkback. The Walkback window pops up automatically when errors are detected. It gives a view of the state of your program at that point.



Debugger. The Debugger gives an expanded view of the Walkback in four panes. It is a high-level debugging aid to help you correct programming errors.

Inside the Window Pane

Most of what you do in **Smalltalk/V** will involve activities inside a window pane. For many of the programs that you write, you will be able to find program code that has already been typed into **Smalltalk/V**. You can copy this code and edit the text strings to produce the new code you are trying to write. This saves you from having to re-invent source code each time you want to do something in **Smalltalk/V**.

In this section you will learn how to enter and edit text strings and how to make the most of the powerful **cut**, **copy** and **paste** methods found in the pane menu. For more detailed discussion of these subjects, see Chapter 15.

Selecting Text

To do almost anything in **Smalltalk/V**, you must select some text to work with. This simply means marking any piece of data, such as a block of code, an expression, some text lines, etc.

Once you select text you can **copy** or **cut** it from its current location then **paste** it elsewhere using the relevant menu selection from the pane menu.

When working with text, you will see an I-beam in the text pane. The I-beam is the special pointer used when selecting or editing text strings. The I-beam marks the selection point for working in text. You can position the I-beam anywhere you like in the pane, even between letters in a word.

Selecting Text Using the Mouse

To select text using the mouse, first activate the window. Move the cursor to one end of the text string. Press and hold down the left mouse button. While continuing to hold down the button, move the cursor to the other end of the text you wish to select. As you move the cursor, text is reversed. When the text you want to select is reversed, release the left mouse button. The selected text remains reversed until you deselect it. To deselect text, click the left mouse button somewhere else in the window pane.

Selecting Text Using the Keyboard

To select text using the keyboard, make sure the window is active. Move the cursor to one end of the text you wish to select and press the **+** key on the numeric keypad. Then move the cursor to the other end of the text, and press the **-** key on the numeric keypad. The selected text is reversed. If you did not select exactly what you wanted, move the cursor and press the **-** key again.

Selecting a different piece of text is done in exactly the same way. Smalltalk/V deselects the old selected text and reverses the newly selected text strings.

Quick Selection of Single Words and Single Lines

To quickly select a whole word in a text string, position the cursor anywhere on the word to be selected and click the left mouse button twice or press the **+** key twice in the same place—a double click. The whole word is reversed without having to drag the cursor from one end of the word to the other.

To quickly select a whole line of text, place the cursor just inside the window border to the left of the line you wish to select and click the left mouse button twice or press the **+** key twice. The whole line is selected. If you continue holding down the mouse button while you drag the mouse either up or down, line after line of text is selected until you let go of the mouse button.

Scrolling Text

The text inside of a pane can be larger than the pane itself. Scrolling moves different text into the pane for viewing. To scroll text into the pane you can use either the mouse or the keyboard.

Scrolling Text Using the Mouse

Make sure that the window is active. Then press the right mouse button and hold it down. The cursor changes from an arrow to a scroll cursor and the scroll bar appears on the right side of the pane. The shaded area in the scroll bar shows where the text in the window is generally located in the overall document.

Continue to hold down the button while dragging the mouse downward. The scroll cursor changes to an arrow. Drag the arrow downward outside the text pane and the text in the pane scrolls up. Drag the arrow up beyond the top of the pane and the text scrolls downward. You can use the same technique to scroll text left and right across the pane by dragging the arrow beyond the pane frame in either direction.

Scrolling Text Using the Keyboard

To scroll text in a pane with the keyboard, first make sure that the window is active. Then press the **Home** and **End** keys to scroll the text left and right, respectively, or the **PgUp** and **PgDn** keys to scroll the text up and down, respectively. To scroll in large amounts, hold down the **Shift** key while pressing one of the scrolling keys.

Scrolling to Select Large Text Areas

If the text you want to select is too long to fit in the pane at one time, there are two ways to select large areas of text. 1) Move the cursor to one end of the text string. Press and hold down the right mouse button while dragging the arrow downward as described in scrolling above. Move the cursor back into the pane when the other end of the desired text is visible. Then adjust the selection until it is just what you want before releasing the mouse button. Or 2) you can place the I-beam at one end for selection, then scroll the pane until the other end of the desired text is visible. Move the cursor over the last character to be selected. To select the text, press the shift key and click the left mouse button or press the — key on the numeric keypad. The entire section is selected.

Inserting Text

To insert text into a pane, first activate the window. Then move the cursor to the location where you want to insert text. Click the left mouse button or press the + key on the numeric keypad. Notice the I-beam in the pane. The I-beam marks the insertion point for new text.

You can insert text in two ways: either by directly typing in the new text string or by using the **paste** option from the pane menu to bring in text you have selected and copied from another place. If you make a mistake, use the **backspace** key to delete your mistakes.

Deleting Text

The easiest but slowest way to delete text is to place the I-beam after the last character you want to delete and press the **backspace** key until all the characters you wish to delete disappear from the screen. Remember that you place the I-beam by clicking the left mouse button or pressing the + key on the numeric keypad.

To delete whole words or strings of words, select the text to be deleted and either press the **backspace** key once, or begin typing new text, or **paste** in copied text from elsewhere. All three actions delete the selected text.

Zooming in on a Text Pane

When **zoom** is selected using either the **zoom button** on the window label bar or **zoom** from the **next menu** pane menu, **Smalltalk/V** zooms in on the text window so that it fills the whole screen. You can then have a full display text view for any text entry or editing you need to do. Alternatively, you can press the **F8** function key to activate the **zoom** feature.

To return the display to normal, reselect **zoom**, or press the **F8** function key again, or click the left mouse button when the cursor is inside of the window label bar.

Starting Out

Now that you are familiar with the objects and methods of the **Smalltalk/V** environment, you can jump right in and get to work. As you work your way through this last section of the chapter and the tutorials that follow, you will quickly come to understand and be able to use **Smalltalk/V**.

Evaluating Text

To evaluate a **Smalltalk/V** expression, you must first enter the text into a pane and select it. Make sure that the cursor is inside of the window, and then pop up the **pane menu**. Select either **do it** or **show it** from the menu.

When you select **do it**, **Smalltalk/V** executes the selected text. When you select **show it**, **Smalltalk/V** executes the selected text and displays the result of the expression on the screen.

For example, place the cursor into any active window pane. Type in the following expression and evaluate it:

3 + 4

When you evaluate this expression with **do it**, **Smalltalk/V** executes the expression but shows no results on the screen. When you evaluate this expression with **show it**, **Smalltalk/V** both executes the expression and shows the results. In the tutorials that follow, you will usually use the **show it** method for evaluation.

Compilation Errors

Type the following expression into a window, select it, and then evaluate it, using the **do it** choice on the **pane menu**:

Turtle

```
home;  
north;  
black;  
mandela: x2 diameter: 300
```

The result should be:

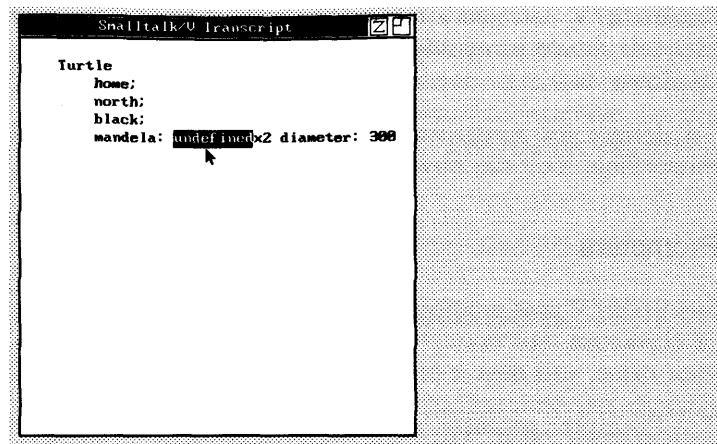


Figure 2.15
Compilation Error

The compiler detected an error. The error message is inserted in front of the error and is selected. To delete the error message, press the **backspace** key. Now correct the expression by changing the letter **x** to the number **1** so that the line reads:

mandela: 12 diameter: 300

Runtime Errors and Walkback Windows

Select the corrected expression from the above example and evaluate it. (Make sure that you select the entire expression, not just the corrected line.) The result should be:

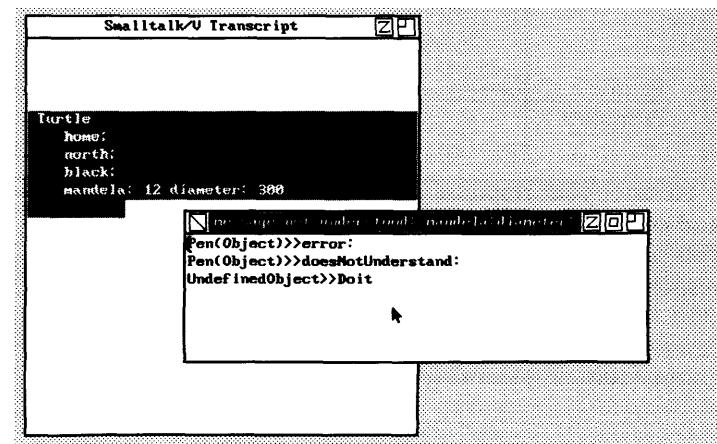


Figure 2.16
Runtime Error

Smalltalk/V detected a runtime error. In this case, the problem is a misspelled word: **mandela** should be **mandala**. The window notifying you of the error is called a walkback window.

To correct the error, simply activate the window with the expression and fix the misspelling. You can now select and evaluate the text again.

Prompters

Smalltalk/V uses prompters to ask questions to which you respond by entering a string of characters. For example, the code for this prompter shows that it asks you to enter your name, and then it prints it:

```
Prompter prompt: 'your name ?' default: ''
```

To see how this works, enter the above expression into a pane, select it, and then evaluate it using the **Show It** choice on the pane menu. The result should be:



Figure 2.17
Prompter

You enter your response in the pane of the prompter. A prompter lets you enter, delete and edit the text in the usual ways, except that it accepts only a single line of text. When you press the **return** or **enter** key, the line of text is returned to Smalltalk/V as your response and the prompter is closed.

Reusing Text

One of the nice things about Smalltalk/V is that you can edit text that you have previously evaluated, and then evaluate the expression again. For example, in the prompter example above, change 'your name?' to 'your age?'. Evaluate the expression using show it from the pane menu.

Browsing

Some windows, like the Disk Browser have several panes. Some of these panes have lists from which you can select items.

Pop up the system menu by placing the cursor on the screen background and clicking the left mouse button or using the + key on the numeric keypad. Select browse disk from the system menu. You are presented with a menu of devices. Choose one to open a Disk Browser window for that device. The pane in the upper left corner lists all of the directories on the disk. You can scroll this list both horizontally and vertically as you can with any window pane.

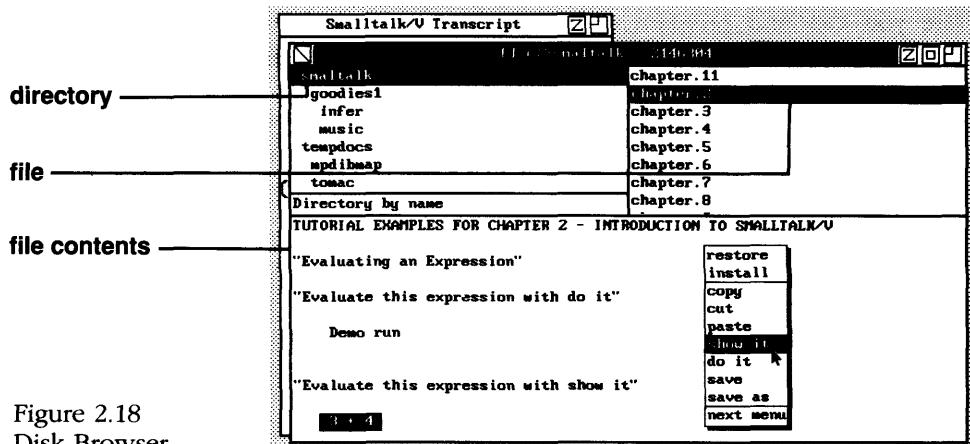


Figure 2.18
Disk Browser

To select an item from the directory list, move the cursor over the desired item and click the left mouse button or press the + key on the numeric keypad. The item is then reversed. The upper right pane then displays a list of files in the directory you just selected. If you select a file in this list, the large pane at the bottom of the window displays the file contents. Part 3 of this manual explains the Disk Browser in much more detail.

Now pop up the **system menu** by placing the cursor on the screen background and clicking the left mouse button or using the **+** key on the numeric keypad. Select **browse classes** from the system menu to open the Class Hierarchy Browser window.

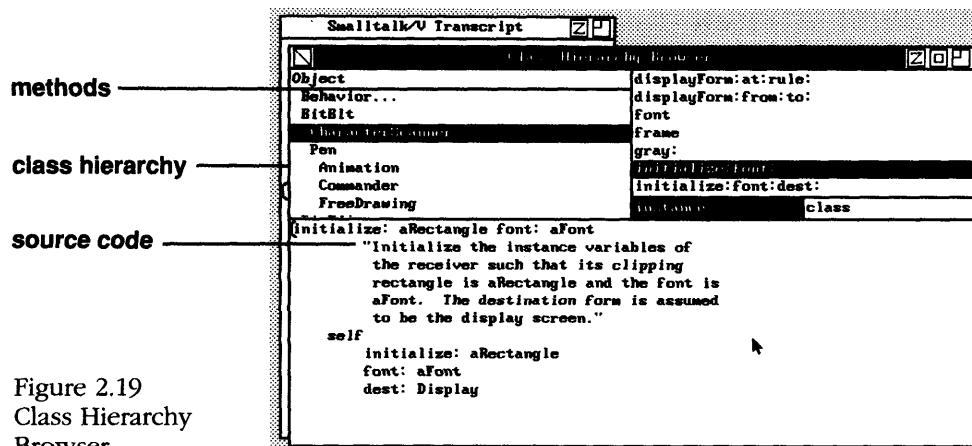


Figure 2.19
Class Hierarchy
Browser

Here you can find program code for objects you may want to use in your own Smalltalk/V program. By bringing the code into any edit or text pane, you can then select it, copy it and paste it in your own program. Once the copied code is in your own program, you can edit and evaluate it until Smalltalk/V does what you want it to do.

Tutorial Files

In the tutorials that follow, you'll be seeing many examples and programs. We've already provided these examples for you in several disk files. To save yourself the time and effort of typing in these examples, you can use the Disk Browser to access them. Open a Disk Browser for the device that contains Smalltalk/V and select the directory in which Smalltalk/V is installed.

The example files are organized by chapter. For example, the examples for Chapter 2 are in the file **Chapter.2**. You can see the examples at the beginning of the chapter displayed on the screen. To see the rest, just scroll the pane. You can then treat these examples by evaluating and editing them just as if you had entered them yourself.

You should now be familiar enough with the Smalltalk/V environment to proceed to the following tutorials. If you want to review any topics covered in this tutorial, you can either repeat the corresponding section of the tutorial or refer to a detailed description in **Part 3, The Smalltalk/V 286 Reference**.

3 OBJECTS AND MESSAGES

Now that you have toured the Smalltalk/V environment, you are ready to learn about the Smalltalk language itself. This chapter concentrates on the concepts of *objects* and *messages*, the basis of the Smalltalk language. You'll also be introduced to global and temporary variables.

Throughout the tutorials, you will be asked to evaluate sample pieces of code. As described in the previous chapter, you can find these examples stored in a disk file. Simply use the Disk Browser to access these files. The examples for this chapter are stored in the file **chapter.3**.

Even if you are an experienced Smalltalk programmer, use these examples. They will help you to understand the Smalltalk/V environment, as well as introduce you to some advanced Smalltalk applications. Of course, if you are new to Smalltalk, these examples will be even more valuable.

Simple Objects

Objects are the basic building block of the Smalltalk language. They are analogous to pieces of data in other languages. For example,

'this is a string'

is a Smalltalk object, a string of characters. It's very much like a string in any other language. Here are some other simple Smalltalk objects that have counterparts in most languages:

1234	"an integer"
\$A	"the single character A"
#(1 2 3)	"an array of three integer objects"

Look at the last example, the array. It's an object which itself contains other objects. Let's look at some more examples:

**#('array' 'of' 'four' 'strings')
#('array' 'of' 5 'strings' 'and' 2 'integers')**

As you can see from the last example, all of the objects contained inside of an object do not have to be of the same type or size. Part of the power of Smalltalk comes from this capability. Consider this more complex object:

#(1 ('two' 'three') 4)

This is an object (an array) with three objects inside of it. The second object in the array is another array of two strings.

Simple Messages

Of course, an object can do nothing by itself. In Smalltalk, you send *messages* to objects to make things happen. Messages are similar to function calls in other languages. For example, look at this Smalltalk expression, composed of a single message:

20 factorial

This sends the message **factorial** to the object **20**. To evaluate this expression, either find it in the tutorial file **chapter.3** or type it into a pane and select it. Then use the **show it** choice on the pane menu to compile and evaluate it. The result should be a very large integer:

2432902008176640000

Let's try another simple message. When you select and evaluate the following expression, the result should be the integer 15, the size of the string:

'now is the time' size

A message is composed of three parts: a *receiver object*, a *message selector*, and zero or more *arguments*. In the above example, the string is the receiver object, the message selector is **size**, and there are no arguments. Or consider:

#(1 3 5 7) at: 2

In this example, the array is the receiver, **at:** is the message selector, and the **2** is the argument.

A message always returns a single object as its result, just like functions in most other languages. Similarly, the message selector is like the function name, and the receiver object is like the first function parameter. The above example asks for the second element of the array; the result is the integer 3.

Now try evaluating this example:

'20' factorial

A walkback window appears. Since **20** is enclosed in quotes, it is a string, to which **factorial** makes no sense. As this example illustrates, *objects always know the messages that are appropriate for them*. Part 4 of this manual lists all of the different kinds of objects provided by **Smalltalk/V**, and the messages that they respond to.

To get rid of the walkback window, select the **close** button from the walkback window label bar by positioning the cursor over the button and clicking the left mouse button or pressing the numeric keypad + key.

Unary Messages

Messages with no arguments are called *unary* messages. Try evaluating these unary messages:

```
#( 'array' 'of' 'strings' ) size
'now is the time' asUpperCase
'hello there' reversed
#( 4 'five' 6 7 ) reversed
$A asciiValue
65 asCharacter
```

Keyword Messages

Messages with one or more arguments are called *keyword* messages. Try evaluating these keyword messages:

```
'now is the time' at: 6
'Hello' includes: $e
'hello' at: 1 put: $H
'The quick brown' copyFrom: 4 to: 9
```

In the last two examples, the message selectors `at:put:` and `copyFrom:to:` are divided up by the arguments. In these examples, `at:put:` and `copyFrom:to:` work with strings, but these same messages work for arrays as well:

```
#( 9 8 7 6 5 ) at: 3
#( 1 ( 2 3 ) 4 5 ) includes: #( 2 3 )
#( 1 0 4 5 ) at: 2 put: #( 2 3 )
#( 9 8 7 6 5 ) copyFrom: 1 to: 2
```

From these examples, you may have noticed another important point: different kinds of objects can respond to the same message in different ways. These arrays respond differently to the `at:put:` and `copyFrom:to:` message selectors than the strings in the previous examples.

Arithmetic Messages

Smalltalk arithmetic looks the same as in most other languages. For example:

`3 + 4`

But, like other Smalltalk expressions, this is a message. The integer 3 is the receiver, `+` is the message selector, the integer 4 is the argument, and the integer 7 is the result. Here are some more common arithmetic messages you can evaluate.

`5 * 7`

"multiplication"

`5 // 2`

"integer division (truncation)"

`4 \\\ 3`

"integer remainder"

`2 / 6`

"rational division"

The last expression illustrates *rational arithmetic*, or the arithmetic of fractions. In Smalltalk, rational arithmetic is exact; there is no rounding or truncation. This is because Smalltalk stores the result as a numerator and a denominator (reduced to its simplest form), rather than computing an approximate real number.

Arithmetic expressions in Smalltalk also differ from most other languages in their order of evaluation. Smalltalk evaluates an arithmetic expression strictly from left to right, with no precedence among operators. For example, evaluate this expression:

`3 + 4 * 2`

The result is 14, not 11. You can, however, use parentheses to control evaluation order, as in this example:

`3 + (4 * 2)`

If your computer has a coprocessor, you can perform floating point arithmetic as well:

`1.5 + 6.3e2`

If a walkback window appears, your computer does not have a coprocessor. Close the walkback window and continue.

Binary Messages

Arithmetic messages are examples of *binary* messages, messages with one argument and one or two special characters (other than digits and letters) as the selector. Binary messages are always evaluated strictly from left to right, unless you have used parentheses. For example, evaluate these non-arithmetic messages:

```
'hello', ' there'
#(1 2 3), #(4 5 6)
```

In these examples, the special character is the comma. It concatenates the argument with the receiver object.

Messages inside of Messages

As we stated before, messages are like functions, in that they return an object. This means that anywhere an object appears in an expression, you can use a message which returns a similar kind of object. For example, evaluate:

```
'hello' size + 4
'now' size + #(1 2 3 4) size
#(1 12 24 36) includes: 4 factorial
```

The last expression above is really two messages. 4 factorial is a unary message, and is computed first. The result then becomes the argument for the includes: message. Now evaluate this more complex expression:

```
4 factorial between: 3 + 4 and: 'hello' size * 7
```

This expression is composed of five messages. The five message selectors are factorial, +, size, *, and between:and:. As you can see, unary messages are always evaluated before binary messages, which in turn are evaluated before keyword messages. As usual, you can override this precedence by using parentheses:

```
'hello' at: (#(5 3 1) at: 2)
```

This expression is composed of two at: messages: one to an array, and one to a string. To see what happens when there are no parentheses, try:

```
'hello' at: #(5 3 1) at: 2
```

This expression is a single message with two arguments. The message selector becomes at:at:, clearly not what we had in mind.

Expression Series

You can't do much with just a single expression. Here's a series of expressions, which you can evaluate as a single unit. Select the entire series before popping up the pane menu:

```
Turtle black.  
Turtle home.  
Turtle go: 100.  
Turtle turn: 120.  
Turtle go: 100.  
Turtle turn: 120.  
Turtle go: 100.  
Turtle turn: 120
```

Each message in the series is separated from the next by a period. We put each message on a separate line purely for appearance. The receiver of all the messages is the object **Turtle**, one of the objects supplied in the Smalltalk/V environment. To get a different picture, change the word **black** to **white** and evaluate the expression series again.

To clean up the screen, select **redraw screen** from the system menu.

Cascaded Messages

A *cascaded* message is a shorthand way of writing a series of messages that are sent to the same receiver. For example, the following expression draws the same figure as the previous example (only in a different color):

```
Turtle  
  white;  
  home;  
  go: 100;  
  turn: 120;  
  go: 100;  
  turn: 120;  
  go: 100;  
  turn: 120
```

The receiver is written only once, and each message (except for the last) is terminated with a semi-colon instead of a period. The indentation, again, is optional; it simply makes the code easier to read.

Simple Loops

We can simplify the above example by using a message that loops a specified number of times:

```

Turtle
  black;
  home.
3 timesRepeat: [ Turtle go: 100; turn: 120 ]

```

In this example, the argument to the **timesRepeat:** message is a block of code. Blocks of code are written as a series of messages enclosed in square brackets, [and]. We'd normally write the above example as:

```

Turtle
  black;
  home.
3 timesRepeat: [
  Turtle
    go: 100;
    turn: 120 ]

```

This makes the cascaded message inside the block easier to see.

Objects and Messages Are Safe

The previous series of expressions illustrates another point about objects and messages. Objects have a state; they can remember things. Messages change an object's state. The **Turtle** object remembers its position, heading, and color. The messages **black** and **white** change its color, the messages **go:** and **home** change its position, and the message **turn:** changes its heading. Let's look at another expression that emphasizes this point:

```
'hello' at: 1 put: 23
```

When you evaluate this expression, a walkback error window pops up because 23 is not a character. Since you can only change the state of the string by sending messages, the string can check the validity of the arguments. This makes Smalltalk a very safe language.

Temporary Variables

Temporary variables are so called because Smalltalk discards them as soon as you are done using them. Temporary variables are declared by enclosing them in vertical bars in the first line of an expression series. Temporary variable names must start with a lower case letter, while the rest of the name can be any combination of upper and lower case letters and digits. For example, look at this short program that uses three temporary variables and a loop to compute an array of several factorials:

```

| temp index factorials |
factorials := #( 3 4 5 6 ).
index := 1.
factorials size timesRepeat: [
    temp := factorials at: index.
    factorials at: index put: temp factorial.
    index := index + 1].
^factorials

```

The first line declares three temporary variables: **temp**, **index**, and **factorials**. A temporary variable can hold any type of object. To give it a value, you use an *assignment expression*.

Assignment Expressions

The above example uses four assignment expressions:

```

factorials := #( 3 4 5 6 ).
index := 1.
temp := factorials at: index.
index := index + 1.

```

The first two assign objects to the temporary variables, while the last two assign the results of messages. Since the result of a message is always a single object, they actually assign objects to the temporary variables.

Return Expressions

The last expression in the factorial example above is:

^factorials

The caret (**^**) indicates that this is the value to be returned as the result of the expression series. Such a statement beginning with a caret is called a *return expression*.

Global Variables

Smalltalk/V has many objects built into it, many of which are contained in *global variables*. Unlike temporary variables, Smalltalk does not automatically dispose of global variables when you are finished using them, and their use is not confined to a single set of expressions. For example, Smalltalk/V provides (among others) these three global variables:

Turtle
Transcript
Disk

We have been using the global variable, **Turtle**. Global variables always contain a single object. For example, select **Turtle**, and then use the **show it** choice on the pane menu to see its current contents. The result is itself a single object.

Global variable names always begin with an upper case letter, with the remainder upper and lower case letters and digits. Type in the following, select it, and show it:

Sammy

When the global variable does not currently exist, you get a menu which allows you to choose whether or not you want to create it. If you do not create the global variable, **Smalltalk/V** assumes you made an error, and displays an error message. This keeps you from accidentally creating global variables when you misspell something. As with temporary variables, you use assignment statements to assign values to global variables:

Sammy := 'Sammy Jones'

Putting It All Together

To conclude this first tutorial, here's a graphical program that draws flowers composed of several polygons.

```
"Draw a polygon flower"
| sides length |
sides := 5.
length := 240 // sides.
Turtle
    black;
    home;
    north.
sides timesRepeat: [
    Turtle go: length.
    sides - 1 timesRepeat: [
        Turtle
            turn: 360 // sides;
            go: length]]
```

The first line is a *comment*. Comments are any string of characters enclosed in double quote marks ("comment"). **Smalltalk/V** ignores comments when it compiles the program; they simply add clarity to the code. Comments can appear anywhere in an expression series.

Evaluate the above example, and note the results. For a slightly different flower, change the number of sides. To make the polygons spread further apart, change:

Turtle go: length

to the following expression (you might want to change the color to **white** as well):

Turtle

```
up;
go: length // 2;
down;
go: length.
```

which produces:

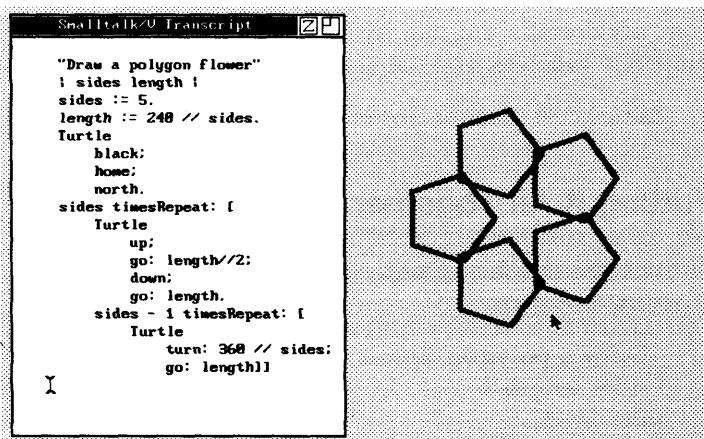


Figure 3.1
Polygon Flower

What You've Now Learned

At this point, you should be familiar with:

- simple objects
- simple messages
- unary, keyword, and binary (including arithmetic) messages
- messages inside of other messages
- message series
- cascaded messages
- temporary and global variables
- assignment expressions
- return expressions
- comments

If you want to review any of these topics, simply refer back to the appropriate section in this chapter. Of course, Part 3 describes all of these topics in greater detail.

Here's one final point which you can think about as you proceed on to the following tutorials. As we mentioned, **Turtle**, which you have been using throughout this tutorial, is a global variable. You have been able to use this variable in a number of different situations, without knowing anything about its internal contents. This is a unique feature of Smalltalk; you can use an object simply by knowing its *external behavior*, without knowing anything of its *internal behavior*. You'll see further examples of this throughout the following tutorials.

Now that you've learned the above topics, you can proceed on to the next tutorial: control structures.



Digitized by srujanika@gmail.com

4 CONTROL STRUCTURES

In the previous tutorial, you learned some of Smalltalk's basic expressions. But like any language, Smalltalk cannot do much unless it can make decisions: evaluate a condition and perform an action based on the result, or repeat actions a specified or unspecified number of times. This chapter introduces you to Smalltalk's *conditional expressions* and *control structures*, which perform these tasks.

As always, you can access the examples for this tutorial if you do not want to type them in. Simply use the Disk Browser to retrieve the contents of the file `chapter.4`.

Comparing Objects

Smalltalk compares objects by sending messages. The normal comparisons of `<`, `<=`, `=`, `>=`, `>`, and `~` are implemented as binary messages. For example, evaluate these expressions:

```
3 < 4  
#( 1 2 3 4 ) = #(1 2 3 4)
```

All objects understand equality, `=`. Many objects also define the relational operators, or *ordering messages*, as in these examples:

```
'hello' <= 'goodbye'
```

Since these comparison messages are binary messages, parentheses are often needed if there are other binary messages in the expression. For example, evaluate the following expression with and without parentheses:

```
5 = (2 + 3)
```

As you have seen from evaluating these examples, comparisons return either `true` or `false`.

Testing Objects

Many objects understand messages that let you test something about their state or condition. For example, evaluate these expressions:

```
$a isUpperCase  
('hello' at: 1) isVowel  
7 odd
```

These messages return `true` or `false` as well.

Conditional Execution

Most languages use if statements to conditionally execute a series of statements. Smalltalk uses blocks of code and messages to do the same thing. For example, look at this expression, which computes the greater of two numbers:

```
| max a b |
a := 5 squared.
b := 4 factorial.
a < b
    ifTrue: [max := b]
    ifFalse: [max := a].
^max
```

The comparison message `a < b` returns either true or false, which then becomes the receiver of the `ifTrue:ifFalse:` message. It, in turn, executes the corresponding block of code.

As you have seen, all messages return a result. The `ifTrue:ifFalse:` message, like any message, also returns a result. Evaluate the following expression:

```
3 < 4
    ifTrue: ['the true block']
    ifFalse: ['the false block']
```

The message `ifTrue:ifFalse:` returns the result of the last expression in the block that it executes. Let's look at another example:

```
| string index c |
string := 'Now is the time'.
index := 1.
string size timesRepeat: [
    c := string at: index.
    string
        at: index
        put:
            (c isVowel
                ifTrue: [c asUpperCase]
                ifFalse: [c asLowerCase]).
    index := index + 1].
^string
```

The above example converts all consonants to lower case and all vowels to upper case, using the `c isVowel` test on each letter to find out which to do.

The other messages that execute a block of code conditionally are **ifTrue:**, **ifFalse:**, and **ifFalse:ifTrue:**.

Boolean Expressions

The examples so far have depended on single comparison or testing messages, such as **<** or **isVowel**. But often you need to perform a compound test. To do so, you use the **and:** and **or:** messages. For example, look at the following code fragment, which tests whether a character is not a digit:

```
( c < $0 or: [ c > $9 ] )
```

The receiver of the **or:** message is the result of the first comparison, either **true** or **false**. The argument is a block of code whose last expression also returns **true** or **false**. The **and:** message works in the same way:

```
( c >= $0 and: [ c <= $9 ] )
```

To see how this is used in a complete example, evaluate this expression:

```
"compute the value of the first integer in a string"
| string index answer c |
string := '1234 is the number'.
answer := 0.
index := 1.
string size timesRepeat: [
    c := string at: index.
    (c < $0 or: [c > $9])
        ifTrue: [ ^answer].
    answer := answer * 10
        + c asciiValue - $0 asciiValue.
    index := index + 1].
^answer
```

Notice the return expression **ifTrue: [^answer]** in the middle of the above example. This exits the expression as soon as the first non digit is encountered.

You can perform more complex tests by nesting the expressions. For example, look at the following fragment, which tests if a character is a digit or one of the letters from A-F.

```
( c isDigit or: [ c >= $A and: [ c <= $F ] ] )
```

Looping Messages

You've already seen one simple looping message, `timesRepeat:`. Here's a simple expression that uses another simple looping message to copy a file:

```
"copy a disk file"
| input output |
input := File pathName: 'go'.
output := File pathName: 'junk'.
[input atEnd]
    whileFalse: [output nextPut: input next].
input close.
output close
```

You may have noticed the message `atEnd` in the above example. This message returns `true` when there are no more characters to read from the input file stream; otherwise, it returns `false`. The input file stream is read with the `next` message, which returns the next character in the file. The output file stream is written with the `nextPut:` message, which writes its argument to the output.

The message `whileFalse:` is sent to a block of code with another block of code as an argument. The message repeatedly evaluates its receiver block for as long as the argument is `false`. When the receiver block evaluates to `true`, the expression closes the input and output files.

As you might expect, Smalltalk also provides a corresponding `whileTrue:` message. To see it, look at this graphical example, which uses Turtle to draw some polygons:

```
"draw several polygons"
| sides |
sides := 3.
[sides <= 6]
    whileTrue: [
        sides timesRepeat: [
            Turtle
                go: 60;
                turn: 360 // sides].
        sides := sides + 1]
```

Simple Iterators

The Turtle example above increases the temporary variable sides from 3 to 6 by 1, and evaluates some code for each value along the way. Like most other languages, Smalltalk provides iteration statements to do this more easily. For example, here's the same expression, using one such iteration statement:

```
"draw several polygons"
3 to: 6 do: [ :sides |
  sides timesRepeat: [
    Turtle
      go: 60;
      turn: 360 // sides]]
```

The iteration message is `to:do:`, which has two arguments. It takes the receiver object, 3, as the lower limit of the iteration, and uses the first argument, 6, as the upper limit. The second argument is a block of code, which itself uses a block argument, `sides`, to draw a polygon with `sides` number of sides. The iteration message assigns the values 3 thru 6 successively to the block argument, evaluating the block for each value.

The `to:do:` message uses an increment of one. But you can also specify your own increment by using the `to:by:do:` message, as in this example:

```
"compute the sum of 1/2, 5/8, 3/4, 7/8, 1"
| sum |
sum := 0.
1/2 to: 1 by: 1/8 do: [ :i |
  sum := sum + i].
^sum
```

The first argument, 1, is the upper limit, while the second argument, `1/8`, is the increment.

Block Arguments

As you can see from the last example, a block argument is declared in the first part of the block, preceded by a colon, `:`, and separated from the statements in the block by a vertical bar, `|`. For example, here's a block with one argument:

```
[ :character | character isVowel ]
```

In this example, the block argument is `character`. Block arguments are a kind of temporary variable, but do not have to be declared at the beginning of the expression series.

Generalized Iterators

Blocks with arguments allow Smalltalk to supply several generalized iteration messages: **do:**, **select:**, **reject:**, and **collect:**.

The do: iterator

The simplest of these is the **do:** message:

```
"count vowels in a string"
| vowels |
vowels := 0.
'Now is the time' do: [ :char |
    char isVowel
        ifTrue: [vowels := vowels + 1]].
^vowels
```

The **do:** iterator causes the string to iterate across itself and pass each character to the block. The above example is equivalent to the following:

```
"count vowels in a string"
| vowels string index |
vowels := 0.
index := 1.
string := 'Now is the time'.
[index <= string size]
    whileTrue: [
        (string at: index) isVowel
            ifTrue: [vowels := vowels + 1].
        index := index + 1].
^vowels
```

The **do:** message can also iterate arrays, as in this example:

```
"draw several polygons"
#( 3 4 12 24 ) do: [ :sides |
    sides timesRepeat: [
        Turtle
            go: 60;
            turn: 360 // sides]]
```

or file streams:

"Strip all carriage return characters (ascii 13) from a disk file. Answer the number of characters stripped."

```
| output stripped |
stripped := 0.
output := File pathName: 'stripped.go'.
(File pathName: 'go') do: [ :char |
    char = 13 asCharacter
        ifTrue: [stripped := stripped + 1]
        ifFalse: [output nextPut: char]]..
output close.
^stripped
```

The above expression reformats a DOS text file for use with Unix, replacing DOS's carriage return/line feed pair at the end of lines with Unix's single line feed.

The **select:** Iterator

A more powerful iterator is the **select:** message:

```
"count the vowels in a string"
('Now is the time' select: [ :c | c isVowel ] )  
size
```

The **select:** message iterates across its receiver and returns all of the elements for which the argument block evaluates to true. In this case, the result is a string of all of the vowels in the original string. The message **size** then tells us how many elements were selected.

The **reject:** Iterator

The **reject:** message is another generalized iterator:

```
"answer all digits whose factorial is
less than the digit raised to the 4th power"
#( 1 2 3 4 5 6 7 8 9 ) reject: [ :i |
    i factorial >= ( i * i * i * i ) ]
```

The **reject:** message works just as **select:**, but answers all elements of the receiver for which the block of code returns false, instead of true.

The **collect:** iterator

The **collect:** message evaluates the block of code for each element of the receiver and answers the collection of all of the results returned by the block:

```
"square each element in the array"
#(1 13 7 10) collect: [:i| i * i]
```

To help see the differences between **select:**, **reject:**, and **collect:**, evaluate the following expressions:

```
#(1 2 3 4 5 6 7) select: [ :c | c odd ]
#(1 2 3 4 5 6 7) reject: [ :c | c odd ]
#(1 2 3 4 5 6 7) collect: [ :c | c odd ]
```

Concluding Example

Our concluding example is inspired by the limitations of DOS file names. DOS limits file names to eight characters with a three character extension. Often, you need to abbreviate long names that appear inside of programs. A good algorithm is to remove lower case vowels from the original name from right to left. If this doesn't shorten it enough, you might truncate what's left to eight characters. With names already shorter than eight characters, you might want to pad the name with blanks, and not throw out any characters. Here, then, is a Smalltalk solution:

```
"abbreviate a long file name to 8 characters"
| name length |
name := 'LongFileName'.
length := name size.
^( name reversed reject: [ :c |
  c isVowel and: [
    c isLowerCase and: [
      ( length := length - 1 ) >= 8 ] ] )
  reversed, '
  copyFrom: 1 to: 8
```

Let's examine this example in detail. The caret (`^`) on the fourth line tells us that the remainder of the program will return a single result. We reverse the name so that we can throw out characters from the end of the original name first. Similarly, we reverse the result of the **reject:** message to put the abbreviated name back in the proper order. We then append blanks to the resulting string and return the first eight characters as the answer.

Look more closely at the expression inside of the argument block to the **reject:** message:

```
[ :c |  
  c isVowel and: [  
    c isLowerCase and: [  
      ( length := length - 1 ) >= 8 ] ] ]
```

Remember that the `reject:` message eliminates only those characters for which this block evaluates to true. It's easy to see why the first two tests are `isVowel` and `isLowerCase`, since they are the possible characters to eliminate. The final test is more complex:

```
( length := length - 1 ) >= 8
```

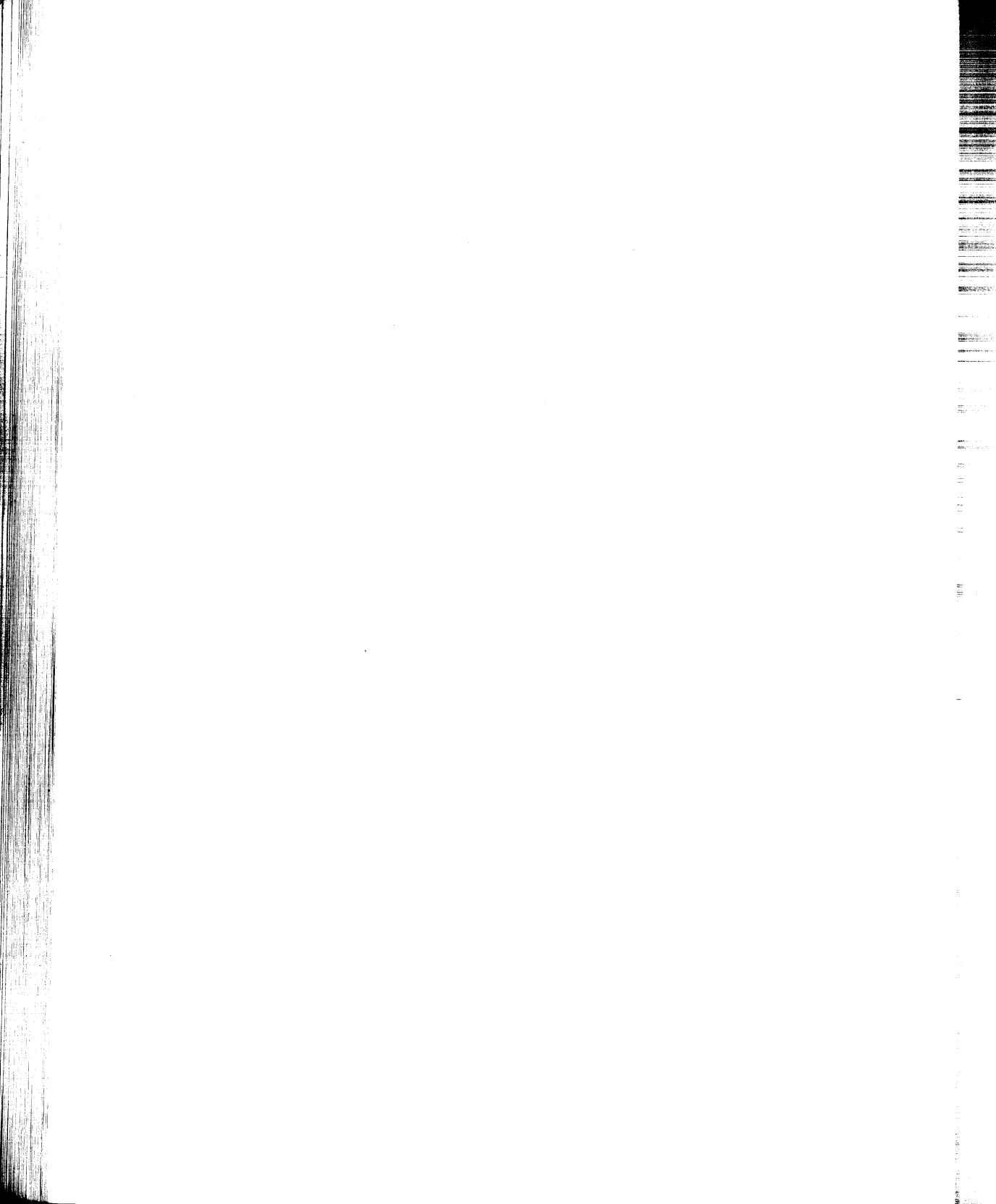
This expression must evaluate to true to delete the character, and false to keep it. The expression decrements the temporary variable `length`. If the `length` is less than 8, the character is to be kept; otherwise it is eliminated. Since we initially set `length` to the size of the string `name`, this expression of code returns true at most `name size - 8` times, which is the number of characters we want to eliminate.

What You've Now Learned

After finishing this chapter, you should be familiar with:

- comparing and testing objects
- conditional statements
- boolean expressions
- simple loops
- simple iterators
- block arguments
- the generalized iterators, `do:`, `select:`, `reject:`, and `collect:`

If you want to review any of these topics, you can either repeat the corresponding section of the tutorial, or refer to a detailed explanation in **Part 3, The Smalltalk/V 286 Reference**.



5 CLASSES AND METHODS

In the preceding chapters, you have learned Smalltalk versions of techniques which are common to most programming languages. For example, you have learned how to form basic expressions, and how to use loops and conditional statements. In this chapter, you will learn some of the concepts that make Smalltalk unique: **class** and **method**. You will examine some **Smalltalk/V** classes and methods, and add new methods to these classes for numeric processing, pattern matching, and graphics. You'll also learn how to use **Inspectors**, windows for viewing and changing the internal variables that define the state of objects.

Beginning with this tutorial, you will make changes to the **Smalltalk/V** environment itself. In order to make these changes permanent (so that you can use them in later tutorials), be sure to *save the image* whenever you leave **Smalltalk/V**. That is, when you select **exit Smalltalk** from the system menu, be sure to then use the **save image** function.

As always, you can find the examples for this tutorial in the disk file **chapter.5**. Use the Disk Browser to retrieve them, if you wish.

Classes

Problem solving using Smalltalk involves *classifying* objects according to their similarities and differences. You've already seen the external behavior of objects, by sending messages to them and observing the results. A **class** defines the behavior of similar objects by specifying their insides: the variables they contain and the methods available for responding to messages sent to them.

Every object is an *instance* (member) of a class. For example, `#(1 2 3)` and `#(sam joe)` are instances of class **Array**, whereas 'north' and 'south' are instances of class **String**. All objects know which class they belong to. For example, evaluate the following expressions:

```
#(Francesca Jackie Marisa) class  
'Rakesh Vijay' class  
Turtle class
```

An object's internal variables are called *instance variables*; they are themselves containers for other objects. For example, objects in class **Fraction** have instance variables **numerator** and **denominator**. For the object representing the fraction $1/7$, the instance variable **numerator** contains the object 1 and the instance variable **denominator** contains the object 7.

Methods

Methods are Smalltalk code, the algorithms that determine an object's behavior and performance. They are like function definitions in other languages. When a message is sent to an object, a method is evaluated, and an object returned as a result. Evaluate the following message expression:

(1/7) numerator

When the message **numerator** is sent to the fraction $1/7$, Smalltalk evaluates the method **numerator** defined in class **Fraction**:

```
numerator
^numerator
```

The first line of the method defines the method name. (Notice that it matches the selector in the corresponding message.) The second line returns the result **numerator**, the instance variable of the receiver fraction object. As a more complex example, evaluate the following message expression:

(2/3) * (5/7)

Sending the message ***** to the fraction $2/3$ with the fraction $5/7$ as the argument evaluates the method ***** in class **Fraction**:

```
* aNumber
^(numerator * aNumber numerator) /
(denominator * aNumber denominator)
```

The first line defines the method name (*****) and the name for the argument, **aNumber**, which is used in the rest of the method to represent the argument object. The method returns a new fraction whose numerator is the product of the receiver and argument numerators, and whose denominator is the product of the receiver and argument denominators.

Notice that **numerator** and **denominator** appear both as instance variables and messages in this method. In processing the example message, the argument **aNumber** contains the fraction $5/7$, while the instance variables **numerator** and **denominator** contain 2 and 3 respectively.

As you can see from this example, Smalltalk objects are abstract data types. The multiply method operates on behalf of the receiver object ($2/3$), whose internal variables **numerator** and **denominator** are accessible. The argument is another object ($5/7$). Even though it is the same class as the receiver, its internal variables are not available in this method, and so messages must be used to request the desired information. This Smalltalk feature provides complete safety from outside manipulation.

The Class Hierarchy Browser

In using the Smalltalk/V environment up to this point, you may be wondering where you do your actual programming. To program in Smalltalk/V, you use a special window called the *Class Hierarchy Browser*. It lets you browse and change existing class and method definitions, and create new ones. Open a Class Hierarchy Browser window now by evaluating the following expression:

```
ClassHierarchyBrowser new openOn: (Array with: Integer
with: Fraction with: String with: DemoClass)
```

A new window will appear on the screen. Remember that you can resize or move this window as you learned in Chapter 2. A Class Hierarchy Browser window is now available for the classes **Integer**, **Fraction**, **String** and **DemoClass**, as you can see from the top left pane. Select the entry for class **Fraction**; you'll see the following window:

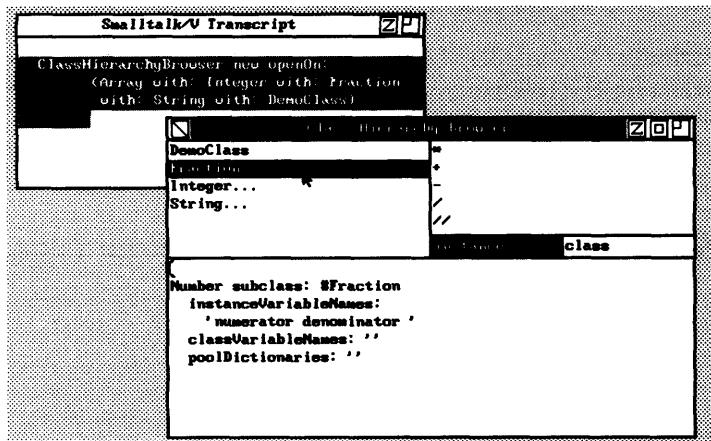


Figure 5.1
Class Hierarchy
Browser

The top right pane shows the methods defined for class **Fraction**, the selected class. The bottom pane shows the *class definition message* for class **Fraction**. The class definition message shows the characteristics that make up a class. Notice the **instanceVariableNames:** argument; it's a string specifying that the instance variable names are **numerator** and **denominator**. You'll learn about this message's other arguments later in the chapter.

Select the method ***** in the top right pane; the source code for the method appears in the bottom pane. This pane is a text editor which you can use to change existing methods and create new ones. Try selecting other methods, and look at the source code.

The Special Variable "self"

Now let's add the following new method to class **Fraction**:

fraction

"Answer the receiver minus its integral part."
^self - self truncated

This method returns a fraction less than one, the receiver of the message minus the integral part of the receiver. The method contains the word **self**, a special variable representing the object which is the receiver of the **fraction** message. Add the method to class **Fraction** using the following steps:

- Pop up the pane menu in the top right pane and select **new method**.
- You'll see a prototype method in the bottom pane. Replace it with the source code for the **fraction** method defined above.
- Pop up the pane menu in the bottom pane and select **save**.

Smalltalk/V compiles the new method and installs it in class **Fraction**. Try it out by evaluating the following messages:

(22/7) **fraction**
(2/3) **fraction**

Creating New Objects and the Special Object "nil"

You have seen several messages which create new objects, such as:

'bigger', 'string'
1 / 3

Classes are also objects, and so can be used in message expressions. A common way to create a new object is to send a message to its class. For example, evaluate the following messages:

Array **new: 10**
Array **new**
Pen **new**
Date **today**
Time **now**

The first message creates an array with 10 elements, all initialized to the object **nil**. The object **nil** is the sole instance of class **UndefinedObject**; it is assigned to the instance variables of all new objects. This means that unless an object assigns a value to its instance variables, they contain **nil**. The second message, on the other hand, creates an array with no elements at all.

The third message creates a pen, an instance of class **Pen**; if you show it, it displays itself as "a Pen". This is the default way for an object to display itself. (In Chapter 7, you'll learn how to include more information when an instance displays itself.)

The final two messages create an instance of class **Date** (representing the current date), and an instance of class **Time** (representing the current time), respectively.

Instance Variables

Objects can contain both *named* and *indexed* instance variables. Named instance variables are accessed by name, as with **numerator** for fraction objects. Indexed instance variables are identified by integers beginning with 1. They are always accessed via messages, such as:

```
'location' at: 2
'parts' at: 5 put: $y
```

An object's class specifies the named instance variables, and whether or not indexed instance variables can be used in its instances. The number of named instance variables is fixed for all instances of the class. The number of indexed instance variables is defined when you create the object, and may differ among instances of a class. For example, the two strings above have eight and five indexed instance variables, respectively.

For a complete description of how to specify class information, refer to Part 3, **The Smalltalk/V 286 Reference**.

Recursion

A powerful programming technique is *recursion*. Recursion is often used when an algorithm or data structure is defined in terms of itself. In Chapter 3, you saw examples using the factorial message. Let's look at the factorial *method*, defined in class **Integer**:

```
factorial
    "Answer the factorial of the receiver."
    self >1
        ifTrue: [^(self - 1) factorial * self].
    self <0
        ifTrue: [^self error: 'negative factorial'].
    ^1
```

The **factorial** method multiplies the receiver by the factorial of the quantity, receiver minus one. If the receiver is less than or equal to one, the answer is one. As in this example, a recursive solution is often a straightforward translation of a mathematical definition into a Smalltalk method.

Evaluate the following expression, which sends the `factorial` message to each of the elements of an array and returns the answers in a new array:

```
#(0 1 2 3 4 10 15 20) collect: [ :n | n factorial ]
```

As another example of recursion using integers, consider how to compute Fibonacci numbers. A Fibonacci number is a statistical function used in many applications. The nth Fibonacci number for n greater than 2 is defined to be the sum of the Fibonacci numbers for n - 1 and n - 2. (The Fibonacci number for n less than 3 is one.) Use the Class Hierarchy Browser to add the following method to class `Integer` (Note that you can copy the method from the tutorial file `chapter.5` and paste it over the `new method` template in the bottom pane of the Class Hierarchy Browser):

```
fibonacci
    "Answer the nth fibonacci number,
     where n is the receiver."
    ^self <3
        ifTrue: [1]
        ifFalse:
            (self - 1) fibonacci + (self - 2) fibonacci]
```

Notice that the `fibonacci` method returns its result differently from the `factorial` method. Instead of using the caret (^) in multiple places, as in `factorial`, `fibonacci` uses a single caret to return the result of the `ifTrue:ifFalse:` message. That result is the result of either of the two blocks, depending on the value of the <message. Test the `fibonacci` method by evaluating the following message:

```
#(1 2 3 4 5 6 7 10 20) collect: [ :m | m fibonacci ]
```

Pattern Matching

The following example illustrates simple pattern matching applied to strings. (Later, you'll see how Smalltalk's inheritance allows this same method to do pattern matching for several other classes as well.) Add the method `indexOfString:` to class `String` using the Class Hierarchy Browser:

```

indexOfString: aString
    "Answer the index position of the first occurrence
     of aString in the receiver. If no such element
     is found, answer zero."
| index1 index2 limit1 limit2 |
limit2 := aString size.
limit1 := self size - limit2 + 1.
index1 := 1.
[index1 <= limit1]
whileTrue: [
    index2 := 1.
    [index2 <= limit2
        and: [(self at: index1 + index2 - 1)
            = (aString at: index2)]]
        whileTrue: [index2 := index2 + 1].
    index2 > limit2
        ifTrue: [^index1].
    index1 := index1 + 1].

```

^0

This method starts at the beginning of the receiver string and searches for the first occurrence of the argument string. It returns either the index of the first character in the receiver's matching substring, or 0 if there are no matches. The method contains two nested **whileTrue:** loops. The outer loop proceeds through the characters of the receiver. Beginning at each character reached in the outer loop, the inner loop compares the characters of the argument to corresponding characters of the receiver.

Test the **indexOfString:** method by evaluating each of the following messages:

```

'abcdebcde' indexOfString: 'ebg'
'abcdebcde' indexOfString: 'bcd'
'abcdebcde' indexOfString: 'c'
'abcdebcde' indexOfString: 'abcdebcde'
'abcdebcde' indexOfString: ""

```

Adding a Method to a Graphics Program

In Chapter 3 you used a series of expressions to draw a polygon flower on your display. The next example packages those expressions into a method, and extends the graphics demo program so that you can choose the polygon flower from the demo program's menu.

To create the new method, add the following to class **DemoClass** using the Class Hierarchy Browser:

```

polyFlower
    "Draw a polygon flower"
    | sides length |
    Display white: rectangle.
    sides := Prompter
        prompt: 'Number of sides?'
        defaultExpression: '30'.
    length := 240 // sides.
    pen
        home;
        north.
    sides timesRepeat: [
        pen
            up;
            go: length // 2;
            down;
            go: length.
    sides - 1 timesRepeat: [
        pen
            turn: 360 // sides;
            go: length]]

```

The **polyFlower** method differs from the polygon flower expressions used earlier in two ways. Firstly, this method uses a prompter window, rather than a constant, to determine the number of sides. Secondly, this method uses the instance variable **pen**, rather than the global variable **Turtle**, to draw the polygon flower.

To add "poly flower" as a choice in the graphics demo menu, extend the **demoMenu** method in class **DemoClass**, as follows:

```

demoMenu
    ^Menu
    labels: ('exit\poly flower\walking line\' ,
              'dragon\mandala\mandalas\pentagons\' ,
              'spirals\ellipses\bouncing ball') withCrs
    lines: #(1 5 8)
    selectors: #(exit polyFlower walkLine dragon mandala
                  multiMandala multiPentagon multiSpiral multiEllipse bounceBall)

```

Replace the **demoMenu** method in class **DemoClass** with the version above. Now try the extended demo program by selecting **run demo** from the system menu.

Class Variables

Class variables are global variables accessible to all instances of a class. They are used to share data within a class. Class variables begin with a capital letter.

Let's add a class variable to **DemoClass** to count the number of times we perform the mandala graphics demo method. First, define the new class variable using the Class Hierarchy Browser. Select **DemoClass**, then edit the class definition to have the name **MandalaCount** following **VariableCount** in the **classVariableNames:** argument. Then pop up the pane menu and select **save**. This creates the class variable and recompiles **DemoClass**. Now add the following instance method to **DemoClass**:

```
mandalaCount
^MandalaCount
```

Then add the following code at the end of the **mandala** method in **DemoClass**:

```
MandalaCount isNil
ifTrue: [MandalaCount := 1]
iffFalse: [MandalaCount := MandalaCount + 1]
```

To see how many times **mandala** has been drawn, evaluate the following expression before and after running the demo program:

```
DemoClass new mandalaCount
```

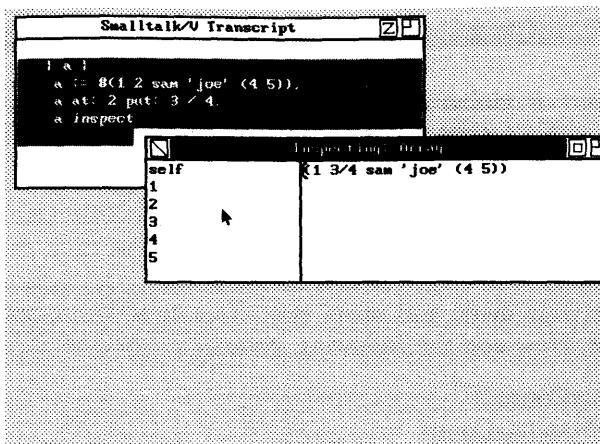
Inspectors

An *Inspector* is a window which allows you to view and change an object's instance variables. Evaluate the following expression to create an inspector window on an array:

```
| a |
a := #(1 2 sam 'joe' (4 5)).
a at: 2 put: 3 / 4.
a inspect
```

The inspector on an array looks as follows:

Figure 5.2
Inspector



The left pane of the inspector shows the names of the named instance variables and the numbers of the indexed instance variables. You select a name or number in the left pane to see the object contained in that variable in the right pane. You can create an inspector on the contents of an instance variable by selecting the variable, popping up the pane menu and selecting **inspect** or by clicking on a selected variable. Try this by inspecting the second instance variable of the array which contains a fraction object (3/4).

The fraction object has instance variables named **numerator** and **denominator** with values 3 and 4, respectively. Let's try changing the **denominator** variable. First select **denominator**, and then go to the right pane and use the text editor to replace 4 with 100. Then pop up the pane menu and select **save**. The instance variable is now changed. Just to make sure, close the **fraction** inspector window, return to the **array** inspector window, and select **self**. There it is; the fraction has been changed to 3/100.

What You've Now Learned

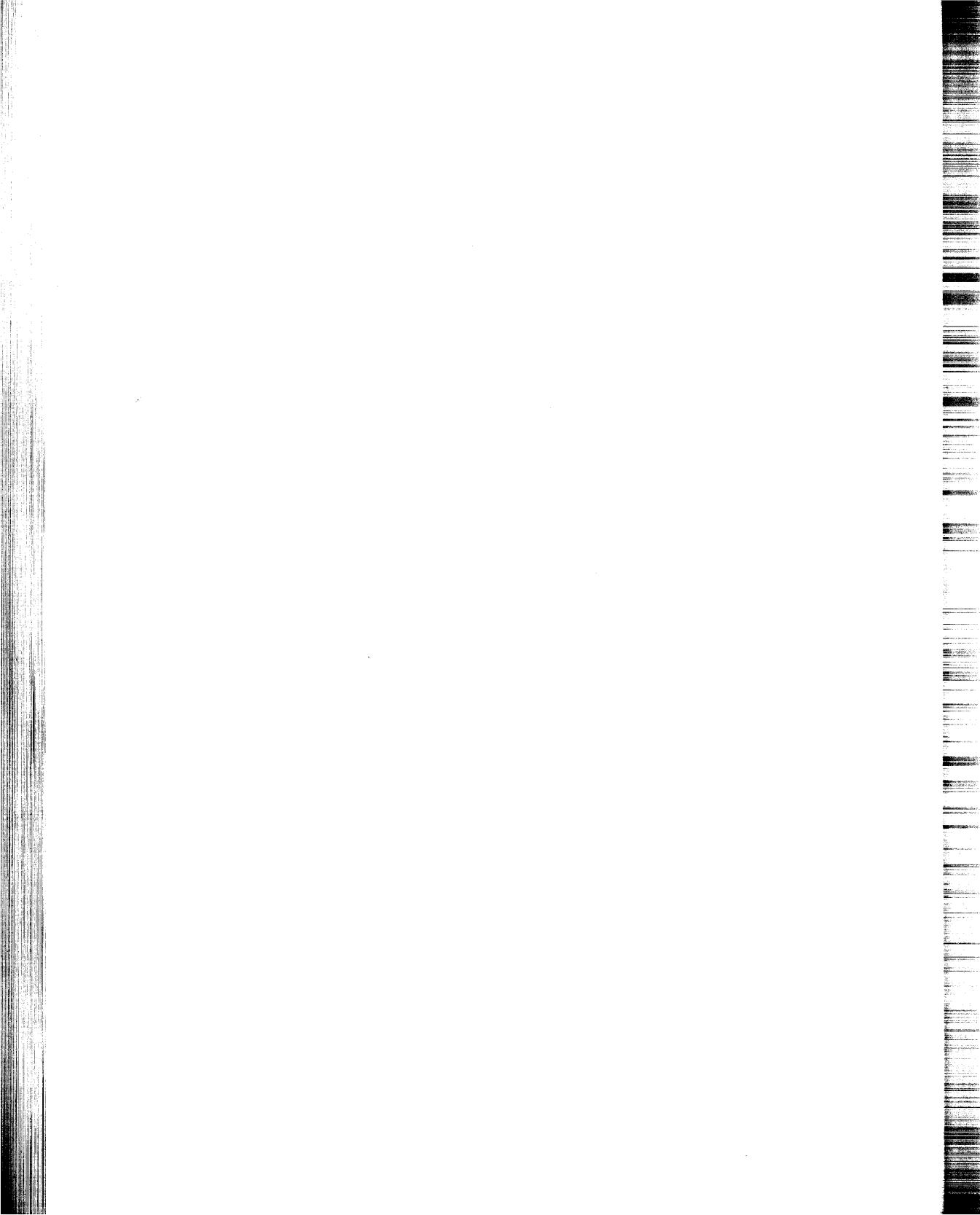
By the end of this tutorial, you should now be familiar with:

- classes and methods
- the Class Hierarchy Browser
- the special variable **self**
- creating new objects
- the special object **nil**
- named instance variables
- indexed instance variables
- recursion
- pattern matching
- adding new methods

- class variables
- inspectors

As always, if you need to review any of these topics, you can repeat that section of the tutorial, or refer to a complete description in **Part 3, The Smalltalk/V 286 Reference**.

If you are going to exit the **Smalltalk/V** environment before proceeding to the next tutorial, be sure to save the image.



6 INHERITANCE

This chapter presents Smalltalk's *class hierarchy* and the concept of *inheritance*. You'll see inheritance through an example of animal classification and see how to generalize the pattern matching method you saw in Chapter 5. You'll also see how to add new classes to Smalltalk/V, using the Class Hierarchy Browser. And finally, you'll be introduced to the Smalltalk concept of **polymorphism**, and how to process recursive data structures.

As always, the examples for this section are stored on a disk file, **chapter.6**. You can use the Disk Browser to load these files if you do not want to type the examples.

You'll also add new classes and methods to your environment during this lesson, so be sure to save the image when you exit the environment.

The Class Hierarchy

Much of Smalltalk's power comes from arranging its classes in a hierarchy. Each class has an immediate superclass and possibly one or more subclasses, with class **Object** at the top of the hierarchy. You're already familiar with this same system in biology, which arranges living organisms in classes, based on characteristics common to each class. Classes higher in the hierarchy represent more general characteristics, while classes lower in the hierarchy represent more specific characteristics. For example, **fish** and **tree** are more abstract than **halibut** and **maple**.

In Chapter 5, you saw how Smalltalk organizes its code (methods) by class. In this and later chapters, you will see how you can develop generic problem solutions using abstract classes, and then develop more application-specific solutions which "specialize" the general solution by adding a small amount of code in subclasses.

Close the Class Hierarchy Browser window opened in Chapter 5, pop up the system menu, and select the **browse classes** choice. This opens a new Class Hierarchy Browser on the entire class hierarchy. Select class **Boolean** in the class list pane. Then pop up the pane menu and select the **hide/show** menu choice or click on **Boolean** a second time. Now select class **True**, which shows you the following window:

SmalltalkV Transcript

Object
Behavior...
BitBlit...
BitEditor
Boolean
False
True
ClassBrowser
ClassListInSubBrowser

```

a
and:
basicHash
eqv:
hash
ifFalse:
ifFalse: ifTrue:
ifTrue:

Boolean subclass: #True
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''

```

Figure 6.1
Class True

Notice that the classes in the class list pane are indented. The indentations show the class hierarchy. Each class is a superclass of the classes indented below it. As you can see, **Object** is the superclass of all classes, and **Boolean** is the superclass of **True** and **False**.

A class with “...” following its name has subclasses that are not displayed. When you first open the Class Hierarchy Browser, it displays only the first level subclasses of class **Object**. This keeps the pane from becoming too cluttered. To display a class' hidden subclasses (or hide a class' displayed subclasses), simply use **hide/show**, as you did above to show **True** and **False** in class **Boolean** or double click on the class name. Try hiding the subclasses of class **Boolean**.

Inheritance

Inheritance is the Smalltalk capability which allows you to re-use software by specializing already existing general solutions. To see this, we'll define a new class hierarchy of *animals*:

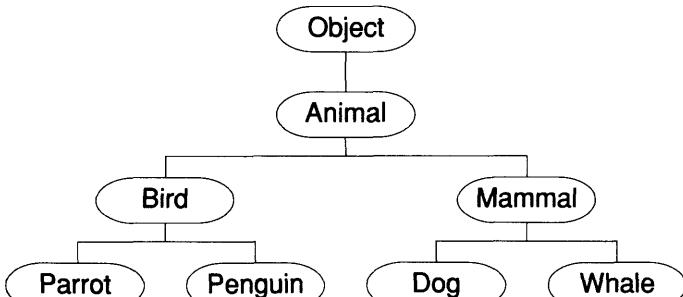


Figure 6.2
Animal Hierarchy

You'll see these same classes again in the following chapters to illustrate collections, graphics and window applications. The class **Animal** is a subclass of class **Object**. In turn, classes **Bird** and **Mammal** are subclasses of class **Animal**. Finally, classes **Parrot** and **Penguin** are subclasses of class **Bird**, and classes **Dog** and **Whale** are subclasses of class **Mammal**.

Whenever you define a new class, you also declare its instance variables. The following shows in parentheses the instance variables defined for each class in the animal hierarchy:

```
Animal (name, knowledge, habitat, topSpeed, color, picture)
Bird (flying)
Parrot (vocabulary)
Penguin ()
Mammal ()
Dog (barksAlot)
Whale ()
```

Inheritance of Instance Variables

In this chapter, we'll use the instance variables **name**, **vocabulary** and **barksAlot**. (You'll see the others used in subsequent chapters.) The instance variable **name** contains a string representing the animal's name, **vocabulary** contains a string of all words known by a parrot, and **barksAlot** contains either **true** or **false**, depending upon how much a dog barks.

An object inherits all the instance variables defined in its superclasses in addition to containing the ones defined in its own class. For example, parrots, penguins, dogs and whales each contain the following instance variables:

```
Parrot
  name, knowledge, habitat, topSpeed, color, picture, flying, vocabulary
Penguin
  name, knowledge, habitat, topSpeed, color, picture, flying
Dog
  name, knowledge, habitat, topSpeed, color, picture, barksAlot
Whale
  name, knowledge, habitat, topSpeed, color, picture
```

Normally, you create new classes using the Class Hierarchy Browser. (You'll see how to do this later in this chapter.) Since we have several classes and methods to define for the animal class hierarchy, however, we've simplified the procedure for you by putting them in a file. To add the animal classes to your Smalltalk/V environment, install the file by evaluating the following expression:

(File pathName: 'animal6.st') fileIn

Now, in order to see these new classes with the Class Hierarchy Browser, activate the Class Hierarchy Browser window, pop up the class list pane menu, and select the update function. Now all the animal classes are visible. If they are not, use the hide/show selection from the class list pane menu. By selecting its classes and methods, you can now browse the animal hierarchy.

The Methods of the Animal Classes

As you can see from the Class Hierarchy Browser, the methods you have just included for class **Animal** are as follows:

answer: aString

"Display a message for the receiver animal on the System Transcript window, consisting of the animal's class name and name preceding aString."

Transcript nextPutAll:

```
self class name, ', name, ': ', aString;
cr
```

name: aString

"Change the receiver animal's name to aString."

name := aString

talk

"Display a message that the receiver can't talk."

self answer: 'I can't talk'

Similarly, the methods for class **Parrot** are:

talk

"Display a message containing the receiver parrot's vocabulary."

self answer: vocabulary

vocabulary: aString

"Change the receiver parrot's vocabulary to aString."

vocabulary := aString

And finally, the methods for class **Dog** are:

bark

"Have the receiver dog bark by ringing the bell and displaying a bark message."

Terminal bell.

barksAlot

```
ifTrue: [self answer: 'Bow Wow, Bow Wow, Bow Wow!']
ifFalse: [self answer: 'Woof']
```

beNoisy

"Change the status of the receiver dog to noisy."

barksAlot := true.

self answer: 'I'll bark a lot'

beQuiet

"Change the status of the receiver dog to quiet."

barksAlot := false.

self answer: 'I won't bark much'

talk

"Have the receiver dog talk by barking unless barksAlot is nil, in which case the superclass can decide how to talk."

barksAlot isNil

ifTrue: [super talk]

ifFalse: [self bark]

We didn't define any methods for classes **Penguin** and **Whale**. However, these classes do inherit the methods of class **Animal**, so we can create whale and penguin objects and send messages to them, as we do later in this chapter.

Inheritance of Methods

Like instance variables, methods are also inherited. When a message is sent to an object, Smalltalk looks for the corresponding method defined in the object's class. If it finds the method, Smalltalk performs it. If it doesn't find the method, however, Smalltalk repeats the procedure in the object's superclass. This process continues all the way to class **Object**. If no method is found in any superclass, a walkback window pops up to display the error.

For example, look at the **name:** method defined in class **Animal**. Since this method is not defined in any of **Animal**'s subclasses, the **name:** method in class **Animal** is evaluated whenever a **name:** message is sent to instances of classes **Dog**, **Parrot**, **Penguin**, or **Whale**.

As another example, look at the `talk` method in the animal classes. Classes `Penguin` and `Whale` inherit `talk` from class `Animal`, whereas classes `Dog` and `Parrot` re-implement their own versions of `talk`.

The Special Variable “super”

Occasionally, you may want to override a method, instead of using a method higher in the superclass chain. Generally, you'd do this whenever the specialized processing done by a method doesn't apply in a particular case. You would instead use the more general processing of a method with the same name which appears higher in the superclass chain.

For example, look at the method `talk` for class `Dog`. A dog doesn't know how to talk if its instance variable `barksAlot` is undefined (has the value `nil`). In this case, it uses the following message to request the superclass' `talk` method:

```
super talk
```

The special variable `super` represents the same object as the special variable `self`: the receiver in the method in which it appears. The difference is that when a message is sent to `super`, Smalltalk looks for the method not in the receiver object's class, but instead in the superclass of the class containing the method in which `super` appears. In the example `talk` method, the search begins in `Mammal`, the superclass of class `Dog`. There is no `talk` method in class `Mammal`, but there is one in class `Animal`, so that one is used.

Creating Animal Objects

Evaluate the following expressions to create and assign to global variables five animal objects: two dogs, a penguin, a parrot and a whale (the animals “talk” to the System Transcript, so first reframe the Disk Browser window to not overlap the System Transcript):

"creating animals"
Snoopy := Dog new.
Snoopy name: 'Snoopy'.
Snoopy beQuiet.
Lassie := Dog new.
Lassie name: 'Lassie'.
Lassie beNoisy.
Wally := Penguin new.
Wally name: 'Wally'.
Polly := Parrot new.
Polly name: 'Polly'.
Polly vocabulary: 'Polly want a Cracker'.
Moby := Whale new.
Moby name: 'Moby'

Polymorphism

Polymorphism is a unique characteristic of object-oriented programming whereby different objects respond to the same message with their own unique behavior. For example, evaluate the following messages to see how the various animals respond to the **talk** message:

"let's hear them talk"
Lassie talk.
Snoopy talk.
Wally talk.
Polly talk; talk; talk.
**Polly vocabulary: 'Screeech@#!? Don''t bother me!'.
 Polly talk.
 Moby talk.
 Snoopy beNoisy; talk.
 Lassie beQuiet; talk**

Polymorphism lets you use entirely new classes of objects in existing applications, as long as they implement the message protocol required by the application. This greatly facilitates the reusing of generic code. A simple example is the method **max:** defined in class **Magnitude**, which returns the "maximum" of two objects:

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

The existing `max:` will work in any new subclass of `Magnitude`, as long as the new class implements the `greater than (>)` method.

More General Pattern Matching

In Chapter 5, you created a method `indexOfString:` to do pattern matching on strings. By relocating this method to a superclass of class `String`, we can use it to do pattern matching for several more classes. We'll change the name of the method to `indexOfCollection:` to suggest its more general capability, but we won't change the processing. Use the Class Hierarchy Browser to add the method `indexOfCollection:` to class `IndexedCollection`, a subclass of `Collection`:

```

indexOfCollection: aCollection
  "Answer the index position of the first occurrence
   of aCollection in the receiver. If no such element
   is found, answer zero."
  | index1 index2 limit1 limit2 |
  limit2 := aCollection size.
  limit1 := self size - limit2 + 1.
  index1 := 1.
  [index1 <= limit1]
    whileTrue: [
      (self at: index1) = (aCollection at: 1)
      ifTrue: [
        index2 := 2.
        [index2 <= limit2
          and: [(self at: index1 + index2 - 1) =
                  (aCollection at: index2)]]
          whileTrue: [index2 := index2 + 1].
        index2 > limit2
        ifTrue: [^index1].
      index1 := index1 + 1].
    ^0
  
```

Try the more general pattern matcher by evaluating the following examples using strings and arrays.

```

'the time has come' indexOfCollection: 'tim'
#($c $a $n $y $o $u $ ) indexOfCollection: 'you'
#(1 2 3 (4 5) 'abc' 6) indexOfCollection: #(2 3)
#(1 2 3 (4 5) 'abc' 6) indexOfCollection: 'abc'
  
```

Processing Recursive Data Structures

As an example of polymorphism and the processing of recursive data structures, consider the following method for equality (=), which appears in class **IndexedCollection**. This method compares an instance of class **IndexedCollection** or one of its subclasses (e.g., an array) to another similar object by sending the = message to corresponding elements of both objects. If the element is a kind of indexed collection, then the method performs a recursive send, invoking the = method. If the element is an object such as a number, the method performs a non-recursive send, invoking a different = method.

```
= aCollection
    "Answer true if the elements contained by
     the receiver are equal to the elements
     contained by the argument aCollection."
| index |
self == aCollection
    ifTrue: [ ^true].
(self class == aCollection class)
    ifFalse: [ ^false].
index := self size.
index ~= aCollection size
    ifTrue: [ ^false].
[index <= 0]
    whileFalse: [
        (self at: index) = (aCollection at: index)
        ifFalse: [ ^false].
        index := index - 1.]
^true
```

This method also demonstrates the difference between equality (=) and equivalence (==). Equality tests whether two objects contain the same elements. Equivalence, on the other hand, tests whether two objects are, in fact, the same object. For example, the expression

```
self == aCollection
```

tests whether the receiver object is the same actual object as the argument. If they are, then they are obviously equal, and the method returns **true**.

To help see this difference, evaluate the following statement:

```
| a b |
a := #( 1 2 3 4 ).
b := #( 1 2 3 4 ).
```

$$\wedge a = b$$

This expression returns **true**, because the two objects contain the same elements. Now substitute **=** with **==**, and re-evaluate the statement. This returns **false**, because, although the two objects contain the same elements, they are still two different objects. As an example of a true equivalence, evaluate the following expression:

```
| a b c |
a := #( 1 2 3 4 ).  
b := a.  
c := b.  
a == a.
```

To use the inherited **=** method on recursive data structures, evaluate these expressions:

```
#(1 (2 (3))) = #(1 (2 (3)))
#(john smith) = #(john smith)
#(1 'two' 3) = #(1 'two' 3)
```

Since the **indexOfCollection:** method defined above compares elements with the **=** message, it can be applied to nested (recursive) collections. For example, show the results of the following expressions:

```
#((1 2)(3 4)(5 6)) indexOfCollection: #((3 4)(5 6))
#(1 2 3 (4 5) 'abc' 6) indexOfCollection: #(3 (4 5) 'abc')
#(1 2 3 (4 5) 'abc' 6) indexOfCollection: #('abc')
```

A New Class: **MonitoredArray**

The final example of this chapter creates a new class to monitor the frequency of access to the data in an **array**. For instance, suppose you have an array of sales tax rates for California, indexed by zip code minus 90000 (California zip codes begin with 9). If you know how frequently each sales tax rate is looked up by zip code, you can compute the average sales tax paid, shipments to each region, and several other statistics.

To do this, we create class **MonitoredArray** as a subclass of class **Array**. A monitored array is like a normal array, except that it also maintains a parallel array containing the number of times the **at:** message was used for each index value. A monitored array can be substituted for an array in any application. Like any subclass, it inherits all the behaviors of its superclass (**arrays**), and implements some special behaviors of its own.

To add the new class, select class **Array** on the Class Hierarchy Browser. You'll find it as a subclass of **FixedSizeCollection**, which is a subclass of **IndexedCollection**, which is a subclass of **Collection**. With the cursor in the class list pane, pop up the pane menu and select **add subclass**. You'll then see a prompter, asking you for the **Array Subclass**

name; enter **MonitoredArray**. Finally, you'll see another menu, from which you should choose the **variableSubclass** entry. The class list is then updated, with class **MonitoredArray** selected.

Now you must specify the new class' instance variables. Proceed to the text pane below and edit the class definition to appear as follows:

```
Array variableSubclass: #MonitoredArray
instanceVariableNames: 'atCounts'
classVariableNames: ''
poolDictionaries: ''
```

Pop up the text pane menu and select the **save** entry. The class definition is updated.

Class Methods

Class methods respond to messages sent to class objects, rather than to instances of the class. Class methods are often used for creating initialized instances of a class.

As an example, we'll create a class method for class **MonitoredArray**. Select class **MonitoredArray** using the Class Hierarchy Browser. Then select the pane labeled **Class**. This reverses the pane contents, indicating that any new methods added are class methods. Now pop up the method list pane menu, and select **new method**. The contents pane then displays a template reminding you of a new method's required components. Type the following method into the contents pane, replacing the template:

```
new: anInteger
    "Answer a new MonitoredArray."
| answer |
answer := super new: anInteger.
answer initialize.
^answer
```

Now pop up the pane menu, and select **save**, which adds this class method to class **MonitoredArray**. We re-implement **new:** for class **MonitoredArray**, because the inherited **new:** method for arrays does not initialize the **atCounts** instance variable.

The remaining three **MonitoredArray** methods are instance methods. Select the Class Hierarchy Pane labeled **Instance**, and add these three instance methods one at a time:

```
accessCounts
    "Answer the array of 'at:' counts."
^atCounts
```

```

at: anInteger
    "Answer the element in the receiver at
     index position anInteger. Increment
     the count for accesses to the receiver
     using anInteger."
atCounts
    at: anInteger
        put: (atCounts at: anInteger) + 1.
    ^super at: anInteger

initialize
    "Private - Initialize the MonitoredArray by
     allocating and initializing the parallel
     atCounts array."
    | size |
    size := self size.
    atCounts := Array new: size.
    1 to: size do: [ :index | atCounts at: index put: 0]

```

As an example of using a `MonitoredArray`, evaluate and show the results of the following expressions:

```

| array |
array := MonitoredArray new: 20.
1 to: 10 do: [ :i |
    1 to: 10 do: [ :j | array at: i + j]].
^array accessCounts

```

What You've Now Learned

After completing this tutorial, you should now be familiar with:

- the class hierarchy
- inheritance, both of methods and of variables
- polymorphism
- general pattern matching
- processing recursive data structures
- class methods

As always, you can review any of these topics by repeating the corresponding section of the tutorial, or by referring to the detailed description in **Part 3, The Smalltalk/V 286 Reference**.

If you exit the environment before beginning the next tutorial, be sure to save the image.

7 STREAMS AND COLLECTIONS

This chapter introduces you to two of Smalltalk's most widely used hierarchies: the stream classes and the collection classes. At the end of this chapter, you'll see four interesting examples using both of these hierarchies.

As always, the examples for this tutorial are stored in a disk file, **chapter.7**. You can use the Disk Browser to retrieve these examples, if you do not want to type them.

You will also be altering the image during this tutorial, so be sure to save the image when you exit the environment.

Streams

Smalltalk supports many different kinds of stream objects. You already saw one of them when you accessed disk files using **FileStream** objects. Streams are also used for accessing the keyboard and mouse (class **TerminalStream**), and for accessing internal collections of objects, such as strings and arrays (classes **ReadStream**, **WriteStream**, and **ReadWriteStream**). The stream classes are arranged in a hierarchy with the class **Stream** as the superclass. You can use the Class Hierarchy Browser (explained in Chapter 6) to explore this hierarchy.

This chapter will present a series of examples using streams, which should give you a good introduction. Part 3, **The Smalltalk/V 286 Reference**, gives a detailed description of streams and all of the messages that can be used with them.

Streams are frequently used for scanning input or producing edited output. For example, look at this method, which does both:

```
"Replace occurrences of % with the date today"
| input output char dateStamp |
dateStamp := Date today printString.
input := ReadStream on: 'The date today is %'.
output := WriteStream on: String new.
[input atEnd]
  whileFalse: [
    (char := input next) = $%
      ifTrue: [output nextPutAll: dateStamp]
      ifFalse: [output nextPut: char]].
^output contents
```

This example creates two streams. The **on:** message is sent to the class **ReadStream** to create a stream on the argument string '**The date today is %**'. The **on:** message is also used to create a **WriteStream** on an empty string, to hold the edited output.

As the names imply, **ReadStream** can only be read and **WriteStream** can only be written. As we have seen previously with the disk file examples, streams are read with the **next** message and written with the **nextPut:** message. The message **atEnd** tests if there is more input to be read. The message **nextPutAll:** writes several objects to a stream at once. In the above example, the argument is a string of characters containing today's date.

The above example streams over strings of characters. It uses an empty string for the write stream because streams automatically grow as necessary, to accommodate the objects written to them. The **contents** message returns a string containing all of the objects written to the stream.

To change the above example to use disk files instead of streams on strings, simply change the messages that create the streams **input** and **output**. This illustrates one of Smalltalk's most powerful features: you can write programs that are dependent on the *behavior*, rather than the *structure*, of data. This means that you can write and test a program using simple internal objects, such as streams on strings, and then easily extend it to use external files.

Streams are not restricted to reading and writing only characters. For example, this method reads and writes arrays of numbers:

```
"Compute several factorials"
| input output |
input := ReadStream on: #( 1 5 10 20 ).
output := WriteStream on: Array new.
[input atEnd]
  whileFalse: [
    output nextPut: input next factorial].
^output contents
```

Although these examples do not show it, streams can also be repositioned, much like a random access file, using the **position:** message. The argument is an integer. You can also use the **position** message to access a stream's current position.

Printer Stream

To see how easy it is to make major enhancements to **Smalltalk/V**, let's add a new class **PrinterStream**. This new class will allow you to use all of the stream messages with your printer. Define **PrinterStream** as a subclass of **WriteStream** (which is a subclass of **Stream**) with the type **subclass**. It needs no instance variables, class variables, or pool dictionaries. (Chapter 6 explains how to make new classes using the **Class Hierarchy Browser**.) After the class is created, the class definition displayed in the **Class Hierarchy Browser** should be:

```
WriteStream subclass: #PrinterStream
  instanceVariableNames: ""
  classVariableNames: ""
  poolDictionaries: ""
```

Class **PrinterStream** needs only two methods. Again, we've included these methods for you in a file. Evaluate the following expression to install them:

```
(File pathName: 'prntrst7.st') fileIn
```

This installs the following methods:

```
nextPut: aCharacter
  "Write aCharacter to the receiver."
  | string |
  string := ' '. "a string with one blank"
  string at: 1 put: aCharacter.
  string outputToPrinter.
  ^aCharacter

nextPutAll: aString
  "Write aString to the receiver."
  aString outputToPrinter.
  ^aString
```

Remember to use the **update** function from the class list pane menu to display the new methods.

PrinterStream will inherit its other methods (and all of its variables) from the classes **WriteStream** and **Stream**. To create an instance of the **PrinterStream** class in the global variable **Printer**, evaluate the following expression:

```
Printer := PrinterStream new
```

As an example of using a printer stream, produce a printed report by evaluating the following program:

```
"Print the first 10 even numbers and their factorials"
| width factorial |
width := 20 factorial printString size.
2 to: 20 by: 2 do: [:i |
  factorial := i factorial printString.
  Printer
    next: 4 - i printString size put: Space;
    nextPutAll: i printString, ',';
    next: width - factorial size put: Space;
    nextPutAll: factorial;
    cr]
```

If you do not have a printer, you can use the global variable `Transcript` instead of `Printer` in the above example. This will print the report in the System Transcript window. (The `Transcript` object is not a kind of stream, but it does support many of the same messages as streams.)

Collections

Collections are objects which contain a collection of other objects. You have already seen two kinds of collections: `Arrays` and `Strings`. `Strings` are fixed sized sequences of characters, while arrays are fixed sized sequences of arbitrary objects. You have used the iterator messages `do:`, `collect:`, `select:`, and `reject:` with arrays and strings. These messages are understood by all of the collection classes, three of which are `Dictionary`, `Bag`, and `Set`.

Dictionaries

Dictionaries store and retrieve objects by using a key. For example, let's create a simple phone book. First, create a global variable containing an empty dictionary by evaluating the following:

```
PhoneBook := Dictionary new
```

To add phone numbers to the phone book, use the `at:put:` message:

PhoneBook

```
at: 'Marisa' put: '645-1082';
at: 'Francesca' put: '555-1212';
at: 'Jackie' put: '392-481-5000';
at: 'Rakesh' put: '645-1083';
at: 'Vijay' put: '645-1083'
```

In the above expressions, the strings '`Marisa`' and '`Francesca`' are the keys, and the strings '`645-1082`' and '`555-1212`' are the corresponding values. Notice that `at:put:` is also used to access the elements of strings and arrays. With dictionaries, however, the first argument is the key in the dictionary, instead of the position in the array or string.

To retrieve an object from a dictionary, use the `at:` message with the key as the argument. For example the following expression returns the string '`645-1082`':

```
PhoneBook at: 'Marisa'
```

To test if an object exists as a key in the dictionary, use the `includesKey:` message, as in the following expression:

```
(PhoneBook includesKey: 'Aaron')
  ifTrue: [PhoneBook at: 'Aaron']
  ifFalse: ['Not in phone book']
```

A simpler way to do this is to use the `at:ifAbsent:` message. The first argument is the key and the second argument is a block of code that will be executed if the key is not in the receiver dictionary. For example.

```
PhoneBook at: 'Aaron' ifAbsent: ['Not in phone book']
```

The keys and the values stored in a dictionary can be any kind of object.

Dictionaries are such useful objects that a special inspector window exists called the *Dictionary Inspector*. To open a dictionary inspector on the phone book, evaluate the following expression:

```
PhoneBook inspect
```

The pane on the left of the window is a sorted list of all of the keys in the dictionary, in our case the names of people in the phone book. When you select a key, the corresponding value is displayed in the pane on the right, in our case the person's phone number. By using the pane menus, entries can be edited, added, and removed.

Bags

Bags store an arbitrary number of objects of any kind. Unlike arrays, there is no implied order or sequence to the elements (objects) inside the bag. Elements are added to a bag with the `add:` message. To test if an object is in a bag, use the `includes:` message. For example, this expression reads a file and reports the frequency with which each letter occurs:

```
| input answer f c |
input := File pathName: 'chapter.7'.
answer := WriteStream on: String new.
f := Bag new.
[input atEnd]
  whileFalse: [
    (c := input next) isLetter
      ifTrue: [f add: c asLowerCase]].
0 to: 25 do: [ :i |
  c := ($a asciiValue + i) asCharacter.
  answer
    cr; nextPut: c; space;
    nextPutAll: (f occurrencesOf: c) printString].
^answer contents
```

Sets

A **Set**, like a **Bag**, stores arbitrary objects. The difference is that a **Set** does not store the same object more than once. For example, this expression computes the set of characters that occur in one file and not in another:

```
| set1 set2 |
set1 := Set new.
set2 := Set new.
(File pathName: 'chapter.7') do: [:c| set1 add: c].
(File pathName: 'chapter.6') do: [:c| set2 add: c].
^set1 reject: [ :c | set2 includes: c]
```

The message **asSet** creates a set out of the receiver collection object. This is a good way to eliminate duplicates from a collection. For example, to compute the unique vowels in a string, evaluate the following:

```
'Now is the time' asSet select: [ :c | c isVowel ]
```

Generic Code

You've now seen the iterator messages **select:**, **reject:**, and **collect:** used with strings, arrays, sets, and bags. These messages can be used with all of the different kinds of collections in **Smalltalk/V**. As such, they are excellent examples of generic code, code that is type and data independent. Smalltalk's ability to allow you to write generic code sets it apart from most other languages. Here is the code in class **Collection** for the **select:** message that is inherited by bags, sets, dictionaries, sorted collections, ordered collections, and other collection classes:

```
select: aBlock
| answer |
answer := self species new.
self do: [ :element |
  (aBlock value: element)
    ifTrue: [answer add: element]].
^answer
```

This method assumes nothing about the structure or type of the collections with which it deals. It depends only on an object's behavior, the existence of the **species**, **do:**, and **add:** it sends to them. Smalltalk's polymorphism (discussed in Chapter 6) makes this possible.

By exploring the **Collection** classes using the Class Hierarchy Browser, you can see many more examples of the power of generic code.

Blocks as Objects

The `select:` message above shows another interesting feature of Smalltalk: the use of blocks of code as objects. To illustrate this, look at the following invocation of the `select:` method.

```
'Now is the time' asSet select: [ :c | c isVowel ]
```

The receiver of the `select:` message is the set of all characters in the string '`Now is the time`'. The argument to the `select:` message is a block of code with one block argument, `[:c | c isVowel]`. This block of code is as much an object as the string '`Now is the time`'. As such, we can use it as an argument for the `select:` method, which you saw previously. When the method is invoked, the block of code is assigned to the argument `aBlock` in the `select:` method.

A block of code executes when it is sent the message `value`, `value:`, or `value:value:`, depending on whether the block has zero, one, or two block arguments, respectively. Since it uses one argument, the `select:` message evaluates the block `aBlock` using the `value:` message.

As you now know, all messages return a result. The result of evaluating a block is the result of the last expression in the block. In the above example, the block `[:c | c isVowel]` returns `true` or `false`, depending on whether or not the object passed to the block, `c`, is a vowel.

Patterns

Block objects in turn let you build very powerful objects. For example, look at the class `Pattern`. Patterns are generalized and efficient pattern matchers. A pattern object consists of a collection of objects to match, and a block of code to execute when the pattern is successfully matched. For example, this expression computes the number of occurrences of a phrase in a file:

```
"Compute occurrences of a phrase in a file"
| pattern count input word |
count := 0.
(pattern := Pattern new: #( 'now' 'is' 'the' ))
    matchBlock: [count := count + 1].
input := File pathName: 'chapter.7'.
[(word := input nextWord) isNil]
    whileFalse: [pattern match: word asLowerCase].
^count
```

This example uses an array of strings as the pattern. Any collection of objects can be used as the pattern, as long as it can be indexed using the `at:` message.

Computing Letter Pair Frequencies

The following example computes the frequency with which letter pairs occur in a file, and stores the result in the global variable, Pairs:

```
"compute letter pair frequencies"
| last pair |
Pairs := Bag new.
last := Space.
(File pathName: 'chapter.7') do: [ :c |
    (last isLetter and: [c isLetter])
        ifTrue: [
            (pair := String new: 2)
                at: 1 put: last;
                at: 2 put: c.
            Pairs add: pair asLowerCase].
    last := c]
```

The following expression, in turn, produces a report of the pair frequencies that occur more than 60 times in Pairs:

```
"print letter pair frequencies greater than 60
in the Transcript"
| frequent |
Transcript cr.
frequent := Pairs asSet select: [ :pair |
    (Pairs occurrencesOf: pair) > 60 ].
frequent asSortedCollection do: [:pair |
    Transcript
        nextPutAll: pair;
        tab;
        nextPutAll: (Pairs occurrencesOf: pair)
            printString;
        cr]
```

The message `asSortedCollection` creates a new kind of collection, a `SortedCollection`. `SortedCollections` are described in detail in [Part 3, The Smalltalk/V 286 Reference](#). Briefly, they are collections in which all of the elements are stored in sorted order. As you can see from the above example, they are useful for sorting a collection of objects before outputting a report.

Animals Revisited

In Chapter 6, we built a simple hierarchy of animal classes. In this section, we will give those animals an environment (habitat) in which to live and a way to acquire knowledge and interact with their habitat.

The habitat will have a set of animals that inhabit it. Every animal will store knowledge as a collection of patterns, instances of class **Pattern**. In this case a pattern is a sequence of words that, when recognized by the pattern, evaluates a corresponding block of code. This causes the animal to react to a word sequence in some prescribed way. The global variable **Script** contains a stream of words to send to all of the animals. Giving many different patterns to a single animal provides that animal with a rich set of behaviors.

Animal Habitat

Create a new class **AnimalHabitat** as a subclass of class **Object**, and assign to it five instance variables, **animals**, **replyStream**, **animator**, **inputString**, and **inputPane**. (Chapter 6 explains how to do this using the Class Hierarchy Browser.) When the new class is successfully created with the five instance variables, you should see the following class definition when it is selected in the Class Hierarchy Browser:

```
Object subclass: #AnimalHabitat
instanceVariableNames:
    'animals replyStream animator
     inputString inputPane'
classVariableNames: ''
poolDictionaries: ''
```

The instance variable **animals** will contain the set of animals that inhabit the habitat. (The instance variables **replyStream**, **animator**, **inputString**, and **inputPane** are used in a later tutorial.)

Now evaluate the following expression to file in the methods for the **AnimalHabitat**:

```
(File pathName: 'habitat7.st') fileIn
```

Click the cursor over the **instance** label to view the methods list. The new methods are:

```
add: anAnimal
    "Add anAnimal as an inhabitant of the receiver.
     Notify anAnimal of its new habitat."
animals isNil
    ifTrue: [animals := Set new].
animals add: anAnimal.
anAnimal habitat: self
```

play

```

    "Play the Script to all of the animals."
    | word |
    Script reset.
    animals do: [ :animal | animal reset ].
    [Script atEnd]
    whileFalse: [
        word := Script next asLowerCase.
        animals do: [ :animal | animal reactTo: word ]]

```

script: aString

```

    "Change Script to the stream on the
     words in aString."
    | stream word |
    stream := ReadStream on: aString.
    Script := ReadWriteStream on: Array new.
    [(word := stream nextWord) isNil]
    whileFalse: [
        Script nextPut: word ]

```

Now evaluate the following expression to create a global variable, **Habitat**, containing an instance of the **AnimalHabitat** class:

Habitat := AnimalHabitat new

Animal Knowledge

To put animals inside of the habitat, you must first add some methods to the **Animal** class. Evaluate the following expression to add the required methods:

(File pathName: 'animal7.st') fileIn

The new methods in class **Animal** are:

```

habitat: aHabitat
    "Change habitat to aHabitat"
    habitat := aHabitat

```

```

learn: aString action: aBlock
    "Add a pattern of the words in aString to the
     receivers knowledge. The action to perform
     when the pattern is matched is aBlock."
    | words pattern |
    knowledge isNil
        ifTrue: [knowledge := Dictionary new].
    words := aString asLowerCase asArrayOfSubstrings.
    pattern := Pattern new:
        (Array with: name asLowerCase), words.
    pattern matchBlock: aBlock.
    knowledge at: words put: pattern

reactTo: aWord
    "Send a word to every pattern in knowledge."
    knowledge isNil
        ifTrue: [^self].
    knowledge do: [:pattern | pattern match: aWord]

reset
    "Reset all patterns in knowledge"
    knowledge isNil
        ifTrue: [^self].
    knowledge do: [:pattern | pattern reset]

```

Using the Habitat

First, let's add some animals to the habitat. The following expressions use the animals that were created in Chapter 6:

```

Habitat
add: Snoopy;
add: Polly

```

Now, set up a script to work with:

```

Habitat script:
'Snoopy is upset about the way that Polly is
behaving. It is as if whenever anyone asks
Polly to talk, Polly will be nasty. Maybe if
instead of Snoopy barking at Polly when he
wants Polly to talk, Snoopy quietly asks Polly
to be pleasant for a change, things would go
better. Now maybe Snoopy barking quietly will
not make Polly nasty.'

```

Before playing the script, we need to give the animals some knowledge:

Snoopy

```
learn: 'barking' action: [Snoopy talk];
learn: 'quietly' action: [Snoopy beQuiet; talk];
learn: 'is upset' action: [Snoopy beNoisy; talk].
```

Polly

```
learn: 'to be pleasant' action:
[Polly vocabulary: 'Have a nice day'; talk];
learn: '* nasty' action:
[Polly vocabulary: 'Why are you bothering me';
talk].
```

The asterisk (*) in '* nasty' stands for none or more arbitrary words. To play the script to the animals, evaluate the following expression:

Habitat play

Look in the System Transcript to see the responses from the animals.

A Network of Nodes

As a final example of streams and collections, we will build a network of nodes, and determine paths through the network. Many problems can be described in terms of networks of nodes and paths through the network, such as route maps, pert charts, and many kinds of optimization problems.

Network

A *network* is a collection of nodes that are connected to each other. Create the class **Network** as a subclass of class **Object**, and define a single variable named **connections**. When you have created the class and the instance variable, the class specification in the Class Hierarchy Browser should be:

```
Object subclass: #Network
instanceVariableNames:
  'connections'
classVariableNames: ''
poolDictionaries: ''
```

The instance variable **connections** will hold a dictionary of connections between nodes. The key to the dictionary will be a node, and the value stored under that key will be a set of all of the nodes to which it is connected. Use the following expression to file in the methods for class **Network**:

(File pathName: 'network7.st') fileIn

The methods are:

```

connect: nodeA to: nodeB
    "Add a connection from nodeA to nodeB."
    (connections
        at: nodeA
        ifAbsent: [connections at: nodeA put: Set new])
            add: nodeB.
    (connections
        at: nodeB
        ifAbsent: [connections at: nodeB put: Set new])
            add: nodeA

initialize
    "Initialize the connections to be empty."
    connections := Dictionary new

pathFrom: nodeA to: nodeB avoiding: nodeSet
    "Answer a path of connections that connect nodeA
     to nodeB without going through the nodes in
     nodeSet. This result is returned as a new
     network. Answer nil if there is no path"
    | answer |
    nodeSet add: nodeA.
    (connections at: nodeA ifAbsent: [^nil]) do:
        [:node |
         node = nodeB
         ifTrue: [
             ^Network new initialize
             connect: nodeA to: node].
         (nodeSet includes: node)
         ifFalse: [
             answer := self
             pathFrom: node
             to: nodeB
             avoiding: nodeSet.
             answer isNil
             ifFalse: [
                 ^answer connect: nodeA to: node]].
         ^nil
    
```

```

printOn: aStream
    "Print a description of the receiver on aStream."
    connections keys asSortedCollection do: [ :node |
        node printOn: aStream.
        (connections at: node) asSortedCollection do:
            [ :neighbor |
                aStream
                    cr;
                    nextPutAll: ' >> '.
                neighbor printOn: aStream].
            aStream cr]

```

Notice the recursion in the **pathFrom:to:avoiding:** message. This is a simple solution; it does not find the optimal (shortest) path. If you want to find such an optimal solution, however, you need only change this one method. This is another of Smalltalk/V's characteristics. You can quickly build program fragments to start exploring the nature of the problem being solved. When you better understand the problem, the changes are quick and localized.

Network Nodes

Before using the **Network** class, define the class **NetworkNode** as a subclass of class **Object**, with two instance variables, **name** and **position**. After you have created the class and its instance variables, the class specification should be:

```

Object subclass: #NetworkNode
    instanceVariableNames:
        'name position'
    classVariableNames: ""
    poolDictionaries: ""

```

Then use the following expression to file in the methods for class **NetworkNode**:

```
(File pathname: 'nodes7.st') fileIn
```

The methods are:

```

<= aNode
    "Answer true if the receiver name is less or
     equal to aNode name."
    ^name <= aNode name

hash
    "Answer receiver's hash."
    ^name hash

```

```

name
    "Answer receiver's name."
^name

name: aString position: aPoint
    "Set the receiver's name and position."
    name := aString.
    position := aPoint

printOn: aStream
    "Print a description of the receiver on aStream."
aStream
    nextPutAll: 'Node( ', name;
    space;
    nextPutAll: position printString;
    nextPut: $)

```

Building a Network

Now evaluate the following expression to create an empty network in the global variable Net:

```
Net := Network new initialize
```

Then evaluate these expressions, to create six nodes and connect them together into a network:

```

N1 := NetworkNode new name: 'one' position: 300 @ 100.
N2 := NetworkNode new name: 'two' position: 400 @ 150.
N3 := NetworkNode new name: 'three' position: 500 @ 120.
N4 := NetworkNode new name: 'four' position: 200 @ 50.
N5 := NetworkNode new name: 'five' position: 350 @ 195.
N6 := NetworkNode new name: 'six' position: 550 @ 130.
Net
    connect: N1 to: N2;
    connect: N2 to: N3;
    connect: N4 to: N5;
    connect: N5 to: N1;
    connect: N3 to: N6;
    connect: N3 to: N5;
    connect: N3 to: N1

```

You can ask the network to print itself by evaluating the following expression using show it:

```
Net
```

Now evaluate the following expression and show the results, to find a path from **N1** to **N5**:

Net pathFrom: N1 to: N5 avoiding: Set new

To see if there is a path that does not go through **N3**, evaluate the following expression:

Net pathFrom: N1 to: N5 avoiding: (Set with: N3)

What You've Now Learned

After having completed this tutorial, you should be familiar with:

- streams, including **PrinterStream**
- collections, including Dictionaries, Bags, and Sets
- generic code

As always, you can review any of these topics by repeating the corresponding section of the tutorial, or by referring to a detailed description in **Part 3, The Smalltalk/V 286 Reference**.

If you exit the environment before beginning the next tutorial, be sure to save the image.

8 DEBUGGING

This chapter uses what you learned in Chapter 7 to build a complete program using collections and streams. The program, however, purposely contains several errors. This chapter, then, shows you how to locate and correct errors using the **Smalltalk/V** debugger.

As always, the examples for this chapter are stored in the disk file, **chapter.8**. You can use the Disk Browser to retrieve these examples.

Since you will again be making modifications to your **Smalltalk/V** environment, be sure to save the image when you exit **Smalltalk/V**.

A Document Retrieval System

The first thing we'll do is implement a new class, **WordIndex**, which allows you to create a database of documents and locate them based upon the words that they contain. *Documents* are ASCII text files, viewed as a series of words containing alphanumeric characters separated by a series of non-alphanumeric characters. You query the database by supplying a collection of word strings, which returns a collection of the file names of all the documents that contain all the words. You could use the word index, for example, to locate resumes in a personnel system, such as all employees whose resumes contain the words **C** and **Unix**.

Instances of class **WordIndex** have instance variables **documents** and **words**.

documents is a set of strings of the document file path names whose words have been entered into the word index.

words is a dictionary, with each key containing a string for a word and each value being a set containing the path names of all documents containing the word.

Therefore, the class definition is:

```
Object subclass: #WordIndex
  instanceVariableNames:
    'documents words'
  classVariableNames: ""
  poolDictionaries: ""
```

Add **WordIndex** class definition and methods to your **Smalltalk/V** image by evaluating the following expression:

(File pathName: 'wrdindx8.st') fileIn

Now select update in the Class Hierarchy Browser's class list so that you can browse the methods of class WordIndex. There are six methods defined for class WordIndex, as follows:

```

addDocument: pathName
    "Add all words in document described by
     pathName string to the words dictionary."
    | word wordStream |
    (documents includes: pathName)
        ifTrue: [self removeDocument: pathName].
    wordStream := File pathName: pathName.
    documents add: pathName.
    [(word := wordStream nextWord) == nil]
        whileFalse: [
            self addWord: word asLowerCase to: pathName].
    wordStream close

addWord: wordString for: pathName
    "Add wordString to words dictionary for
     document described by pathName."
    (words at: wordString) add: pathName

initialize
    "Initialize a new empty WordIndex."
    documents := Set new.
    words := Dictionary new

locateDocuments: queryWords
    "Answer an array of the pathNames for
     all documents which contain all words
     in queryWords."
    | answer bag |
    bag := Bag new.
    answer := Set new.
    queryWords do: [ :word |
        bag addAll:
            (documents at: word ifAbsent: [#()])].
    bag asSet do: [ :document |
        queryWords size =
            (bag occurrencesOf: document)
                ifTrue: [answer add: document]].
    ^answer asSortedCollection asArray

```

```

removeDocument: pathName
    "Remove pathName string describing a
     document from the words dictionary."
    words do: [ :docs | docs remove: pathName].
    self removeUnusedWords

removeUnusedWords
    "Remove all words which have empty
     document collection."
    | newWords |
    newWords := Dictionary new.
    words associationsDo: [ :anAssoc |
        anAssoc value isEmpty
            ifFalse: [newWords add: anAssoc]].
    words := newWords

```

How Class WordIndex Works

Next, we'll describe class `WordIndex` in terms of the high-level messages which create an index and make queries.

We mentioned earlier that we've included some intentional errors; this is the first place where they occur. For this reason, don't evaluate these messages until the tutorial tells you to do so.

We'll construct and use the word index in three steps. First, we create an empty word index in an expression such as the following (remember, don't evaluate this expression yet):

`Index := WordIndex new initialize`

The `initialize` method *initializes* instance variables of the `WordIndex`; that is, `documents` now contains an empty set and `words` contains an empty dictionary.

Next, we'll add the words from documents to the `WordIndex`. The `addDocument:` method creates a file stream to scan the document, repeatedly sends the `nextWord` message to the file stream to obtain each word, and then uses the `addWord:for:` method to enter each word/document pair in the `words` dictionary. For example, to add the words from the Chapters 5 and 6 sample files, you would use the following expressions (again, don't evaluate these yet):

`Index addDocument: 'chapter.5'.`
`Index addDocument: 'chapter.6'.`

To query the word index, you use the `locateDocuments:` message, as in the following examples (again, do not evaluate them):

```
Index locateDocuments: #(‘show’ ‘class’)
Index locateDocuments: #(‘where’ ‘the’ ‘turtle’)
Index locateDocuments: #(‘each’ ‘talk’)
```

Each query above returns an array of strings, containing the document path names for all documents that contain all words in the query. The **locateDocuments:** method is somewhat more complex than the other methods in its class. It uses a bag to accumulate all the path names of all the files that contain each word in the query. (Remember that bags, unlike sets, can contain multiple occurrences of the same object.) It then examines the bag to find any documents which are repeated as many times as there are words in the query; these are the documents which contain all the words.

Debugging Class WordIndex

Now that you've seen how this class is supposed to work, let's see if it does. (From this point, start evaluating the sample expressions again.) First, build a new word index and assign it to the global variable **Index**:

```
Index := WordIndex new initialize
```

Now try adding the tutorial files for Chapters 5 and 6 by evaluating the following **addDocuments:** messages:

```
Index addDocument: ‘chapter.5’.
Index addDocument: ‘chapter.6’.
```

Oops! Instead of adding the tutorial files, we get a walkback window:

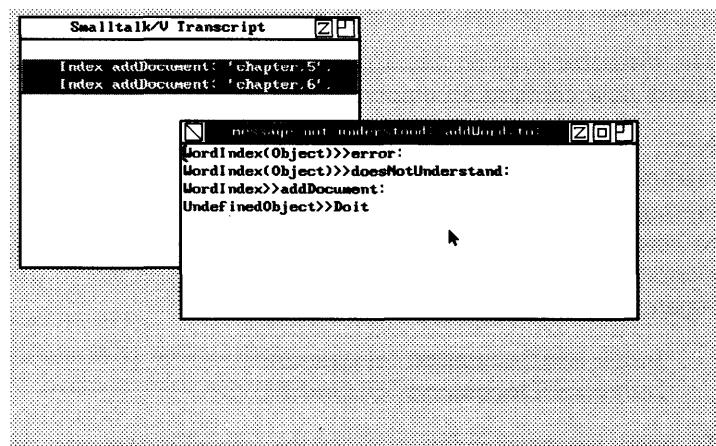


Figure 8.1
Walkback Window

As you saw in Chapter 2, a walkback window describes an error condition. The label shows the error condition, and the text pane shows the most recently sent messages, with those most recently sent appearing first.

In the above walkback window, the label says that the `addWord:to:` message is not understood, while the top line in the text pane shows `WordIndex` as the class of the object which did not understand the message.

Whenever you get a walkback window, you generally do one of three things:

1. You can determine what the problem is from the information contained in the walkback window. In this case, you normally close the walkback window and then go fix the problem.
2. You can determine that the walkback window occurred either as a result of you typing the control and break keys simultaneously, or because a `halt` message was sent. In this case, there is nothing wrong with the program, so you can pop up the pane menu for the walkback window and select `resume`. The walkback window closes and execution continues.
3. You can decide that you need more information, and would like to use the `debugger` to obtain it. In this case, you pop up the pane menu for the walkback window and select `debug`. The walkback window closes and the debugger window opens.

In our case, we probably have enough information in the walkback window to fix the problem. Look at the code for class `WordIndex` using the Class Hierarchy Browser. We defined a method `addWord:for:`, but sent the message `addWord:to:` (in the `addDocument:` method) which was not understood. We used the wrong message!

Correct the `addDocument:` method to use `addWord:for:` instead of `addWord:to:`, and then try again to add the tutorial files to the word index, using the following expressions.

```
Index addDocument: 'chapter.5'.
Index addDocument: 'chapter.6'.
```

Not fixed yet! This time, you get a new walkback window:

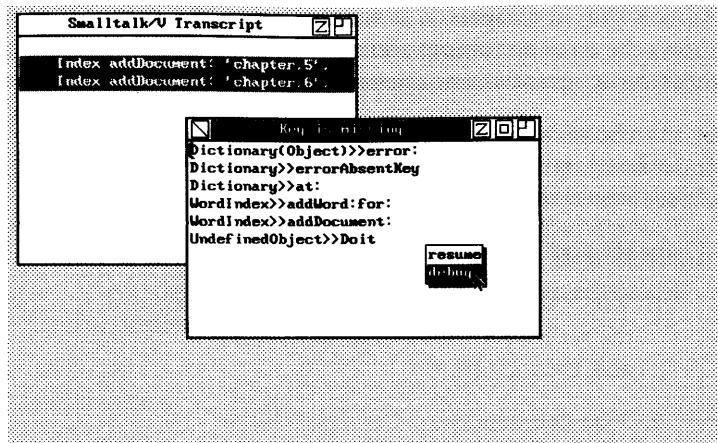


Figure 8.2
Opening
the Debugger

The label of the walkback window says **Key is missing**. Since the problem is not obvious, let's see if we can get some more information by using the debugger. Pop up the walkback window pane menu and select **debug**. You'll see the following debugger window:

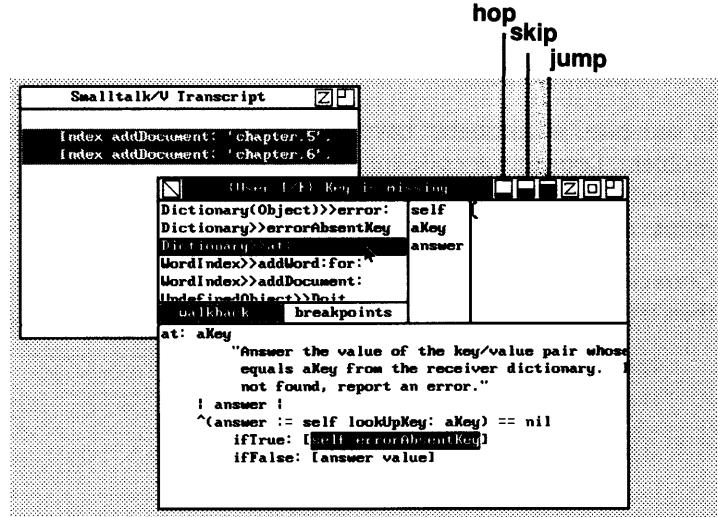


Figure 8.3
Debugger with
Window Buttons

The debugger window gives you an expanded view of the walkback in several panes. The top left pane (a list pane) repeats the walkback information; you can use this pane to select walkback lines. When you select a walkback line, the other panes contain related information. Select the entry containing **Dictionary>>at:**

The bottom pane displays the source code for the selected method, in this case **at:** from class **Dictionary**:

```

at: aKey
| answer |
^ (answer := self lookUpKey: aKey) == nil
  ifTrue: [self errorAbsentKey]
  ifFalse: [answer value]

```

The text that is reversed is the expression currently being evaluated in this method.

As you can see, this method invokes another dictionary method `lookUpKey:`, and then invokes `errorAbsentKey` if the key is missing, which eventually results in the walkback window.

The two panes on the top right are an inspector for the receiver, arguments and temporary variables of the selected method. In this case, you see the receiver `self`, the argument `aKey` and the temporary `answer`. Select `self`; you see that the value is an empty dictionary. Now select `aKey`; the value is the string '`tutorial`', the first word in the file. We tried to do a dictionary lookup on an empty dictionary, `self`, with the first word in the file as key.

Select the line containing `addWord:for:` in the walkback pane on the top left of the window. Now select the argument `wordString`. Again, it's the string '`tutorial`'. We tried to access the `words` dictionary with a key, without first testing whether or not the key is present! Correct the `addWord:for:` method in the bottom pane of the debugger to look as follows:

```

addWord: wordString for: pathName
  "Add wordString to words dictionary for
   document described by pathName."
  (words includesKey: wordString)
    ifFalse: [words at: wordString put: Set new].
  (words at: wordString) add: pathName

```

Now pop up the bottom pane menu and select `save`. Notice what happens. The entries above `addWord:for:` in the walkback list are discarded, because a method they would return to has been changed. The `addWord:for:` method is still selected. Now pop up the menu in the walkback list pane and select `restart`. Execution resumes by re-sending the selected message.

The debugger window disappears, and the method builds the index. With the dictionary now built, let's try to make some queries. Try evaluating the expression below.

```
Index locateDocuments: #'(show' 'class')
```

Another walkback window pops up, indicating that there is another error. Immediately open a debugger window on this new error. You'll see the following window:

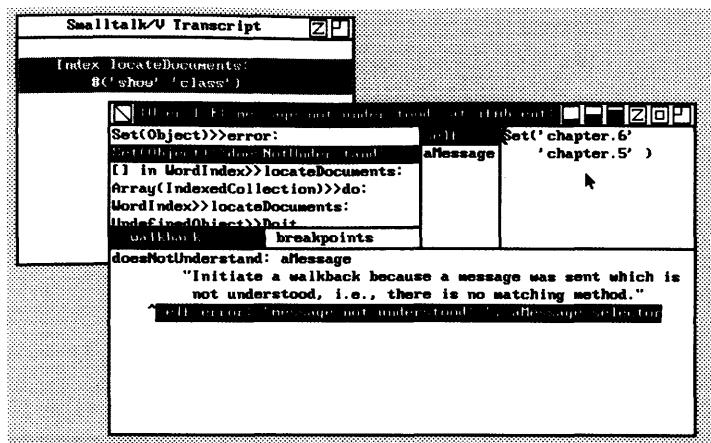


Figure 8.4
Inspecting
Variables

The message `at:ifAbsent:` was sent to an instance of class `Set`, which did not understand it. Select the top walkback line containing `Set(Object)>> doesNotUnderstand:`, and then select `self` in the temporary variable list. The value is:

```
Set('chapter.6' 'chapter.5')
```

Now select the third walkback line, representing a block in the `locateDocuments:` method, and examine the values of the temporary variables. Then look at the source code for the method. The `at:ifAbsent:` message being executed is reversed. It uses instance variable `documents` as receiver. The value printed out for the set above confirms this, because it does contain the document path names.

Let's look at this statement. Either we sent the wrong message to `documents` or `documents` is the wrong receiver. This statement is trying to add to variable `bag` all the documents that include the string contained in variable `word`. The receiver is indeed wrong. This statement should instead use the `words` dictionary:

```
bag addAll:
  (words at: word ifAbsent: [#()])
```

Change the `locateDocuments:` method using the debugger, save it, and restart at `locateDocuments::`. It works! Try the following queries:

```
Index locateDocuments: #'(where' 'the' 'turtle')
Index locateDocuments: #'(each' 'talk')
```

Hop, Skip and Jump

Now that you have class `WordIndex` debugged, let's see how you can use the debugger to learn how an application operates by watching it send messages. Open a debugger window to step through execution of the query you just performed by evaluating the following expression:

`self halt.`

`Index locateDocuments: #(each talk)`

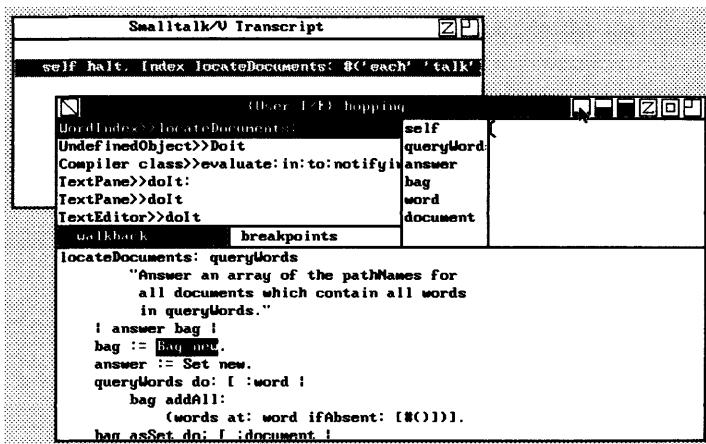


Figure 8.5
Debugging
an Expression

There are 6 buttons on the right side of a debugger window label bar as seen in Figure 8.5. The first three called **hop**, **skip** and **jump** are related to debugging. **Hop**, **skip** and **jump** each cause limited program execution. **Hop** executes the least amount: one Smalltalk message send or assignment statement. **skip** executes more than **hop**: up to the next message send or assignment in the current method or up to the next breakpoint, whichever comes first. (Refer to Chapter 16 for a description of breakpoints). **Jump** executes more than **skip**: up to the next breakpoint or the end of the debugged expression.

Try selecting the **hop** button twice and watch the debugger window. Execution state is now at the beginning of execution of the expression, shown in Figure 8.5. Select **hop** again. Notice how execution proceeds in small amounts, with the next statement to be executed highlighted after the step. You can examine the state of objects after each **hop**.

Now try selecting the **skip** button a few times. Notice that the highlighting stays within the same method until the method finishes execution. This allows you to concentrate on a single method activation and ignore lower level messages.

What You've Now Learned

By the end of this tutorial, you should be familiar with walkback windows and how to use the debugger. If you want to review, you can either repeat the tutorial, or refer to the detailed description in **Part 3, The Smalltalk/V 286 Reference**.

As always, if you exit **Smalltalk/V** before beginning the next tutorial, be sure to save the image.

9 GRAPHICS

In these tutorials, you have seen some of Smalltalk/V's remarkable graphics. In this tutorial, you will learn how Smalltalk/V produces such graphics.

The examples in this chapter repeatedly alter the top half of the screen. For this reason, you will want to open any windows *in the bottom half of the screen only*. After each example, you can restore the screen to its previous state by selecting **redraw screen** from the system menu. As always, you can find the sample code for this tutorial in the disk file **chapter.9**. Use the Disk Browser to retrieve these examples from this file. Since this file is larger than 10,000 bytes, only the beginning and end of it can be seen in the text pane. Pop up the text pane menu and select **read** it to make the entire file accessible.

You will again be adding new methods and classes to the environment in this tutorial, so be sure to save the image when you exit Smalltalk/V.

Some Basic Concepts

Smalltalk/V owes its graphical capability to *bit-mapped* graphics (also called *raster* graphics). A line is drawn with a continuous vector of dots. A cursor is formed with a rectangle of black and white dots. Even a character is formed with a block of dots, instead of an ASCII value.

These dots are displayed on a monitor screen as colored pixels. They are stored internally as a *Bitmap*, contained in a *Form*. A Bitmap is a matrix of bits, with a value 1 representing white and 0 representing black. To refer to an individual dot within a Bitmap, you use *Points*. To move a group of dots from one place to another (either within the same Bitmap or between different Bitmaps), you use *Rectangles* to denote the areas involved. Thus **Point**, **Rectangle**, and **Form** are Smalltalk/V's basic graphic data structures.

Point

A Point refers to a position within a two dimensional array. It has two instance variables: **x**, the column coordinate, and **y**, the row coordinate. To create a **Point**, you use the binary message **@**. For example, the expression:

5 @ 10

creates a **Point** referencing column 5 and row 10. Evaluate the following expressions:

(5 @ 10) x
(5 @ 10) y

These expressions return the values 5 and 10, respectively. You can also add, subtract, multiply, divide, or compare Points, as in these examples:

```
(1 @ 2) + (-1 @ -2)
(1 @ 2) - (1 @ 2)
(1 @ 2) * (1/2 @ (1/2))
(1 @ 2) < (3 @ 4)
(3 @ 5) > (3 @ 4)
```

These expressions combine or compare **x** of the receiver with **x** of the argument and **y** of the receiver with **y** of the argument. This is why the last expression returns **false**; the first **x**, 3, is not greater than the second, also 3.

You can mix a Point with a scalar:

```
1 @ 2 + -1
1 @ 2 // 2
```

which applies the scalar to both **x** and **y** of the Point. You cannot, however, compare point coordinates and a scalar. For example, try evaluating this expression:

```
1 @ 2 < 3
```

To alter one of the two coordinates, simply use the messages **x:** and **y:**. For example, evaluate the following expressions as a group:

```
| aPoint |
aPoint := (5 @10).
aPoint x: 1.
aPoint y: 2.
^aPoint
```

Rectangle

A *Rectangle* is represented by two points: an *origin* (the top left point) and a *corner* (the bottom right point). With this information, Smalltalk can determine its extent (the width and height of the elements contained within the rectangle) as:

corner - origin

To create a Rectangle, you normally send messages to a point, as in this example:

```
1 @ 1 corner: 100 @ 100
```

Or equivalently:

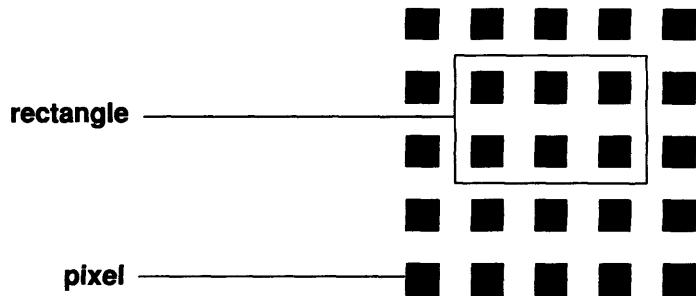
```
1 @ 1 extent: 99 @ 99
```

A Rectangle includes the bits *inside* of the rectangle. The rectangle itself is imposed on gaps between bits. For example, the Rectangle

1 @ 1 corner: 4 @ 3

contains 6 bits (3 horizontal and 2 vertical) as illustrated below:

Figure 9.1
Pixels and
Rectangles



There are many operations you can perform on Rectangles. For example, try evaluating each of the following:

```
(0 @ 0 extent: 100 @ 100) center
(0 @ 0 extent: 100 @ 100) insetBy: 10
(-5 @ -10 extent: 20 @ 20)
    intersect: (1 @ 2 extent: 20 @ 20)
(-5 @ -10 extent: 20 @ 20)
    containsPoint: 0 @ 0
```

These operations are not the focus of this tutorial, however; refer to Part 3, The Smalltalk/V 286 Reference, for more detailed descriptions.

Form

As we said before, a Point or a Rectangle simply refers to a position or an area of positions. The object that actually holds the graphical image is the *Form*. A Form has many instance variables, but only three concern you: **bits**, **width**, and **height**.

The variable **bits** contains a **Bitmap**, which is the content of the **Form**. This **Bitmap** contains bits for the area represented by a **Rectangle**:

0 @ 0 extent: (width @ height)

So, for example, to create a **Form** containing all 1 bits, evaluate the following expression:

F := Form width: 100 height: 50

To display a **Form**, use the message **displayAt:** with a point for the argument. For example, to display the rectangle created above, you would use:

F displayAt: 0 @ 0

This displays a white rectangle at the top left corner of the screen. You can also display only a portion of a **Form**. For example, evaluate the following expression:

```
| black white aRect |
white := Form width: Display width height: Display height.
black := (Form width: Display width height: Display height)
reverse.
aRect := Display boundingBox. "screen rectangle"
Display height // 2 // 8 timesRepeat: [
    black displayAt: 0 @ 0 clippingBox: aRect.
    aRect := aRect insetBy: 4.
    white displayAt: 0 @ 0 clippingBox: aRect.
    aRect := aRect insetBy: 4].
Menu message: 'continue'.
Scheduler systemDispatcher redraw
```

The message **displayAt:clippingBox:** restricts the area on the screen to be changed. (You'll learn more about clipping boxes later in this tutorial.) The last two lines pop up a menu and then redraw the screen.

You can also display a **Form** on a printer by evaluating the following expression:

(Form new width: 100 height: 100 initialByte: 16rF0)
outputToPrinter

Class **Form** has three immediate subclasses: **BiColorForm**, **ColorForm**, and **DisplayScreen**.

A **BiColorForm** allows a foreground color to be assigned to the 1 pixels, and a background color to the 0 pixels in the bitmap. This allows a single bitmap to represent two colors other than just black and white.

A **ColorForm** has an array of bitmaps. The bits in the same position of each bitmap collectively represent the color of the pixel at that position.

Class **DisplayScreen** and its subclass **ColorScreen** represents a monochrome screen and a color screen respectively. Their bitmaps have fixed physical addresses, as well as fixed sizes, dictated by the mode of the graphics adapter being used. A global variable, **Display**, contains an instance of either **ColorScreen** or **DisplayScreen** depending on whether or not your monitor and graphics adaptor supports color.

The Basic Class of Graphics: BitBlt

BitBlt ("bit block transfer") is the fundamental class of all **Smalltalk/V** graphics. Classes like **Pen** (for drawing) and **CharacterScanner** (for writing text) are subclasses of **BitBlt**.

The basic function of **BitBlt** is to move a rectangular area of bits from one portion of a **Form** or **DisplayScreen** to another. The simplest form of the move requires a source form, a destination form, a rectangle on the source form to be moved, and an origin point on the destination form. Smalltalk can then calculate the corner of the destination rectangle by adding the source rectangle extent to the destination origin.

For example, consider the following code, which copies the top left quarter of the screen to the right:

```
(BitBlt destForm: Display sourceForm: Display)
    sourceRect: (0 @ 0 extent: Display extent // 2);
    destOrigin: (Display width // 2 @ 0);
    copyBits
```

The first line creates a **BitBlt** instance with **Display** as both the source and destination form. The second line specifies the top left quarter of **Display** as the source rectangle. The third line specifies the top center point as the destination origin. The last line moves the bits.

Let's look at another example. This code copies a white form in its entirety to **Display**, while storing the old contents of the screen in another form, **F**:

```
F := Display compatibleForm new extent: Display extent // 2.
(BitBlt destForm: F sourceForm: Display) copyBits.
(BitBlt destForm: Display sourceForm:
    (Form new extent: F extent)) copyBits.
```

The first line creates a form, **F**, with a size one quarter of the display screen. **Display compatibleForm** returns either class **Form** or class **ColorForm** depending on whether **Display** is an instance of class **DisplayScreen** or **ColorScreen**, respectively. The second line copies the top left quarter of the screen onto **F**. When, as in this example, you do not specify a source rectangle and destination origin, the **destForm:sourceForm:** message uses the entire area of the source form and point **0 @ 0** as the source rectangle and destination origin, respectively.

In this copy operation, **BitBlt** copies a larger rectangle onto a smaller form. This poses no problem, however; **BitBlt** never touches any bits beyond those contained in the destination form.

The last two lines create a new white form with the same size as **F**, and then copies it onto the screen. Here, we move a smaller form to a larger one. Again, **BitBlt** moves only as many bits as are contained in the source form. Now evaluate the above expressions; the upper left corner is blanked. But, since the old contents are stored in **F**, you can restore the old screen by evaluating:

(BitBlt destForm: Display sourceForm: F) copyBits

The rest of this section describes the concepts of **BitBlt** with color.

When bits are moved from a **BiColorForm** to either a **ColorForm** or **ColorScreen**, the bits of value 1 in the **BiColorForm** become the foreground color specified by the mask form and the bits of value 0 become the background color. Then these color pixels are moved to the destination with four bits of each color pixel occupying the same position on the four destination bitmaps. Try the following:

```
| aForm |
aForm := BiColorForm width: 100 height: 100.
(BitBlt destForm: aForm sourceForm: nil)
    mask: Form black;
    extent: 50 @ 100;
    copyBits;
    destForm: Display;
    sourceForm: aForm;
    mask: (BiColorForm foreColor: 1 backColor: 4);
    extent: 100 @ 100;
    copyBits
```

This example first initializes **aForm** to be a **BiColorForm** with all bits set to 1. Then it copies zeros into the bits comprising the left half of **aForm**. Finally it copies **aForm** to the display screen assigning color 1 (blue) to 1 bits and color 4 (red) to 0 bits. The result shown on the screen is a 100 x 100 pixel block with the left half in red and the right half in blue.

When bits are moved between a **ColorForm** and a **ColorScreen**, bits are moved between corresponding bitmaps. In other words, this can be viewed as four separate moves between corresponding single bitmaps. Try the following:

```
| aForm |
aForm := ColorForm width: 100 height: 100.
(BitBlt destForm: aForm sourceForm: Display)
    copyBits;
    destForm: Display;
    sourceForm: aForm;
    mask: Form gray;
    copyBits
```

This example copies a block from **DisplayScreen** to a **ColorForm** and then copies it back to the screen with a gray halftone.

When bits are moved from a **ColorForm** or **ColorScreen** to a **BiColorForm**, the colors on the source bitmaps that are the same as the foreground color of the mask form are turned into 1 bits on the destination **BiColorForm** and the rest of the bits are set to zero. For example,

```
| aForm |
aForm := BiColorForm
    width: Display width
    height: Display height.
(BitBlt destForm: aForm sourceForm: Display)
    mask: (BiColorForm color: 11);
    copyBits;
    destForm: Display;
    sourceForm: aForm;
    copyBits
```

extracts the area with color 11 from the screen to **aForm**, and blacks out the rest when **aForm** is copied back to the screen.

When the source form is nil, if the destination is a **BiColorForm**, then bits in the mask form will be tiled over the destination. If the destination object has multiple bitmaps then 1 bits in the mask form will take on the foreground color of the mask form, 0 bits the background color, and the colors will be tiled over the destination area. For example,

```
(BitBlt destForm: Display sourceForm: nil)
    mask: (BiColorForm color: 4);
    destRect: (0 @ 0 extent: 100 @ 100);
    copyBits
```

paints a red rectangle in the top left corner of the screen.

Halftone (Mask)

A *halftone*, or *mask*, is a **Form** which combines with the source form to create the effect of gray tone. This mask Form is restricted to have a width and height of 16. To copy a mask over a larger area, **BitBlt** repeatedly applies (tiles) the content throughout the entire affected area. Since a bit value 1 represents white and 0 represents black, a white mask form contains all 1 bits while a black one contains all 0 bits. There are four possible ways to combine source and mask forms; evaluate each of the following expressions to see the results:

"no source, no halftone (displays solid white)"

```
(BitBlt destForm: Display sourceForm: nil)
    destRect: (0 @ 0 extent: 100 @ 100);
    copyBits
```

"halftone only (gives halftone tiling)"

```
(BitBlt destForm: Display sourceForm: nil)
    mask: Form gray;
    destRect: (0 @ 0 extent: 100 @ 100);
    copyBits
```

In this case, the message **Form gray** returns a prebuilt mask form which yields a gray tone effect. Other prebuilt mask forms can be obtained by sending the message **black**, **darkGray**, **gray**, **lightGray**, or **white** to class **Form**.

"source only (gives source bits)"

```
(BitBlt destForm: Display sourceForm:
    (Form new width: 100 height: 100 initialByte: 16rF0))
    copyBits
```

Where the **initialByte** argument, 16rF0, specifies the initial value for all bytes in the new Form. The result will be white and black vertical strips (each four pixels wide).

"both specified (bits in mask are ANDed with bit in source)"

```
(BitBlt destForm: Display sourceForm:
    (Form new width: 100 height: 100 initialByte: 16rF0))
    mask: Form darkGray;
    copyBits
```

When a mask having colors other than black and white is desired, a **BiColorForm** mask should be used to control which color is painted on the screen or on a **ColorForm**. The following messages are often used to create such a mask form:

BiColorForm color: aColor

Creates a 16 x 16 **BiColorForm** with all 1 bits, with **aColor** as its foreground color and 0 (black) as its background color.

BiColorForm foreColor: fColor backColor: bColor

Creates a 16 x 16 BiColorForm with all 1 bits, fColor as its foreground color and bColor as its background color.

BiColorForm gray foreColor: fColor backColor: bColor

Creates a 16 x 16 BiColorForm with alternating 0 and 1 bits, with fColor as its foreground color and bColor as its background color.

This last mask form can be used to mix two colors together. For example,

```
(BitBlt destForm: Display sourceForm: nil)
mask: (BiColorForm gray foreColor: 1 backColor: 4);
extent: 100 @ 100;
copyBits
```

paints a purple block (mixing blue and red) on the screen. By using this technique, you can expand the number of simultaneously displayable colors from 16 to 120.

Combination Rules

*falls auch lightGray, darkGray
376, 377
376, 377*

A *combination rule* is an Integer which specifies how the source form bits (after being merged with the mask form) are combined with the destination form bits. Since a bit value of 1 represents white and a bit value of 0 represents black, if you OR white with black, the result is white. If you AND them together, the result is black. To specify what you want to happen in different situations, send a message to class **Form**:

Form over	destination becomes source
Form orRule	source OR into destination
Form andRule	source AND into destination
Form under	source AND into destination
Form erase	if source is 1 then destination becomes 0
Form reverse	source XOR into destination
Form orThru	first erase without specifying mask form, then OR with mask form specified

All the examples we've used so far in this chapter have assumed the combination rule **over**. To see the effect of different combination rules, evaluate the following expression, which repeatedly displays the number "8" on a background of white with a black band in the middle, each time with a different combination rule:

```
'8' displayAt: 0 @ 0 font: Font fourteenLine.
F := (Form width: 8 height: 14) fromDisplay.
F := (F magnify: (0 @ 0 extent: 8 @ 14) by: 7 @ 7).
Display white: (0 @ 0 extent: 640 @ 104);
black: (0 @ 24 extent: 640 @ 30).
(BitBlt destForm: Display sourceForm: F)
mask: Form lightGray;
combinationRule: Form over;
copyBits;
destOrigin: 70 @ 0;
combinationRule: Form orRule;
copyBits;
destOrigin: 140 @ 0;
combinationRule: Form under;
copyBits;
destOrigin: 210 @ 0;
combinationRule: Form erase;
copyBits;
destOrigin: 280 @ 0;
combinationRule: Form reverse;
copyBits;
destOrigin: 350 @ 0;
combinationRule: Form orThru;
copyBits
```

Another interesting example is to swap the left half of the display screen with the right half, without using an intermediate Form:

```
| aRectangle |
aRectangle := Display width // 2 @ 0 corner: Display extent.
(BitBlt destForm: Display sourceForm: Display)
combinationRule: Form reverse;
destRect: aRectangle;
copyBits;
sourceRect: aRectangle;
destOrigin: 0 @ 0;
copyBits;
sourceOrigin: 0 @ 0;
destRect: aRectangle;
copyBits.
Menu message: 'continue'.
Scheduler systemDispatcher redraw
```

There is an extra dimension in dealing with multi-bitmap forms. With a single-bitmap form, pixels are represented by binary numbers since they can assume only two colors: black (0) and white (1). Thus when the 8 pixels 10101010 in the source are OR'd into the 8 pixels 11110000 in the destination, the destination becomes 11111010 (5 black pixels followed by white, black and white). A color screen or form, however, has four bitmaps. Each pixel must be represented by a hexadecimal number. For example, when the 8 pixels E0E0E0E0 are OR'd into 33330000 you get F3F3E0E0. Following are some examples dealing with multi-colored forms. They will not work if your machine does not support color.

The over rule is used to copy the source rectangle over the destination rectangle regardless of which colors are contained in the destination area. When you create a BitBlt instance, it defaults to this rule. For example:

```
(BitBlt destForm: Display sourceForm: Display)
    destOrigin: (Display extent // 2);
    copyBits
```

The opaque rule (called orThru for black and white) first blanks (zeroes) out the destination area corresponding to the 1 bits of the source area, then combines the source area with the repeated bits of the mask form by logical AND, and finally ORs the result to the destination area. In short, the 0 bits of the source form will appear to be transparent after the move. For example:

```
| aForm |
aForm := (BiColorForm width: 100 height: 100)
foreColor: 5.
(BitBlt destForm: aForm sourceForm: nil)
mask: Form black;
destRect: (20 @ 20 extent: 60 @ 60);
copyBits;
destForm: Display;
sourceForm: aForm;
mask: nil;
destRect: Display boundingBox;
combinationRule: Form opaque;
copyBits
```

first makes an inner rectangle of 0 bits in aForm and then copies aForm to the screen. You can see that the portion of 0 bits is transparent. This rule also has many other uses. For example:

```
| aForm |
(BitBlt
destForm:
  (aForm := BiColorForm new extent: Display extent)
sourceForm: Display)
  mask: (BiColorForm color: 5; "extract blue color"
copyBits.
aForm foreColor: 4. "change foreground color to red"
(BitBlt
destForm: Display
sourceForm: aForm)
  combinationRule: Form opaque; "change only 1 bits"
copyBits
```

changes blue to red on your screen.

The **orRule** merges two colors together. For example, combining blue (1) with green (2) yields cyan (3). Try the following:

```
| aForm |
aForm := (BiColorForm width: 100 height: 100)
  foreColor: 1. "a blue single-bitmap form"
aForm displayAt: 0 @ 0.
aForm foreColor: 2.
(BitBlt destForm: Display sourceForm: aForm)
  combinationRule: Form orRule;
copyBits
```

The **andRule** is usually used to extract a base color. For example:

```
| aForm |
aForm := (BiColorForm width: Display width height: Display height)
  foreColor: 1. "a blue single-bitmap form"
(BitBlt destForm: Display sourceForm: aForm)
  combinationRule: Form andRule;
copyBits
```

causes any pixel in the destination rectangle having an odd numbered color (a trace of blue) to become blue and all other pixels to become black.

The **reverseRule**, as its name implies, is usually used to reverse the destination color, for example:

```
(BitBlt destForm: Display sourceForm: nil)
    combinationRule: Form reverse;
    extent: 100 @ 100;
    copyBits
```

reverses the color in the top left corner of your screen.

Clipping Rectangle

BitBlt lets you use *clipping rectangles* to restrict your bit transfer to a designated rectangle on the destination Form. For example, each pane within a window sets up a clipping rectangle every time the window is opened or moved or reframed; you never have to worry about writing or drawing beyond that rectangle.

For example, look at the following code, which expands all the black areas in your active window to the left by one pixel. Thus black characters in the window will appear to be bolded, while white ones appear to be thinned:

```
(BitBlt destForm: Display sourceForm: Display)
    clipRect: Scheduler topDispatcher pane frame;
    combinationRule: Form andRule;
    destOrigin: -1 @ 0;
    copyBits
```

In the example, Scheduler topDispatcher returns the Dispatcher controlling the active window, the pane message returns the pane associated with the Dispatcher, and the frame message asks its receiver pane to answer the rectangle surrounding the active window. (*Dispatchers* control keyboard and mouse input; they are described in detail in Part 3.) This final rectangle is used as the clipping area. Now evaluate the expression; you'll see that only the content of the active window is affected, even though the entire screen is copied.

You should note two things from the above example. First, this example copies a form to itself, overlapping a large portion of the source and destination rectangles. BitBlt handles this situation properly as if a copy of the source form was made before transferring the bits.

Second, both the source and destination origin can be somewhere outside their forms. The only restriction is that the coordinates of origin points must be small integers. The final rectangular area affected by the BitBlt is the intersection of the destination form, source form (with the source origin aligned with the destination origin), source rectangle, destination rectangle, and clipping rectangle.

Extension of BitBlt

You've now seen some of BitBlt's raw power. It still, however, needs to be extended to provide an easier user interface for handling tasks like drawing lines or displaying characters. Smalltalk/V therefore provides four BitBlt subclasses: CharacterScanner, Pen, Commander, and Animation.

CharacterScanner

Class CharacterScanner converts a character's ASCII representation into its graphical, readable form. For example, a message such as:

```
'Hello' displayAt: 0 @ 0
```

actually creates a CharacterScanner and tells it to display the String 'Hello' at screen location 0 @ 0.

One of CharacterScanner's major additions to BitBlt is an instance variable containing the font to use when displaying characters. It is an instance of class Font, containing a particular version of the bitmap representation of all characters, which provides information about how to retrieve each character. When a CharacterScanner is told to display a string, it literally uses the ASCII value of each character in the string as an index, then uses its own BitBlt to copy the bitmap pointed to by the index onto the destination form. For example:

```
CharacterScanner new
    initialize: Display boundingBox
        font: Font eightLine;
    display: 'Hello' at: 0 @ 0;
    setFont: Font fourteenLine;
    display: 'Hello' at: 40 @ 0
```

displays Hello at the top left corner of the screen using a font 8 lines high, switches to a font of 14 lines, and displays the same string next to the previous one.

You can use the setForeColor:backColor: message to set the foreground and background colors for the characters to be displayed:

```
CharacterScanner new
    initialize: Display boundingBox
        font: SysFont
        dest: Display;
    setForeColor: (Display compatibleMask color: 1)
        backColor: (Display compatibleMask color: 14)
    display: 'abc' at: 0 @ 0. "display on screen"
```

In a monochrome system, the colors numbered 0-7 are mapped into black and colors numbered 8-15 are mapped into white.

Pen

In the previous chapters, you have seen many examples of turtle graphics. Although you may not have known it, you were using an instance of class **Pen**. When you tell a pen to draw a line from one place to another, the pen actually uses its **BitBlt** to copy from its source form to its destination form at each position along the line. For example, to draw from `0 @ 0` to `9 @ 0`, the pen copies from its source form to its destination form 10 times, starting at `0 @ 0` and moving right by one pixel for each of the successive copies. The end result looks like a straight horizontal line. When you tell a pen to draw diagonally, it moves to successive positions in as straight a line as possible. The algorithm also makes sure that there are no gaps in the line.

Because a pen uses its source form to draw, changing the shape or tip size of a pen simply means changing its source form. To change the color of a pen, use a mask form with the desired gray tone. Of course, you can also change the combination rule and clipping rectangle to get different effects.

A pen always remembers its current *location*, *direction*, and *downState*. Location tells it where to start for the next movement. Direction allows it to calculate the ending point when the `go:` message is used, where only units of movement are specified. The angle is expressed in degrees, with east equal to 0 and north equal to 270. If *downState* is true, the pen draws while it moves; otherwise, it moves without drawing. The mandala drawing method illustrates this:

```

" Pen method "
mandala: sides diameter: diameter
| vertices radius center angle color |
"initialize local variables"
center := self location.
vertices := Array new: sides.
radius := diameter // 2.
angle := 360 // sides.
"set down state to false in order to use Pen to
locate vertices without drawing"
self direction: 270; up.
1 to: sides do: [ :i |
    self go: radius. "move to next new vertex"
    vertices at: i put: self location. "remember it"
    self place: center; "change location to center"
    turn: angle]. "increase direction by angle"
"draw from each vertex to every later one"
self down.
color := 1.
1 to: sides - 1 do: [ :j |
    j + 1 to: sides do: [ :i |
        halftone foreColor:
            (color := 5 - color).
        self place: (vertices at: j);
        goto: (vertices at: i)]]
```

To execute this method, evaluate the following expression:

Pen new mandala: 30 diameter: 300

When you send the message **go:** to a pen, be aware that the actual pixels drawn may not be the same as you specified. This is because the number of pixels drawn are adjusted by a global variable, **Aspect**, which describes the aspect ratio of your display. **Aspect** contains a fractional number, determined by the video adapter board your system uses. The **go:** message multiplies the vertical distance of its argument by **Aspect** to yield the real number of pixels to draw. Because of this adjustment, you are able to draw squares and circles without worrying about your screen's aspect ratio. To find the correct **Aspect** to use with your video adapter board, evaluate the following expression:

Display black: (2 @ 2 extent: 150 @ 150)

With a ruler, measure the width and height of the black rectangle; then set:

Aspect := widthMeasured / heightMeasured

Aspect must be an integer or a fraction, not a floating point. For example, if the black rectangle measured 1 3/4" by 2 1/4" you would set:

Aspect := (7/4) / (9/4)

The Network Example Revisited

With all the graphics techniques we have learned so far, we can now actually draw the network system you saw in Chapter 7.

In class **NetworkNode**, add the following methods to draw the node itself:

```

draw
    "Draw the receiver node with a
    circle around its name."
| font r aspect pen |
font := Font eightLine.
(pen := Pen new)
    defaultNib: 2;
    mask: Form white.
pen place: position.
pen solidEllipse: (r := name size * font width + 15 // 2)
    aspect: (aspect := font height + 16 / r / 2 / Aspect).
pen mask: Form black.
pen centerText: name font: font.
pen ellipse: r aspect: aspect

position
    "Answer the position of the receiver node."
^position

```

The method **draw** draws a solid white ellipse large enough to contain the node name, displays the text for the name in black, and finally redraws the border of the ellipse in black. The method **position** answers the position of the node.

Then, in class **Network**, add another **draw** method to first draw arcs and nodes:

```

draw
    "Draw the network. For each node, it draws
    all the arcs and then the node. All the
    nodes visited are remembered to avoid double
    drawing."
| visited pen |
pen :=Pen new
    defaultNib: 4 @ 3.
visited := Set new.
connections keys do: [ :nodeA |
    visited add: nodeA.
    (connections at: nodeA) do: [ :nodeB |
        (visited includes: nodeB)
        ifFalse: [
            pen place: nodeA position;
            goto: nodeB position]].]
nodeA draw]
```

In Chapter 7 you assigned a network of nodes to the global variable **Net**. Display the network by evaluating all of the following:

```

Display white.
Net draw.
Menu message: 'continue'.
Scheduler systemDispatcher redraw
```

The result should be as follows.

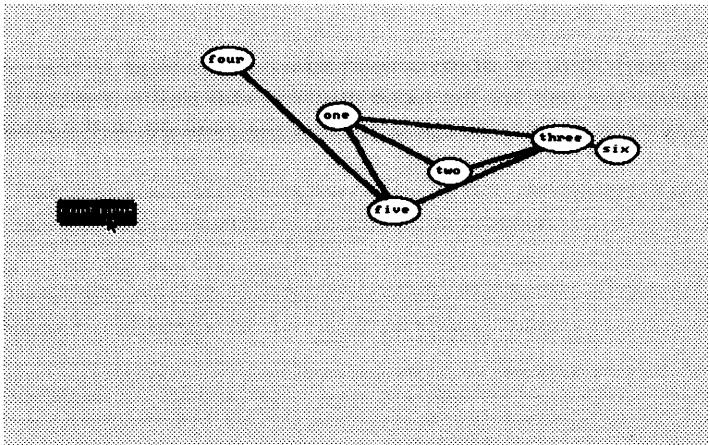


Figure 9.2
Displaying
a Network

Commander

Commander is a subclass of **Pen**. A **Commander** commands an array of pens. It has messages to either fan out or line up all the pens under its command. It also reimplements messages like **place:**, **turn:**, **down**, **up**, **go:**, and **goto:** to pass the message to all of its pens. When the units of movement are small, this creates an illusion of all pens drawing simultaneously. For example, evaluate the following example, which draws five dragons fanning out:

```
Display white; Display boundingBox. "blank screen"
(Commander new: 5)
    fanOut; "set fan out direction"
        up;
        go: 60; "set starting point"
        down;
        dragon: 9. "draw dragon"
Menu message: 'continue'.
Scheduler systemDispatcher redraw
```

The result should be as follows.

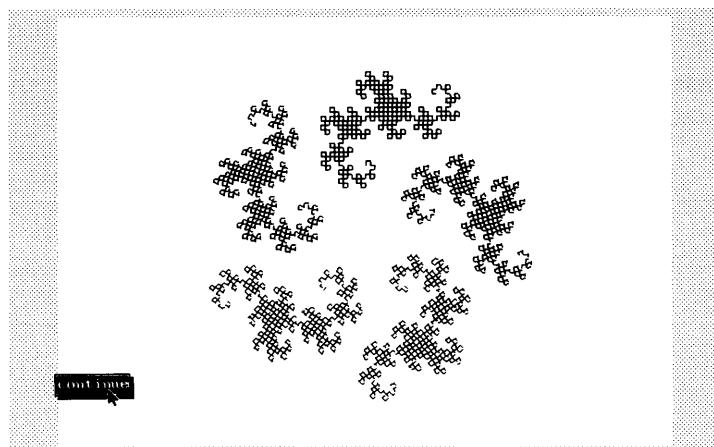


Figure 9.3
Dragon Curves

Animation

Animation is another subclass of **Pen**. An **Animation** contains several collections of images of objects in motion. Each collection of images (each image is an instance of class **Form**) represents the motion of a single object as a series of still frames. When the forms are displayed continuously, it creates the illusion of a moving object, as in a cartoon. You can send messages to an **Animation** to tell any of its objects to move, or change its position or direction.

Movement in class **Animation** differs from the one in **Pen** in that an animation always erases the old image of an object before moving it to a new position. The following example illustrates four walking dogs. The pictures of these dogs were drawn using the **FreeDrawing** tool included on your **Smalltalk/V** diskettes. Evaluate these expressions to create an animation called **Animator**, with four dogs under your command:

```
| dogImages |
dogImages := 
    Array "build an Array of pictures of a walking dog"
        with: (GraphDictionary at: 'dog1')
        with: (GraphDictionary at: 'dog2')
        with: (GraphDictionary at: 'dog3')
        with: (GraphDictionary at: 'dog2').
Animator := Animation new
    initialize: Display boundingBox. "init clipping rectangle"
Animator add: dogImages "add first dog"
    name: 'Snoopy' " with a name Snoopy"
    color: #lightGray. " and a lightGray color"
Animator add: dogImages
    name: 'Lassie'
    color: #black.
Animator add: dogImages
    name: 'Bow'
    color: #darkGray.
Animator add: dogImages
    name: 'Wow'
    color: #white.
```

Now evaluate the following expressions, to issue commands to your dogs:

"issue commands"

Display gray. "make whole screen gray"

Animator

speed: 16; "each picture is displayed 8 pixels apart"

shiftRate: 2; "display picture twice before going

to the next (slows down the leg and tail motion)"

setBackground; "use current screen as background"

tell: 'Snoopy' place: 0@0;

tell: 'Snoopy' direction: 45;

tell: 'Lassie' place:

0 @ (Display height - 60);

tell: 'Lassie' direction: -45;

tell: 'Bow' place:

(Display width - 100) @ 0;

tell: 'Bow' direction: 135;

tell: 'Wow' place:

Display extent - (100 @ 60);

tell: 'Wow' direction: -135;

tell: 'Snoopy' go: 350;

tell: 'Lassie' go: 350;

tell: 'Bow' go: 350;

tell: 'Wow' go: 350.

Menu message: 'continue'.

Scheduler systemDispatcher redraw

The above expressions command the four dogs to move towards the center of the screen.
The following commands cause Snoopy to bounce around the screen at a quicker pace:

Display gray. "make whole screen gray"

Animator

speed: 24;

shiftRate: 1;

tell: 'Snoopy' bounce: 4000.

Menu message: 'continue'.

Scheduler systemDispatcher redraw

The following commands make all four dogs bounce together.

```
80 timesRepeat: [
    Animator
        tell: 'Lassie' bounce: 16;
        tell: 'Bow' bounce: 16;
        tell: 'Wow' bounce: 16;
        tell: 'Snoopy' bounce: 16].
Menu message: 'continue'.
Scheduler systemDispatcher redraw
```

Global variables like **Animator** have varying amounts of memory assigned to them. To free this space after you have finished exploring the **Animator** variable here, type the following and evaluate it:

```
Animator := nil. "get rid of the instance"
```

This will set the global variable to **nil**, the undefined object, freeing up space as you continue.

What You've Now Learned

By the end of this tutorial, you should be familiar with:

- **Point**, **Rectangle**, **Form**, and **Bitmap**
- **BitBlt**
- **Halftone** (mask)
- **Combination Rules**
- **Clipping rectangles**
- **BitBlt** subclasses **CharacterScanner**, **Pen**, **Commander**, and **animation**

As always, you can review any of these topics by repeating the corresponding section of the tutorial, or by referring to a detailed description in **Part 3, The Smalltalk/V 286 Reference**.

If you are going to exit **Smalltalk/V** before proceeding on to the next tutorial, be sure to save the image.

10 WINDOWS

As you've seen throughout these tutorials, windows provide the major interface between you and **Smalltalk/V**. For example, you use the Class Hierarchy Browser window to enter programs into the system, and the Disk Browser window to browse and manipulate files. In the previous chapters, we used standard, supplied windows for the tutorial examples. In this chapter, you'll find out how to make your own windows.

As always, the examples for this tutorial are stored in a disk file. To retrieve these examples, simply use the Disk Browser to load the contents of the file **chapter.10**.

You will also again be making modifications to the **Smalltalk/V** environment in this tutorial. Be sure to save the image when you exit the environment; if you want to repeat any section of this or any tutorial, it will already be there for you.

This tutorial builds on several of the previous tutorial examples in chapters 6, 7, and 9. If you have not done the tutorials in those chapters and saved the image, you should evaluate the following expression to install the needed classes:

(File pathName: 'class10.st') fileIn

The later part of this tutorial extends the animal and animal habitat classes with several new methods. Evaluate the following expression to file in all of the new methods:

(File pathName: 'animal10.st') fileIn

We will explain the new methods as they are used below.

The Promter

Promters are a special kind of window which lets you ask a question and wait for a single response. For example, evaluate the following expression:

**Promter prompt: 'Do you know Smalltalk/V?'
default: 'Yes, I''ve done a tutorial.'**

A window pops up with the **prompt**: argument as the window label, and the **default**: argument shown below it. The line containing the default is a standard text pane, which means that its contents can be edited. The user can either accept the response, or edit it. When you press the **return** or **enter** key, or select **accept** on the text pane menu, the promter accepts your answer and displays it as the result of **show** it.

A unique characteristic of a promter is that as long as it lives, no other windows can be activated. You must either accept or cancel the answer (which closes the promter window) before you can activate another window.

Single Pane Window

Other windows in Smalltalk/V are not as restrictive as prompters. They are more under user, rather than program, control. For instance, often a window may ask a question which you cannot answer without first consulting another window. You would then simply select or open another window, get your answer, then switch back to the previous window and enter the answer.

Let's start with a window with only a single text pane. Evaluate the following expression:

```
LearnDispatcher :=
TextEditor
windowLabeled: 'Learning Status'
frame: (0 @ 0 extent: 400 @ 100).
```

TextEditor is one of the **Dispatcher** classes whose instance variables, when paired with a **TextPane**, provide text editing capabilities. Sending the **windowLabeled:frame:** message to it creates a window occupying a rectangular area (0 @ 0 extent: 400 @ 100) on the screen, with the specified label on the top.

To write text from your current window into the text pane of this new window, evaluate the following expression:

```
LearnDispatcher
nextPutAll: 'I have learned everything about Smalltalk.';
cr.
```

LearnDispatcher is used like a **Stream**, which you saw in Chapter 7. As you recall, the message **nextPutAll:** adds its **String** argument to the end of the receiver contents. The message **cr** then adds a line feed. For a **TextEditor**, **cr** also displays the buffered **nextPutAll:** strings immediately; without it, the display will be delayed until the next line feed is received.

You can now move the cursor over the new window and select it, which activates the window and lets you edit the text in it.

To retrieve the contents of this window from another window, activate a different window and evaluate the expression:

LearnDispatcher **contents.**

To close the window, select **close** from the window menu or select the **close** button on the label bar. Or, if you are in another window, simply evaluate the expression:

LearnDispatcher **closeIt.**

Single Pane Window with More Interaction

The window you just created automatically inherits a window's standard text editing capabilities. In many cases, however, you'll want to customize a window to suit your application's needs.

In Chapter 7, you developed an animal habitat with a script to command animals. In this section we will build a custom window for editing and playing animal habitat scripts that uses its own menus. We will build the window in stages adding features as we go.

Opening the Animal Habitat

The new window we are creating is for editing and playing scripts for animals. Since the class **AnimalHabitat** contains the methods for playing scripts we will add the methods for the new window to this class rather than create a new class. By convention, the message usually used to open a window is either **open** or **openOn:**. Since we will want to pass a default or initial script to the window, we will implement a method for the **openOn:** message. The following expression files in the first version of the **openOn:** method for the **AnimalHabitat** class:

```
(File pathName: 'window1.st') fileIn
```

This method opens a window that behaves in the same way as a workspace or System Transcript window.

```
openOn: aString
    "Create a single pane window with aString
     as its initial script."
| topPane |
inputString := aString.
topPane := TopPane new label: 'Habitat'.
topPane addSubpane: TextPane new.
topPane dispatcher open scheduleWindow
```

Every window has a **TopPane** which encompasses the entire window, including the label. In addition to initializing the window label, it acts as a local scheduler inside the window to pick the active pane. To do this, it needs to know the existence of every subpane. Hence, we send the **addSubpane:** message to the **TopPane**.

The last line first sends the **open** message to the dispatcher associated with the **topPane**, which prompts for the window area and displays the window. It then sends the **scheduleWindow** message to let the **Scheduler** activate the window. The global variable **Scheduler** is an instance of class **DispatchManager**; only the windows known to it can be activated.

Now, to open the window, evaluate:

```
AnimalHabitat new openOn: 'Snoopy be quiet'
```

A blank window with the label "Habitat" will appear.

Congratulations, you have created your first customized window! However, this window cannot do much, except some text editing. It does not even display the initial script, the argument **aString**, since we have not defined a mechanism for the habitat to communicate with the window.

Before proceeding, close the window by selecting **close** from the window menu or select the **close button** on the label bar.

Connecting the Habitat to the Window

Our next version of the **openOn:** method adds a means of communication between the habitat and the window. The following expression files it in.

```
(File pathName: 'window2.st') fileIn
```

Here is the new code:

```
openOn: aString
    "Create a single pane window with aString
     as its initial script."
| topPane |
inputString := aString.
topPane := TopPane new label: 'Habitat'.
topPane addSubpane:
    (TextPane new
     model: self;
     name: #input).
topPane dispatcher open scheduleWindow
```

In this method, we've sent two additional messages to the newly created **TextPane**:

model: self tells the text pane the identity of the controlling application (in our case, an instance of **AnimalHabitat**) so that the text pane can send messages to the application.

name: #input lets the text pane identify itself with the name **#input**. The text pane also uses its name as a message to the controlling application to retrieve the pane contents when it is first opened. Hence we implemented the message **input** in the application class, **AnimalHabitat**, to initialize the contents of the text pane:

```

input
    "Initialize inputPane with inputString."
    aInputString

```

We can now open the new window with the following expression:

```
AnimalHabitat new openOn: 'Snoopy be quiet.'
```

Now the initial script appears when the window is opened.

Customizing the Habitat Pane Menu

If you pop up the pane menu of the above window, you still see a standard text editor menu. In order to play the script, you'll want to customize this menu.

The following expression files in the final version of the `openOn:` method:

```
(File pathName: 'window3.st') fileIn
```

We now present the final version of the `openOn:` message for `AnimalHabitat`:

```

openOn: aString
    "Create a single pane window with aString
     as its initial script."
    | topPane |
    inputString := aString.
    topPane := TopPane new label: 'Habitat'.
    topPane addSubpane:
        (inputPane := TextPane new
         model: self;
         name: #input;
         menu: #inputMenu).
    topPane dispatcher open scheduleWindow

```

We have made two changes to the `openOn:` method. We send the additional message `menu:` to the text pane to inform it that the controlling application has a method called `inputMenu` to create the menu for the input pane. We also assign the instance variable `inputPane` to point to the input pane, so that we can communicate with the pane.

Here is the `inputMenu` method which returns the new menu for the input pane.

```

inputMenu
    "Answer a Menu for the input Pane."
^Menu
    labels: 'copy\cut\paste\play selection\play all' withCrs
    lines: #(3)
    selectors: #(copySelection cutSelection
                  pasteSelection playSelection playAll)

```

The method **inputMenu** simply returns a menu. In this menu, the first three selectors **copySelection**, **cutSelection**, and **pasteSelection** are the standard **TextEditor** messages for copying, cutting, and pasting, which we do not need to implement again in class **AnimalHabitat**. We do, however, need to implement the following two new methods:

```

playSelection
    "Accept selected string as the script
     and play it to animals."
CursorManager execute change.
self script: inputPane selectedString.
self play.
CursorManager normal change

```

```

playAll
    "Accept the entire content of the pane
     as the script and play it to animals."
CursorManager execute change.
self script: inputPane contents.
self play.
CursorManager normal change

```

The **playSelection** method plays only the selected text in the input pane, while **playAll** plays the entire text in the pane. Notice that we change the cursor shape to **execute** (shown as an hour glass) while playing the script, and then change it back to **normal** when finished.

The window is now defined. To try it, first set up the global variable **Habitat** and teach the animals commands in the same way as you did in Chapter 7, and then open the window:

```

Snoopy := Dog new name: 'Snoopy'.
Polly := Parrot new name: 'Polly'.
Habitat := AnimalHabitat new
    add: Snoopy;
    add: Polly.

```

Snoopy

learn: 'barking' action: [Snoopy talk];
 learn: 'quietly' action: [Snoopy beQuiet; talk];
 learn: 'is upset' action: [Snoopy beNoisy; talk].

Polly

learn: 'to be pleasant' action:
 [Polly vocabulary: 'Have a nice day'; talk];
 learn: '*' nasty' action:
 [Polly vocabulary: 'Why are you bothering me';
 talk].

Habitat openOn:

'Snoopy is upset about the way that Polly is behaving. It is as if whenever anyone asks Polly to talk, Polly will be nasty. Maybe if instead of Snoopy barking at Polly when he wants Polly to talk, Snoopy quietly asks Polly to be pleasant for a change, things would go better. Now maybe Snoopy barking quietly will not make Polly nasty.'

After the window is displayed, pop up the pane menu and select the **play all** choice. You'll then see the dialogue in the Transcript window. Next, try selecting only a portion of the script; pop up the same menu, pick the **play selection** choice, and watch the dialogue.

Multi-Pane Windows

Multi-pane windows, such as the Class Hierarchy Browser or the Disk Browser, group related functions into several panes within a single window. This gives the user a clear picture of the application, since panes within one window do not overlay each other. Multi-pane windows are also ideal when panes within the window interact with one another to a high degree. For example, when you select a directory in the Disk Browser's directory list pane, the contents of two other panes (the file list pane and the bottom text pane) are automatically updated.

We'll now create a window which groups the animal pattern matching you saw in Chapter 7 with the animation from Chapter 9. In this window, we add two more panes to the previous example:

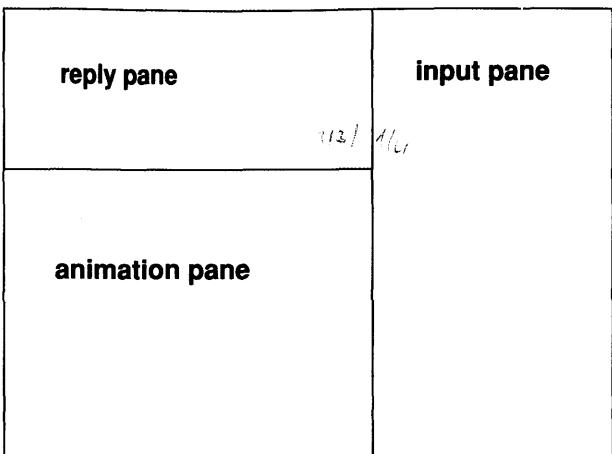


Figure 10.1
Animation
Window

The *input pane* is the same as the one in the previous example, where you can edit the script to be played to the animals. The *reply pane* contains the dialogue from animals, so that we don't have to use the System Transcript window any more. The *animation pane* is the stage where animals can perform in response to commands in the script.

The following expression files in a new `openOn:` method which opens the above multi-paned window and `initWindowSize` which establishes its default size:

```
(File pathName: 'window4.st') fileIn
```

Here is the new code:

```

openOn: aString
    "Create a kennel window with aString
     as its initial script."
    | topPane replyPane |
    inputString := aString.
    (topPane := TopPane new)
        label: 'K E N N E L';
        model: self.
    topPane addSubpane:
        (replyPane := TextPane new
            model: self;
            name: #reply;
            framingRatio: (0@0 extent: 2/3 @ (1/4))).
    topPane addSubpane:
        (GraphPane new
            model: self;
            name: #graph:;
            framingRatio: (0 @ (1/4) extent: 2/3 @ (3/4))).
    topPane addSubpane:
        (inputPane := TextPane new
            menu: #inputMenu;
            model: self;
            name: #input;
            framingRatio: (2/3 @ 0 extent: 1/3 @ 1)).
    replyStream := replyPane dispatcher.
    topPane dispatcher open scheduleWindow

```

We call this window "KENNEL" because it will contain only dogs. (Your Smalltalk/V diskettes include the **FreeDrawing** utility, with which you can draw your own pictures of different animals and try them out.) The window consists of three panes: a text pane called **reply**, a graph pane called **graph**, and another text pane called **input**.

We set the instance variable, **inputPane**, to point to the **input** pane, so that later we can use it to reference the pane's contents. We also set another instance variable, **replyStream**, to point to the dispatcher of the **reply** pane, so that we can later use it like a stream and output things to it.

Since the size of each subpane depends on the size of the whole window, the message **framingRatio:** defines the position and size of each pane relative to its window. The coordinates of the rectangle argument to **framingRatio:** are a fraction of the width or height of the window. For example, if the window rectangle is:

100 @ 100 extent: 300 @ 200

then a framing ratio of **(2/3 @ 0 extent: 1/3 @ 1)** yields the rectangle:

$(100 @ 100) + ((300 @ 200) * (2/3 @ 0)) \text{ extent:}$
 $(300 @ 200) * (1/3 @ 1)$

which is equivalent to:

300 @ 100 extent: 100 @ 200

Since we want the window to open with a particular size, we provide the method **initWindowSize** which is used when the window is first opened:

initWindowSize

“Answer the initial window extent”

$\wedge (\text{Display boundingBox insetBy: } 16 @ 16) \text{ extent}$

If you do not supply an **initWindowSize** method, the system uses a standard default size.

Several other new methods need to be added and existing methods need to be modified before we can try out the new window. The following code files in the changes area additions:

(File pathName: ‘window5.st’) fileIn

The following two methods initialize the two additional panes:

reply

“Initialize reply pane with an empty String.”

$\wedge \text{String new}$

graph: aRect

“Initialize graph pane area to aRect and the animation associated with it.”

$| \text{aForm} |$

aForm := Form

width: aRect width

height: aRect height.

aForm displayAt: aRect origin. “background”

animator := Animation new initialize: aRect.

animals do: [:anAnimal |

animator

add: anAnimal picture

name: anAnimal name

color: anAnimal color].

$\wedge \text{aForm}$

Now, we'll slightly modify the previous example's `playSelection` and `playAll` methods by adding the message `changed:`, which asks the pane identified by its `Symbol` argument to reinitialize the pane contents:

```
playSelection
  "Accept selected string as the script
   and play it to animals."
  self changed: #reply.
  CursorManager execute change.
  self script: inputPane selectedString.
  self play.
  CursorManager normal change
```

```
playAll
  "Accept the entire content of the pane
   as the script and play it to animals."
  self changed: #reply.
  CursorManager execute change.
  self script: inputPane contents.
  self play.
  CursorManager normal change
```

Next, we'll change the `answer:` method to write to the reply pane instead of the System Transcript window:

```
answer: aString
  "Output aString to the reply Pane."
  replyStream
    nextPutAll: aString;
    cr
```

Next, we add a new method to clean up objects that are no longer useful after the window is closed. The message `super release` releases the objects needed by the `changed:` message in methods `playSelection` and `playAll`:

```
release
  "Window is closed, release all the
   objects created by this habitat."
  inputPane := replyStream := nil.
  super release
```

Changes to Animal Classes

At the beginning of this tutorial we filed changes and additions to the `Animal` methods. We present them in this section.

Since we are going to animate the animal objects, we need to provide methods for initializing and accessing an animal's color and picture attributes. We are storing the color as a symbol representing the desired color and the picture as a series of images of the animal in motion that can be used by the **Animation** class described in Chapter 9. Here are these methods:

```

color
    "Answer the receiver's color."
^color

name: aString picture: images color: aColor
    "Initialize the receiver's name, pictures,
     and color."
name := aString.
picture := images.
color := aColor

picture
    "Answer the receiver's array of pictures."
^picture

```

We need to provide behavior that lets the animals move about the graph pane. Since we are only using dogs for our example, we present the new methods for class **Dog**:

```

run: distance
    "Run for distance."
self answer:
    'I am running ',
    distance printString,
    ' feet'.
habitat animator
    speed: topSpeed;
    tell: name go: distance

turn: anAngle
    "Turn direction with anAngle."
self answer:
    'I am turning ', anAngle printString, ' degrees'.
habitat animator
    tell: name turn: anAngle

```

```

walk: distance
    "Walk for distance."
self answer:
    'I am walking',
    distance printString , ' feet'.
habitat animator
    speed: self topSpeed // 2;
    tell: name go: distance

```

In addition there are methods implementing the following new messages: **bounce:**, **topSpeed:**, **direction:**, and **home**. The implementations are very similar to the ones given above. You can use the Class Hierarchy Browser to view them.

Running the Animation

Now that the methods are defined, we can initialize the animals:

```

| dogImages |
dogImages :=
Array
    with: (GraphDictionary at: 'dog1')
    with: (GraphDictionary at: 'dog2')
    with: (GraphDictionary at: 'dog3')
    with: (GraphDictionary at: 'dog2').
Kennel := AnimalHabitat new.
Kennel add:
    (Snoopy := Dog new
        name: 'Snoopy'
        picture: dogImages
        color: #gray).
Kennel add:
    (Lassie := Dog new
        name: 'Lassie'
        picture: dogImages
        color: #black).
Kennel add:
    (Wow := Dog new
        name: 'Wow'
        picture: dogImages
        color: #darkGray).

```

The above expressions add three dogs to the Kennel. Each dog has the same array of pictures but different colors. (Chapter 9 explains animation in detail).

Next we teach each dog the commands that define their behavior. Since all three dogs learn the same set of commands, we show only one dog's learning here:

```
Snoopy learn: 'bark a little'  
    action: [Snoopy beQuiet].  
Snoopy learn: 'bark a lot'  
    action: [Snoopy beNoisy].  
Snoopy learn: 'talk' action: [Snoopy talk].  
Snoopy learn: 'home' action: [Snoopy home].  
Snoopy learn: 'top speed'  
    action: [Snoopy topSpeed: Script peek asInteger].  
Snoopy learn: 'run'  
    action: [Snoopy run: Script peek asInteger].  
Snoopy learn: 'run inside kennel'  
    action: [Snoopy bounce: Script peek asInteger].  
Snoopy learn: 'walk'  
    action: [Snoopy walk: Script peek asInteger].  
Snoopy learn: 'direction'  
    action: [Snoopy direction: Script peek asInteger].  
Snoopy learn: 'turn'  
    action: [Snoopy turn: Script peek asInteger].
```

Finally, evaluate the following expression to open the new window with a starting script:

Kennel openOn:

```
'Snoopy home
Lassie home
Wow home
Snoopy top speed 40
Lassie top speed 30
Wow top speed 20
Snoopy bark a little
Lassie bark a little
Wow bark a lot
Snoopy direction 45
Snoopy walk 200
Snoopy talk
Lassie direction 90
Lassie walk 150
Lassie talk
Lassie turn 225
Lassie run 150
Lassie talk
Wow direction 0
Wow walk 300
Wow talk
Wow turn 135
Wow run 250
Wow talk
Snoopy run inside kennel 4000 feet'
```

Pop up the pane menu of the input pane and select the **play all** choice to play the entire script. You see the dogs' dialogue in the reply pane, and their performance in the graph pane. The final window is shown in the following picture.

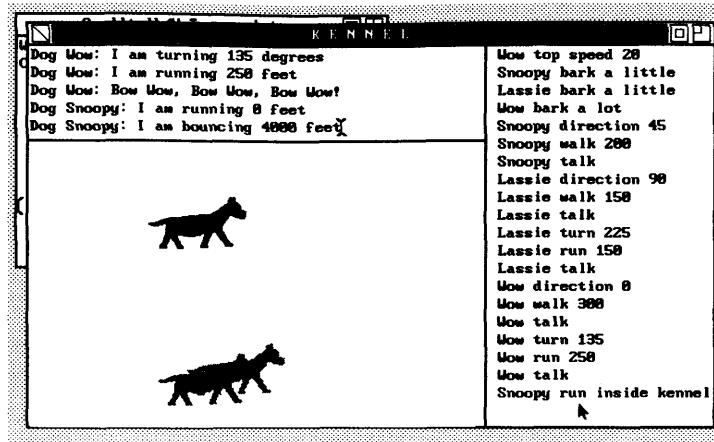


Figure 10.2
Kennel

Try composing your own scripts in the input pane, and then select either all or a portion of it to play.

What You've Now Learned

By the end of this tutorial, you should be familiar with prompters, single pane windows, and multi-pane windows. You've seen a complex interactive application, which provides many of Smalltalk's most important building blocks.

As always, if you want to review any of the topics covered in this tutorial, you can either repeat the corresponding section of the tutorial, or refer to the detailed description in Part 3.

If you are going to exit Smalltalk/V before proceeding on to the next tutorial, be sure to save the image.

11 OBJECT-ORIENTED DEVELOPMENT

The tutorials so far have shown you many strengths of the **Smalltalk/V** language and programming environment. Chapters 11 and 12 are about **Smalltalk/V** application development—the “how” rather than the “what” of OOPS. In Chapter 11, you will learn the OOPS application development cycle, a methodology which will guide you in designing and implementing your own **Smalltalk/V** applications. The process is demonstrated through a simple but complete example.

After working through the phone book example in this chapter, you will be ready for the more complex application in Chapter 12. There, a multi-window, multi-pane application involving state-transition data management is developed using **Smalltalk/V** in a case study of the development of an office information system.

The **Smalltalk/V** Application Development Cycle

Smalltalk/V application development involves defining objects, their interrelationships and their behavior. Together with the **Smalltalk/V** environment, these problem solving techniques support an incremental and evolutionary approach to software development.

The **Smalltalk/V** application development process can be generally divided into six phases:

- State the Problem
- Draw the Window
- Identify the Classes
- Describe Object States
- List the Object Interfaces
- Implement the Methods

State the Problem

Begin your **Smalltalk/V** application development project by making as explicit and cogent a problem statement as possible, describing what you are trying to get your software to do. For example:

Problem Statement: There are many people I call regularly on the phone, so I will create a window application which maintains a list of people and their phone numbers. The phone book list will be sorted in alphabetical order. Selecting a name will retrieve the associated phone number. Names and phone numbers can be added to and deleted from the list.

As you think through and learn more about your problem, you can return to the problem statement to qualify or extend it, refining application goals as you go along. The purpose here is to have a short, understandable statement of the problem which points toward the kind of software solution that will address your need.

Your problem statement, like your Smalltalk/V application, will evolve. The act of bringing your initial design ideas to life will generate insights that enhance your appreciation for the original problem.

Draw the Window

Armed with your problem statement, ask yourself what a new window that helps solve your problem looks like, then draw it. Don't worry about the internal workings of the application and data structures involved. Concentrate on the appearance and interaction. What does the application look like? How does it behave? What existing Smalltalk/V windows have you seen that suggest an approach?

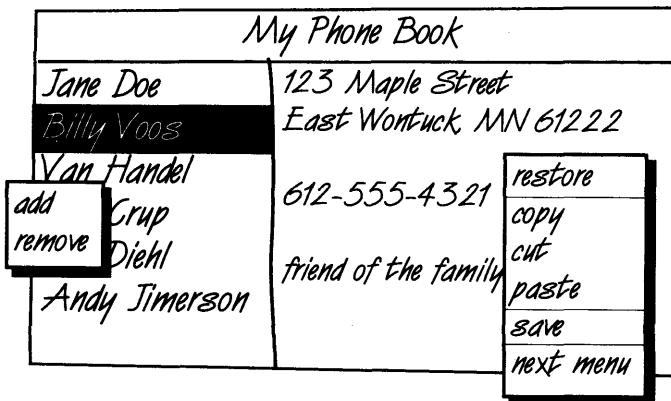


Figure 11.1
PhoneBook
Window Model

Figure 11.1 shows a proposed phone book window. Notice that it includes menu selections. Having an idea of the number, size and contents of the panes of the window in your application is obviously helpful, but it is through the menu selections that things come to life. Menu items are the means for you as user-object to send messages to your application which is, after all, a Smalltalk/V object.

You have now transformed your problem statement into a concrete window to be built. The drawing is a specification of the application user interface and is the starting point for specifying the rest of the application.

The next three phases consist of identifying classes, object state and object interfaces, and are by nature interrelated. Although they are described sequentially, it may be more productive to perform them in parallel or iteratively.

Identify the Classes

Knowing as much as you can about your problem and having a sketch of the application interface, you are ready to identify the classes of objects that implement the application. List both the new classes defined for the application and existing classes from the Smalltalk/V class hierarchy to be used. New classes will be either self-standing (a subclass of class **Object**), or a subclass of an existing class that implements some of the behavior needed by the new class. Existing classes used will generally include user interface classes (**Menu** and subclasses of **Pane**) and subclasses of **Collection**, **Magnitude** and **Stream**.

In our example, we need a new class **PhoneBook** whose instances are the phone book window application. Menus and panes appear in your window sketch, so classes **Menu**, **ListPane**, **TextPane** and **TopPane** are needed. After some consideration, it seems that a **Dictionary** is most appropriate for associating names with telephone numbers, and names and telephone numbers will be represented as **Strings**.

The classes identified so far seem like a natural breakdown of the problem statement and the window picture. But what if the choices aren't so obvious? It's important to make some choice, even if it's not the best. You may be wrong, but with an environment that allows you to make changes easily, you can change to better ones later. Any kind of progress on the application will enhance your understanding of the appropriate classes to be used.

Describe the Object States

Given the problem statement, the window drawing and the classes to be used, what instance variables are needed in the new classes you have defined? In this example, our only new class is **PhoneBook**, which is the window model. What state is appropriate to put in a phone book object? The standard dispatcher and pane classes already keep track of most of the state associated with the window. The model, an instance of class **PhoneBook**, needs the application specific data, which in this case are (1) the dictionary of phone numbers **phones** and (2) the name from the selected list pane entry **selectedName**. Look in file **chapter.11** to see the resulting class definition.

Are there general rules for deciding what instance variables to use? The best guide is to look ahead to the object interface. What kind of questions (messages) will the object be asked? What will be the object's behavior?

List the Object Interfaces

Object interfaces are the messages that the object responds to. List the messages that the new classes must implement. A good starting point for this is to augment the window drawing with the messages that will be used for each pane.

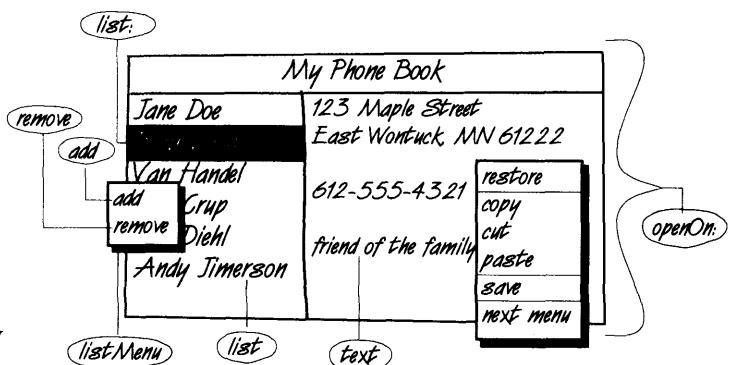


Figure 11.2
PhoneBook Window
with Messages

In Figure 11.2, the message names are circled. In general, messages are used for (1) accessing non-standard menus defined by the application, (2) accessing the contents of panes for display purposes, (3) saving the contents of changed panes, (4) carrying out menu selections, and (5) opening the window. In our example we have the following:

<i>Message</i>	<i>Purpose</i>
add	Add a new name to phone book.
list	Answer the list of names of people in the phone book.
list: aString	Display the phone number for aString.
listMenu	Answer the name list menu.
openOn: aDictionary	Open a phone book window for aDictionary.
remove	Remove selected name from phone book.
text	Answer the phone number for the selected name.
text: aString from: aDispatcher	Enter aString in dictionary as phone number for selected name.

Implement the Methods

All that remains to do is to implement the methods. A good starting point for a window application is to do the window opening message (in our case, `openOn:`). This method is generally the largest in your application. Here it is for the phone book:

```

openOn: aDictionary
    "Open a phone book window on aDictionary. Define
     the pane sizes and behavior and schedule the window."
| topPane |
phones := aDictionary.
topPane := TopPane new.
topPane label: 'Phone Book'.
topPane addSubpane:
    (ListPane new
        menu: #listMenu;
        model: self;
        name: #list;
        change: #list:;
        framingRatio: (0 @ 0 extent: 1/3 @ 1)).
topPane add Subpane:
    (TextPane new
        model: self;
        name: #text;
        change: #text:from:;;
        framingRatio: (1/3 @ 0 extent: 2/3 @ 1)).
topPane dispatcher open scheduleWindow

```

All open messages for a window are very similar. They generally initialize model instance variables, and create window panes by specifying the following for each pane:

- **menu:** The pane menu. Not used if the standard pane menu is used.
- **model:** The pane model, generally `self`.
- **name:** Specifies the selector of the message used to ask model for pane contents.
- **change:** Specifies the selector of the message used by the pane to tell the model that the pane contents are changed.
- **framingRatio:**
A rectangle that specifies the relative position and size of the pane in a window `0@0 corner: 1@1`.

The remaining methods for the phone book are small and are contained in the `chapter.11` file. The following expression files in class `PhoneBook`.

(File pathName: 'chapter.11') fileIn

You will find it as the first expression in file **chapter.11**.

To try it out, evaluate:

```
PhoneBook new openOn: Dictionary new
```

Knowing When to Stop

During the first round of a new application development project, you generally progress through the above six steps sequentially. Once you have your "first cut" at a proposed application, your increased understanding will generate new insights. You will find yourself jumping around the six steps in a non-sequential process. For instance, while writing a method you realize the need for a class instance variable which brings you back to state considerations.

Smalltalk/V's incremental compiling capabilities promote dynamic movement among these development steps. In fact, **Smalltalk/V** application development can be seductive. It is always easy to think about adding just a few more lines of code which will further polish the application's interface or performance. By design, **Smalltalk/V** is an open system, an application always has the potential for enhancement.

That is why it is important to refresh your memory as to the original design goals of your **Smalltalk/V** application development project. If the prototype meets your design criteria, then stop. Use the application for a while and let experience guide subsequent enhancement.

What You've Now Learned

You have seen the **Smalltalk/V** application development cycle applied to a very simple example. You have seen that a **Smalltalk/V** application develops in a revolving process which addresses:

- State the Problem
- Draw the Window
- Identify the Classes
- Describe Object States
- List the Object Interfaces
- Implement the Methods

The next chapter will showcase the **Smalltalk/V** application development cycle in a more involved case study. As always, if you want to review any of the topics covered in this tutorial, you can either repeat the corresponding section of the tutorial, or refer to the relevant topics in Part 3.

12 APPLICATION DEVELOPMENT: CASE STUDY

This chapter gives you a snapshot of the **Smalltalk/V** big picture...

- How do you approach real world problems with Smalltalk solutions?
- What tips and techniques help your application prototyping?
- Where are the "hot spots" in the **Smalltalk/V** manual that help you write your own applications?
- Where do you start?...How do you proceed?

We'll take a case study approach. A case of real world people, not whiz Smalltalk programmers. This chapter is intended for non-programmers, people with something they want their computers to help them do.

The Case Study: A State-Transition Perspective

Our case involves the prototype development of an office automation system, a sales communication application. In the general scheme of things, the case study is an analysis involving **state-transition models** where physical objects pass through a series of states according to a network of connections between the states.

The connections between states reflect the logical constraints and preconditions which order the transition from one state to another. The tokens which pass from node to node along the constraint network are objects experiencing the state changes.

Objects going through state changes is a subject of broad interest. The objects might be:

- A molecule in a chemical reaction
- A widget on the widget assembly line
- A source code instruction processed by a compiler
- A ticket in an airline reservation system

State-transition models relevant to these objects include, respectively:

- A theory on the molecular interaction of the substances involved in the reaction
- A specification of the factory's numerical control tools and assembly line conveyors, including capacities and set-up time
- A graph grammar specifying the step-wise decomposition of a high-level language instruction into a collection of machine instructions
- A diagram describing the paperwork flow involved in getting a ticket to a traveler

Smalltalk/V is ideal for developing applications based on state-transition and other highly graphical and data intensive models. Its screen graphics and rich data abstracting capabilities allow you to produce working prototypes of complex models in record time.

Our case study shows how some self-starters in an optical disk drive sales office develop a **Smalltalk/V** application to track and facilitate their sales communications. Their state-transition network—though they didn't know or care to call it that—is a flowchart of their company's approved sales strategy. The tokens coursing through this sales strategy network are their potential customers.

The Case Study Problem as a **Smalltalk/V** Problem

Where to begin? The Whizzard WORMS reps knew they had a sales communication problem. They knew they wanted to develop a **Smalltalk/V** application to facilitate these communications consistent with corporate marketing strategy. They had a sense of what the application interface would be like—inspired by things they'd seen, code they already had on hand. A lunchtime, napkin-based brainstorming session captured the problem.

On the first napkin they drafted a preliminary objective, their problem statement:

To develop a Sales Communication application (**SalesCom**) that will allow a rep to sit down to its interactive display where he or she will quickly and easily see what communication events are required for the day and to see what the current load of their customers is for each of the steps in the sales strategy. Further, the prototype version will print letters on a case by case basis and will facilitate making telephone follow-up calls. The prototype application will be built as much as possible from reusable code taken from the **Smalltalk/V** tutorial examples and system source files.

Looking over the rep developers' shoulders with the eyes of a generalist, their problem statement can be restated:

To develop a **Smalltalk/V** application using state-transition models to manage the movement of objects through a process. This application will have a multi-pane main window where a database of tokens moving through the process can be singled out for state inspection and manipulation. The application keeps a token state change event schedule. When moving a token through the process, the application prompts the user for preconditions or the result of state transitions where the network exhibits multiple alternate branches.

The reps, as far as WORM disk sales are concerned, see a world of "us" and "them". Inside are the Whizzard reps and outside, a sea of current and potential customers. The rep's job consists of acting as an agent for Whizzard through a series of communication events with a customer which hopefully lead to a sale.

They sketched a diagram of their customer prospecting strategy as shown in Figure 12.1. It was a simple marketing strategy which involved a direct mail campaign, followed by a telephone contact with an attempt to get interested folks to attend a sales seminar.

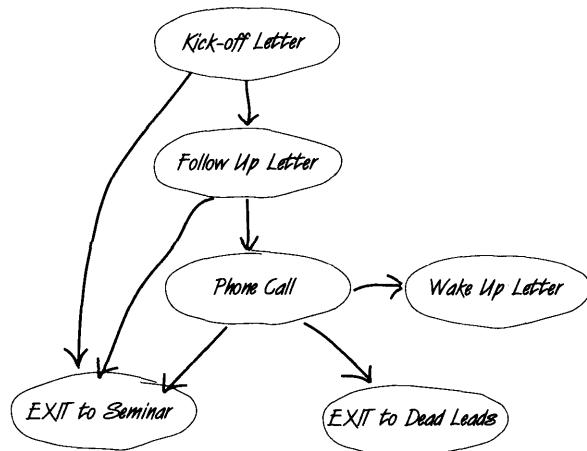


Figure 12.1
SalesCom Sales Strategy

A Window Model for the SalesCom Application

The reps next sketched a rough picture of the ideal SalesCom application window as in Figure 12.2. Their objective was to take inspiration from the various tutorial examples and system features to formalize their vision of the proposed application.

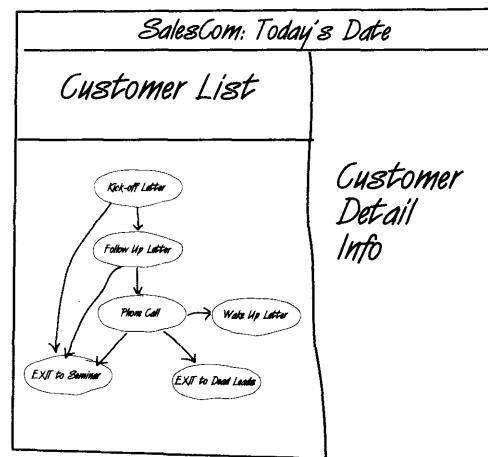


Figure 12.2
Sketch of **SalesCom** Window

There were two purposes for the display as far as they could see:

- They want a basic facility to scan and manage the database of customers. Once one is selected, they want to see the detailed record of that customer.
- They want to promote a visual connection between their real world activity and the formal corporate marketing strategy represented by the (state-transition) diagram in Figure 12.1.

The first requirement suggested the **ListPane** and **TextPane** views in the **SalesCom** window model, the Master Customer List pane and Customer Detail text pane. The inspiration was the **Smalltalk/V** browsers where a clicked selection in a scrolling **ListPane** initiates a look-up of some associated detail information, the source code for a method or the contents of a file for instance.

The second requirement begged for graphic representation based on an extension of the tutorial Network of Nodes example. The "hard" part of taking a network data structure and giving it a visual representation is done already. The reps needed only to give the nodes more representative names and new screen coordinates to transform the random network of Figure 9.2 to that of Figure 12.1.

The **GraphPane** of the **SalesCom** application (inspired by the animation pane of the Animal Habitat tutorial example) facilitates the salesperson's visualization of the communication process and the flow of customers through the sales cycle. Numbers appended to the label of each communication event node indicate the total number of customers currently pooled at that step in the strategy as well as the year-to-date total of customers having passed through the event node.

Menus Enrich the Window Model

The reps next addressed what the application would have to do to move customers through the marketing strategy. What "messages" would the user rep want to send to the **SalesCom** application object to control preparing letters and prompting phone calls?

They gravitated toward brainstorming about the menus of each of the **SalesCom** application subpanes. They found a pad of removable self-stick notes was useful for laying hypothetical menu pop-ups over each pane. After much debate, the reps decided on the configuration in Figure 12.3.

The reps were excited. They finished lunch with a clear sense of direction on their development project. They were ready to start writing code.

Survival Tip # 1: Don't over-plan. Yes, you need a decent statement of your problem and some sense of what the application's primary interface will look like. But you don't have to have all the answers to get going. To determine when to start, ask yourself, "If I write as much code as

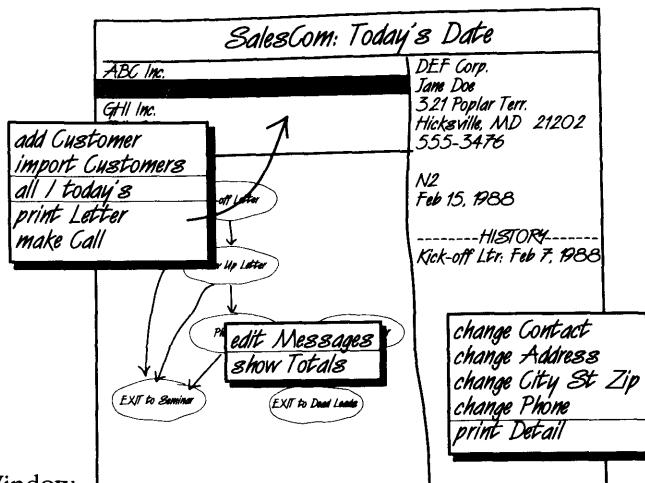


Figure 12.3

Menus Enrich SalesCom Window

it takes to get a prototype to look and behave like the design so far, would it do anything interesting and useful?" When your answer is affirmative, go to it. Smalltalk/V is designed to work with you in an evolving, incremental process.

As they prepared to begin development, the reps took a couple of minutes to draft a list of the classes involved in the SalesCom application:

- SalesCom, the sales communication application and window model class
- ComEvent, the class of communication events
- Network, the network of ComEvents which make up a sales strategy
- Dictionary, to associate customer names and customer data

Getting There in Half the Time: Recycling Code

The reps set to work on a kind of scavenger hunt for reusable code to get the SalesCom application up and running as soon as possible. You can recreate their exploratory process by opening a Class Hierarchy Browser and use your Disk Browser to open the chapter.12 tutorial file.

As you do more Smalltalk/V programming you will see that you re-use code in two basic ways:

- indirectly through inheritance when creating new subclasses, and
- through cut and paste of methods from an existing class to a new, non-hierarchically related class.

You will use both methods in creating the SalesCom application, starting with the inheritance approach by creating a new class of object, ComEvent, as a subclass of the NetworkNode class. You will then go on a cut and paste "raid" on the AnimalHabitat class to borrow the basic window model for the three pane SalesCom application.

If you have not done the tutorial examples in Chapters 6, 7, 9 and 10 and/or have not saved the image along the way, you should add the following classes by evaluating:

```
Object subclass: #NetworkNode
instanceVariableNames:
    'name position'
classVariableNames: ""
poolDictionaries: "
```

Then select and evaluate:

```
Object subclass: #Network
instanceVariableNames:
    'connections'
classVariableNames: ""
poolDictionaries: "
```

and evaluate:

```
Object subclass: #AnimalHabitat
instanceVariableNames:
    'animals replyStream animator inputString inputPane'
classVariableNames: ""
poolDictionaries: "
```

Then select and evaluate the following to file in the relevant methods for these three classes. This is to give you a starting point equivalent to where the sales reps started.

```
(File pathname: 'network9.st') fileIn.
(File pathname: 'nodes9.st') fileIn.
(File pathname: 'class10.st') fileIn.
(File pathname: 'animal10.st') fileIn.
(File pathname: 'window4.st') fileIn.
(File pathname: 'window5.st') fileIn.
```

Update the class list pane of the Class Hierarchy Browser.

Re-working the Network of Nodes

The starting point of the Network of Nodes example can be seen in Figure 9.2.

As a first step, we create a new pool dictionary, **SalesStrategy**, to hold objects that will be global to our new application. Initially we create two new pool variables: **StratNet** to hold the sales strategy network, and **StratNodes** to hold the nodes used in the sales strategy network. Evaluate the following:

```
SalesStrategy := Dictionary new.
SalesStrategy
  at: 'StratNet' put: Network new initialize;
  at: 'StratNodes' put: Dictionary new
```

The network node of the **SalesCom** marketing strategy diagram requires a richer data structure than the simple name and position instance variables defined for the path computation and graphic display examples from earlier tutorial chapters. So we will create a **ComEvent** subclass of **NetworkNode**. In addition to a couple of new methods, a **ComEvent** has an **info** instance variable which is initialized as a **Dictionary** object. **Info** will be used to store information about the body of the letter or phone call script, the delay before the next event and a pointer to the next most likely event, etc. Select and evaluate:

```
(File pathName: 'comevent.cls') fileIn
```

We will use the pool variables **StratNet** and **StratNodes** to build a sales strategy network. **StratNet** and its nodes are given new names, positions and connections which transform the prior random network graphic example into the structured diagram of the **SalesCom** marketing strategy. Evaluate the following to see the transformation:

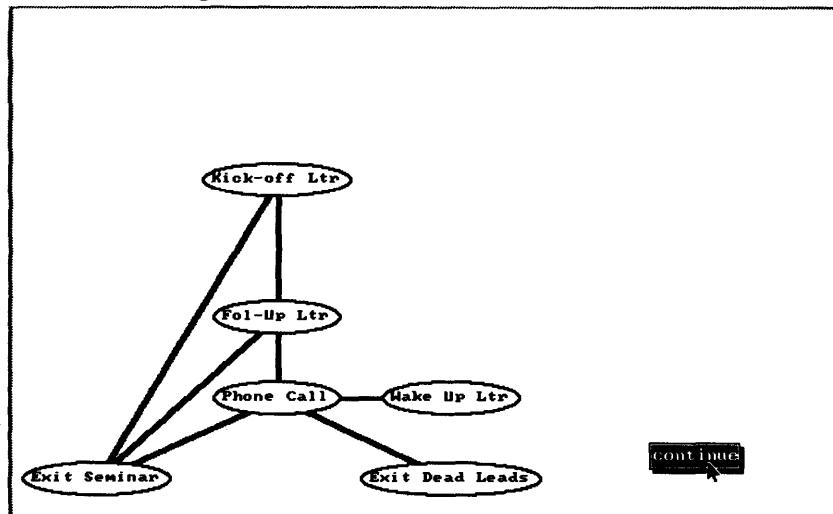
```
#((('Kick-off Ltr'      180  20)
  ('Fol-Up Ltr'        180 120)
  ('Phone Call'        180 180)
  ('Wake Up Ltr'       295 180)
  ('Exit Seminar'      57  240)
  ('Exit Dead Leads'  280 240))
 do: [:nodeInfo |
  ComEvent new initialize;
  name: (nodeInfo at: 1) position:
    (nodeInfo at: 2) @ (nodeInfo at: 3)].
(SalesStrategy at: 'StratNet') initialize.
ComEvent
  connect: 'Kick-off Ltr'      to: 'Fol-Up Ltr';
  connect: 'Kick-off Ltr'      to: 'Exit Seminar';
  connect: 'Fol-Up Ltr'        to: 'Phone Call';
  connect: 'Fol-Up Ltr'        to: 'Exit Seminar';
  connect: 'Phone Call'        to: 'Wake Up Ltr';
  connect: 'Phone Call'        to: 'Exit Seminar';
  connect: 'Phone Call'        to: 'Exit Dead Leads'.
```

To see what the sales strategy net looks like, evaluate the following:

Display white
 (SalesStrategy at: 'StratNet') draw.
 Menu message: 'continue'.
 Scheduler systemDispatcher redraw

Your screen should look like Figure 12.4.

Figure 12.4
SalesCom
 Network
 Screen



The preceding code drew directly on the display. What we want is to draw it to a form so we can display it in a window. We need to edit the methods `draw` in class `Network` and `NetworkNode` to produce an additional method `drawOn:` for each class. You can either directly edit these methods using the Class Hierarchy Browser as indicated below (note the **** changed **** lines) or copy the methods from the file `chapter.12` and paste them over the new method template.

The method `drawOn:` for class `Network` is:

```

drawOn: aForm           "*** changed ***"
    "Draw the network. For each node, it draws all
     the arcs and then the node. All the nodes visited
     are remembered to avoid double drawing."
| visited pen |
      "*** changed ***"
    defaultNib: 4 @ 3.
visited := Set new.
connections keys do: [ :nodeA |
    visited add: nodeA
    (connections at: nodeA) do: [ :nodeB |
        (visited includes: nodeB)
        ifFalse: [
            pen place: nodeA position;
            goto: nodeB position]]].
nodeA drawOn: aForm]      "*** changed ***"

```

The method **drawOn:** for class **NetworkNode** is:

```

drawOn: aForm           "*** changed ***"
    "Draw the receiver node with a circle
     around its name."
| font r aspect pen |
font := Font eightLine.
      "*** changed ***"
    defaultNib: 2;
    mask: Form white.
pen place: position.
pen solidEllipse: (r := name size * font width + 15 // 2)
    aspect: (aspect := font height + 16 / r / 2 / Aspect).
pen mask: Form black.
pen centerText: name font: font.
pen ellipse: r aspect: aspect.

```

Note that the **draw** methods could now be changed to use the new **drawOn:** methods, but we leave this as an exercise for you. (Hint: it can be done in a single line.)

To test the new methods, evaluate the following:

```
| f |
f := Form width: 600 height: 300.
(SalesStrategy at: 'StratNet') drawOn: f.
f displayAt: 0 @ 0.
Menu message: 'continue'.
Scheduler systemDispatcher redraw
```

That's it. You have roughed out one of the two main components of the visual presentation of the SalesCom application.

Take a moment to inspect the ComEvent class. Although the Net drawing of ComEvent nodes is visually similar to the random network tutorial example, notice the impact of the `initialize` and `name:position:` methods. While a ComEvent can be drawn by the same code that draws the original `NetworkNode` of the network tutorial example, a ComEvent has a much richer data structure:

```
initialize
    "Initialize the dictionary data record of information about the
     communication event to be empty."
info := Dictionary new
info
    at: '1.Strategy' put: 'NYS'; "not yet specified"
    at: '2.Type' put: 'L, P or F';
    at: '3.Message' put: 'Type your message here.';
    at: '4.Next Step' put: 'NYS';
    at: '5.Days Till NS' put: 'NYS';
    at: '6.MTD' put: 0;
    at: '7.YTD' put: 0
name: aString position: aPoint
    "Set the name and position while adjusting for screen
     aspect ratio"
super
    name: aString
    position: aPoint x @ (aPoint y * Aspect) truncated.
StratNodes at: aString put: self
```

The next step is to integrate this graphic network display into the multi-pane window you are about to create with code copied from the Animal Habitat tutorial example. Divide and conquer. That's a big part of Smalltalk/V programming strategy.

Raiding the Animal Habitat

Before you can "borrow" some code, you need some place to put it. While you can expect to call on many existing classes in the course of your Smalltalk/V development projects, you can pretty much count on adding a new class for the application model. This is the class which defines the user interface of your application.

To run your "program", you create a new instance of your application model class, then interact with it. In this case, evaluate the following to create the new SalesCom class as a subclass of Object:

```
Object subclass: #SalesCom
instanceVariableNames:
'replyStream'
classVariableNames: ''
poolDictionaries: 'SalesStrategy'
```

The first and most important method in the SalesCom application is the `openOn:` method which describes the window appearance. This method also associates a collection of methods with each of the panes. These methods are triggered as needed when the window or any of its subpanes are created or changed.

You will get much of the code detail and the overall method structure from the `openOn:` method of the `AnimalHabitat` class. Use a Class Hierarchy Browser to locate the method, select all of it and copy it.

Return to the currently empty SalesCom class and add your first method by choosing `new method` from the method list pane menu. Select the entire default method template and chose `paste` from the text pane menu. Then change the method selector from `openOn: aString` to `open`. The method source text pane should look like this at the conclusion of this operation:

```

open
  "Create a kennel window with aString as its initial script."
  | topPane replyPane |
  inputString := aString.
  topPane := TopPane new label: 'K E N N E L';
    model: self.
  topPane addSubpane:
    (replyPane := TextPane new
      model: self;
      name: #reply;
      framingRatio: (0 @ 0 extent: 2/3 @ (1/4))).
  topPane addSubpane:
    (GraphPane new
      model: self;
      name: #graph;
      framingRatio: (0 @ (1/4) extent: 2/3 @ (3/4))).
  topPane addSubpane:
    (inputPane := TextPane new
      menu: #inputMenu;
      model: self;
      name: #input;
      framingRatio: (2/3 @ 0 extent: 1/3 @ 1)).
  replyStream := replyPane dispatcher.
  topPane dispatcher open scheduleWindow

```

Choose save from the method source text pane and you will get an "undefined" message explaining that the **SalesCom** class does not have the instance variable, **inputString**, defined.

What a dilemma. You can't save the method without discarding the changes which, in this case, means losing the entire new method.

Survival Tip # 2: When you are on cut and paste "raids" of existing methods, put the source method's class instance variable name(s) into the temporary variable declaration of the new method in your new class. You can then save the method and decide later whether your new class definition needs to be changed to incorporate the source's instance variable(s) or whether you leave the temporary declarations.

Insert **inputPane**, **inputString**, and **aString** into the temporary variable declaration as follows:

```
| topPane replyPane inputPane inputString aString |
```

And save again. This time it works. But you want a **SalesCom** window to open on a marketing network, not on an **inputString**. Edit the comment. You've determined that you won't be needing a temporary or instance variable called **inputString**, so delete it from the temporary declaration. Also delete the temporary **aString** and remove the first statement setting **inputString**. Edit the **label:** argument now appearing in the first statement of the method. The **SalesCom** window model now looks like:

```

open
    "Create a SalesCom window with aNet
    as its marketing strategy."
    | topPane replyPane inputPane |
    topPane := TopPane new label:
        ('SalesCom: ', Date today printString);
        model self.
    topPane addSubpane:
        (replyPane := TextPane new
            model: self;
            name: #reply;
            framingRatio: (0 @ 0 extent: 2/3 @ (1/4))).
    topPane addSubpane:
        (GraphPane new
            model: self;
            name: #graph;
            framingRatio: (0 @ (1/4) extent: 2/3 @ (3/4))).
    topPane addSubpane:
        (inputPane := TextPane new
            menu: inputMenu;
            model: self;
            name: #input;
            framingRatio: (2/3 @ 0 extent: 1/3 @ 1)).
    replyStream := replyPane dispatcher.
    topPane dispatcher open scheduleWindow

```

Follow the same procedure, copying method **initWindowSize** from **AnimalHabitat** to class **SalesCom** and save it.

SalesCom is shaping up. Evaluate:

```

Test := SalesCom new.
Test open

```

A Walkback window informs you that the **SalesCom** object, **Test**, does not know how to **reply**. Close the Walkback and go back to **AnimalHabitat** to copy the **reply** method and paste it in as a new method in the **SalesCom** class. Save it. Next evaluate:

```

Test open

```

No big deal, another Walkback. A little more careful inspection of `open` reveals the need to copy `graph:`, `input` and `inputMenu` methods from `AnimalHabitat` into `SalesCom`. Start with `graph:`. After pasting it into `SalesCom`, `graph:` will look like this:

```
graph: aRect
    "Initialize graph pane area to aRect and the
     animation associated with it."
    | aForm |
    aForm := Form
        width: aRect width
        height: aRect height.
    aForm displayAt: aRect origin. "background"
    animator := Animation new initialize: aRect.
    animals do: [:anAnimal |
        animator
            add: anAnimal picture
            name: anAnimal name
            color: anAnimal color].
    ^aForm
```

Before saving, notice that most of the code relates to animals and not to our problem. Let's first edit the comment to say what we really want this method to accomplish. Then edit the code to do what is necessary. We can always leave the code incomplete and the comment will serve as a guide to finish it later. In this case, the actual implementation is very simple:

```
graph: aRect
    "Initialize graph pane area to aRect and the
     network strategy graphic associated with it."
    | aForm |
    aForm := Form width: 600 height: 400.
    StratNet drawOn: aForm.
    aForm displayAt: aRect origin clipRect: aRect.
    ^aForm
```

Survival Tip # 3: When re-working a copied method, edit the "comment" first to make it reflect the changes you will be making. Let the edits you make in the comment direct you to lines in the code which implement that aspect of the method responsible for the behavior referred to in the comment. This technique can be so useful that you will develop an appreciation for and will, hopefully, write clear and complete comments. They aren't "training wheels". Comments are an integral part of the communication among the Smalltalk/V community of programmers.

Continue by copying the `inputMenu` method into `SalesCom`. It can be saved without modification, though a `SalesCom` object will not understand if you ask it to perform any of its menu selections related to the Animal Habitat. The reps brought in `inputMenu` to get the `SalesCom` prototype window running as soon as possible. They later used this method as a template to create a menu for each of the `SalesCom` window subpanes as required by their prototype design.

Next copy the `input` method from `AnimalHabitat` to `SalesCom`. Before saving it will look like this:

```
input
  "Initialize inputPane with inputString."
^inputString
```

You can see that this method simply returns the content of `inputString`, an `AnimalHabitat` instance variable, and since you just slipped those instance variable names into the temporary variable declaration of the `open` method, you know this won't work. So to "hardwire" a consistent response from this method until you decide what this method will need to do, change the method to read:

```
input
  "Initialize inputPane with a temporary input string."
^'This is a test input string.'
```

and save the method.

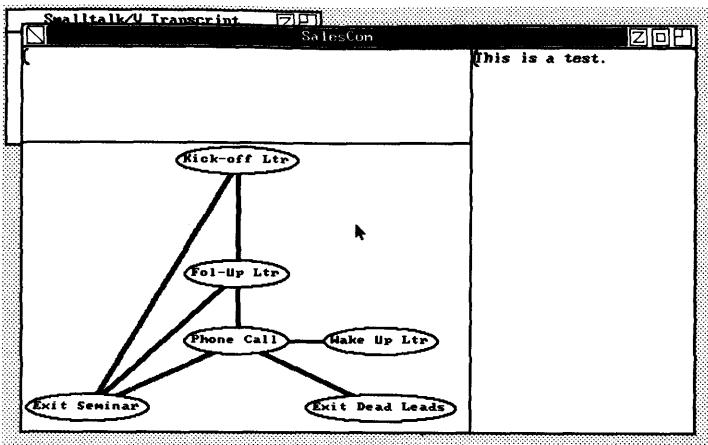
Survival Tip # 4: Simplified, interim versions of methods are an excellent way to "divide and conquer" a Smalltalk/V programming problem. Most often this involves writing a return expression which is a specific instance of the kind of object which will result from the more complex computation that a method is planned to perform. This is especially useful for separating development progress on the interface from progress on the internal manipulations of the involved data structures.

Now select and evaluate:

Test open

You've done it. The first incarnation of a recognizable `SalesCom` object, as seen in Figure 12.5. The pizzaz of the strategy network `GraphPane` hints at the developing interface. The window resembles but certainly isn't an animal habitat anymore. The two `TextPanes` are the targets of the next phase of your `SalesCom` prototype development project.

Figure 12.5
SalesCom Window
Phase One



Customers and Events: A Matter of State

The reps realized they were on track now. They saw two areas of **SalesCom** which needed attention:

- They needed to create an initial state, a test case "world" onto which a **SalesCom** object could open.
- They also needed to add new methods which would make the **SalesCom** window manage the interaction and updating of the Customers and Communication Events databases.

You incorporate many traditional database management operations into the **SalesCom** application simply by making the global variable **Customers** a new **Dictionary** object. The dictionary *keys* will be the customer company names. The value associated with each company is an array, its elements the 'fields' in the customer 'database'. A comment identifies each element in the first case entry. Select and evaluate:

Customers := Dictionary new.

Customers

```

at: 'ABC Inc.' put: #( "Key is company name"
  'Fred Smith'      "Field 1 is Contact Name."
  '123 Maple St.'   "Field 2 is Address."
  'Boston MA 02055'  "Field 3 is City State and Zip"
  '555-4321'         "Field 4 is Phone."
  'Kick-off Ltr'     "Field 5 is Current Event scheduled."
  'Feb 14, 1988'    "Field 6 is Prep Date for Current Event."
  ());
  ("Field 7 is an array for the customer history.");
at: 'DEF Co.' put: #('Jane Doe' '321 Poplar Terr.'
  'Hicksville MD 21202' '555-3476' 'Fol-Up Ltr' 'Feb 15, 1988'
  ('Kick-off Ltr: Feb 7, 1988'));
at: 'GHI Ltd.' put: #('Clive Davies' '999 Oak Ave.'
  'Tustin CA 92680' '555-7890' 'Kick-off Ltr' 'Feb 14, 1988'
  ());
at: 'JKL Inc.' put: #('Bert Jenks' '666 Sycamore St.'
  'Irvine CA 92680' '555-4734' 'Kick-off Ltr' 'Feb 14, 1988'
  ());
at: 'MNO Co.' put: #('Bill Rasp' '345 Apple Ave.'
  'New Vists CA 93232' '555-5678' 'Fol-Up Ltr' 'Feb 20, 1988'
  ('Kick-off Ltr: Feb 6, 1988'));
at: 'PQR Corp.' put: #('Ellie Small' '423 Sassafras Ave.'
  'Tustin CA 92680' '555-2064' 'Phone Call' 'Feb 20, 1988'
  ('Kick-off Ltr: Feb 1, 1988' 'Fol-up Ltr: Feb 8, 1988'));

```

To see the mini-database which you just created, evaluate:

Customers inspect

To inspect a customer—to review and modify the fields in a customer's "record"—select a customer name in the left hand list pane of the Customers Dictionary Inspector. Bring up the pane menu and click on **inspect** to pop up an Inspector on the selected customer. Each element, or field, of the customer record is a selectable line in the new Inspector list pane. Remember to select **save** in the **TextPane** whenever you make field changes that you want to keep.

To explore the many things you can do with **Customers**, turn to the **Dictionary** class entry in the **Smalltalk/V Encyclopedia of Classes**. Try "talking" to the **Customers** dictionary to explore its database-like behaviors.

The reps used the **Encyclopedia of Classes** like a foreign language phrase book throughout the development of **SalesCom**. The reps got the customers to jump through hoops with **keysDo:**, **includes:**, **occurrencesOf:** and **at:put:** messages. Check it out. Flip to the **Encyclopedia** and explore the things you can do to the **Customers**.

Survival tip # 5: Study and understand the format of a class entry in the Encyclopedia of Classes in Part 4 of the Smalltalk/V manual. And learn to actively use the Methods Index in the Appendices to direct your cross-referencing exploration of the behavior of the many classes which make up the Smalltalk/V environment.

With customer objects in hand, the reps turned their attention to the communication events.

The reps used the same basic object inspection technique to fill in the required information in each of the ComEvent network strategy nodes. This included typing in the body of the letters and phone call scripts that will eventually be merged with customer information. To save you time, evaluate:

(File pathName: 'comevents.in') fileIn

To view the new data contained in the ComEvent nodes, select and evaluate:

(SalesStrategy at: 'StratNodes') inspect

Open an Inspector on one of the nodes. Use the Zoom button to get a good view of the letter bodies and phone script. All these data management and multi-window editing features of Smalltalk/V dictionaries are already a part of the capabilities of the SalesCom application. The reps realized this was as sophisticated a data management facility as they needed for the prototype.

SalesCom consists now of two databases, Customers and StratNodes, a network of ComEvents. It will be up to new and original methods to manage these objects' interaction and updating. You will do that by enhancing the behavior of the multi-pane SalesCom window.

Methods and Messages: Bringing the Prototype to Life

The reps next objective was to get each TextPane working with the Customers dictionary. The inspiration was the browser, so they poked around relevant methods, especially open, in the browser classes and determined that they needed one to function as a ListPane. Selections in the ListPane would then trigger a method for the other TextPane to display the data record for the selected customer.

The reps edited open to reflect the following changes:

- replyPane was renamed custsPane and made a ListPane.
- custsPane was assigned a name: method, customers, to return the Sorted Collection of customer company name keys from the Customers dictionary.
- A change: method, viewCust:, was added to custsPane so list selections would trigger display of the customer record in the customer detail TextPane.

- `inputPane` was renamed `custDetPane` to show that it would display customer details.
- `custDetPane` was added to the `SalesCom` instance variable list and recompiled after "killing" the `SalesCom` object, `Test`.
- Each pane was assigned its own menu method.

To see the many changes that the reps made to `SalesCom` during this next phase of prototype development, evaluate:

```
Test := nil.  
(File pathName: 'class12.2in') fileIn.  
Test := SalesCom new.  
Test open
```

You will probably get a walkback saying `has instances`.

Survival Tip # 6: Finding lost instances. Windows cause dependencies to be stored between panes and models in the Dependents dictionary. If you have instances of a class that you cannot find, they are probably in the Dependents dictionary. This usually happens when you are debugging a new window. You remove these instances by executing the expression:

Scheduler reinitialize

All open windows close and the System Transcript reappears.

`SalesCom`'s basic behavior is evolving. Note the reps are using a "divide and conquer" approach to bring the prototype up in stages with the interface taking the lead. The `customers` method returns a typical list of customer names and `viewCust`: simply clears the Customer Detail pane and displays a few lines of sample text.

Once they got the synchronization between panes and the stream to the Customer Detail pane working, they "attached" the `Customers` dictionary to the `SalesCom` application by a quick rework to the `customers` method:

```
customers  
    "Return the keys of a Customers dictionary as a Sorted  
    Collection."  
    ^Customers keys asSortedCollection
```

And an enhancement of `viewCust`: was all that was needed to bring the `Customers` Dictionary "on-line":

viewCust: aCust

"Change: method response for the Customer List pane.
It updates the Customer Detail pane to show the
currently selected customer."

```
| custData custHist prepDate |
curCust := aCust.
custData := Customers at: curCust.
custDetPane cancel.
1 to: custData size - 1 do: [:index |
    replyStream
        nextPutAll: (custData at: index);
        cr].
custHist := custData at: 7. "Now print the history"
replyStream nextPutAll: '-----HISTORY-----'; cr.
1 to: custHist size do: [:index |
    replyStream
        nextPutAll: (custHist at: index);
        cr]
```

You can edit the methods to reflect these changes or paste them in from the tutorial chapter.12 file. Once these two methods are changed, you have reached another plateau in the SalesCom prototype development project. Both the Customers and Communication Events databases are up and accessible interactively in the SalesCom window as in Figure 12.6. Evaluate the expression **Test open** to see it.

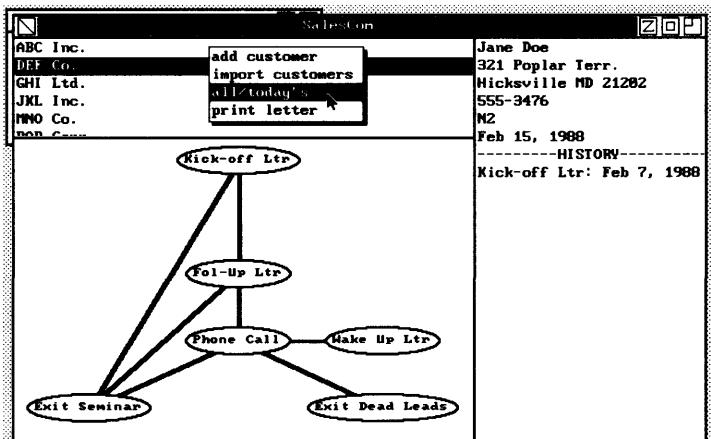


Figure 12.6
SalesCom Window
Phase Two

Menu selections foreshadow but don't yet deliver the full SalesCom environment. Menu selections result in message not understood walkbacks. The rest of the prototype development consisted of writing small methods to perform the actions requested by each of the menu items.

It's Getting Better All the Time: Evolutionary Development

The reps were on a roll. Every method they wrote gave them new insights and ideas for the next. They found that Smalltalk/V lent itself to "team development" as they took turns, one on the keyboard, the other scouring the manual. And by experience, they came up with another tip.

Survival Tip # 7: Keep moving. Write the methods that "jump out at you" and practically write themselves. If you get stuck on a method, put its body in comment quotes and save it. Then go on to the next method you need to write. Something you pick up solving another method's implementation or sometimes just some "creative gestation" will bring you back to unfinished methods. Eventually the toughest methods seem to give way when they are the only thing standing between you and your application running.

To see what the reps were able to accomplish with SalesCom on their maiden Smalltalk/V development project, evaluate:

(File pathName: 'class12.3in') fileIn.
Test open

While they have a long wish list, the reps now had a prototype that met their original SalesCom development objective, see Figure 12.7. With the pop-up mail merge and phone call script windows, SalesCom was a lot more sophisticated a program than the reps ever thought they could have written.

Take a few minutes to take SalesCom on a shake-down cruise. Add a customer and update somebody's contact information. Print a few letters. Make a phone call or two. Push that initial bunch of prospects through a hypothetical sales campaign. Sell a couple of WORMS. Lose a couple to the Dead Leads bin.

Notice how the node highlighting pinpoints the position of the selected customer in the marketing campaign. Track how printing a letter or making a call updates both the customer and communication event records.

To see what makes SalesCom tick, poke around the methods which you have just filed in. Note that while a method like `viewCust:` now looks complex with all the data handling that has found its way in to support the additional interface features, `viewCust:` became complex in stages. The reps wrote simple methods that became more sophisticated a line or two at a time.

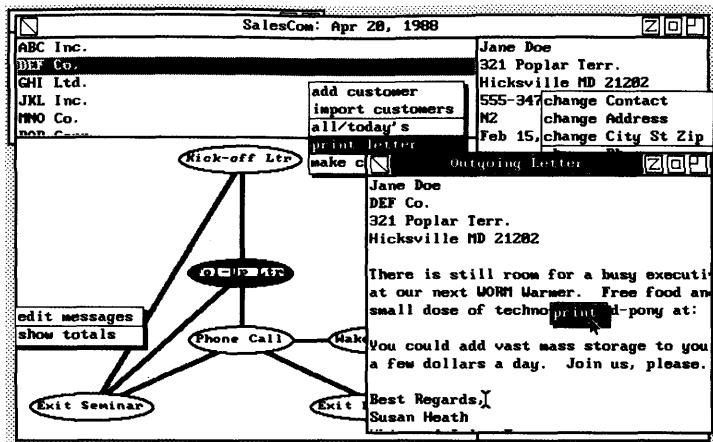


Figure 12.7
Full SalesCom
Prototype

For the record, over the course of the prototype development project, the reps were able to get all the behavior you see exhibited in SalesCom by:

- creating two new classes, and
- writing about thirty new methods.

The reps began using SalesCom the day after they finished the prototype. They had a long list of anticipated changes but they found that use helped focus their subsequent development. Rather than add features that they thought would be useful, they wrote those that their daily use convinced them they needed.

Where to Go from Here

Like many Smalltalk/V applications, SalesCom begs extension. While the prototype was fully functional and positively impacted the reps sales performance, over the next few months they intend to do the following:

- Add batch printing of daily correspondence, including envelope printing, to the mail merge capabilities of SalesCom.
- Make the network of strategy nodes dynamic so exits of one strategy network lead to the top of other strategic clusters of communication event nodes.
- Add a management report window to show month and year to date totals, sales total, and "body counts" of customers in each node of the network as well as compute the average days delay between one event and another, etc.
- Use Smalltalk/V Goodies #2 to use the FieldPane class to create good looking, easy to use Customer and Communication Event data entry and editing interfaces. This will allow users to "tab" from field to field rather than using the SalesCom prototype Inspectors and Prompters.

To take SalesCom to an even higher level of sophistication, the reps (or you) might consider extensions into the realm of discrete event simulation and expert systems:

- Use the multi-tasking capabilities provided by the **Process** and **Semaphore** classes of **Smalltalk/V** to add an Event Driven Simulation component to **SalesCom**. Such an extension would take the **SalesCom** databases' current state and use what it knows about response tendencies to generate simulation data to evaluate future impacts of current marketing decisions.
- Use the backward chaining inference capabilities available from the logic programming extensions of the **Prolog** class supplied as an example program to create a **SalesCom** Expert System. Based on sales management "rules", the **SalesCom** Expert System should be able to analyze data from **SalesCom** Event Driven Simulation runs. The result of this analysis would be a series of recommendations as to how to adjust sales activity now to avoid bottlenecks and "feast or famine" cycles in the future.

As you can see, the potential of **SalesCom** is great and will be reached by a process of evolutionary development. From prototype to bigger and better. With **Smalltalk/V** the limits are those of your imagination and the time and energy you have for development.

What You've Now Learned

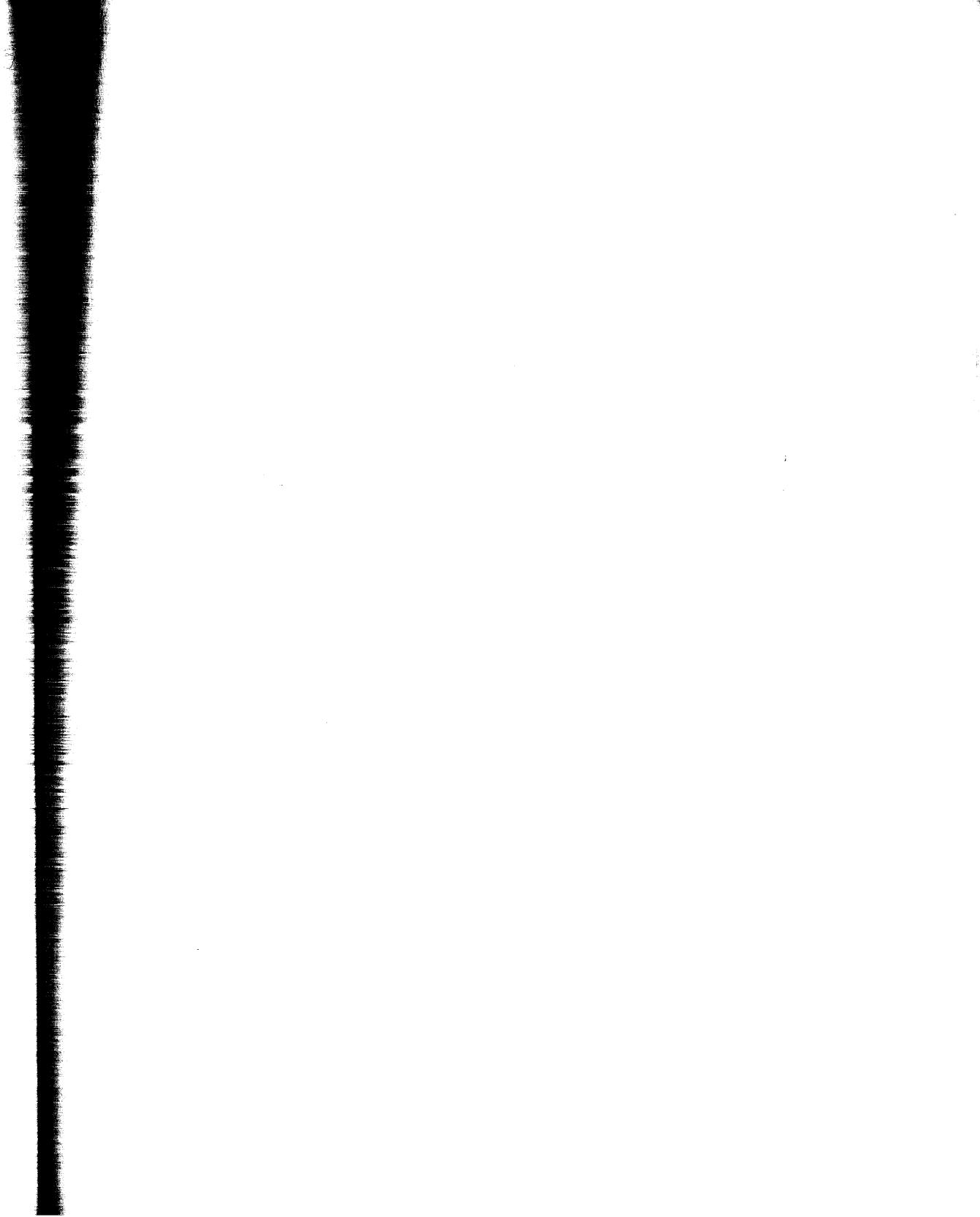
In the last chapter, we introduced the **Smalltalk/V** application development cycle. In this chapter, we've shared a number of tips to help speed your application development:

- *Don't over-plan. Know enough to get going, then do. Smalltalk/V is designed to work with you in an evolving, incremental process.*
- *When on cut and paste "raids" of existing methods, turn source class instance variable references into temporary variable assignments until you determine the destination class' data structure requirements.*
- *When re-working a copied method, edit the "comment" first. Then let the changes made to the comment point the way to lines in the method source code which implement those aspects of the method whose description in the comment has changed. And remember, comments are a critical means of communication among Smalltalk/V programmers.*
- *Simplified, interim versions of methods are an excellent way to "divide and conquer" a Smalltalk/V programming problem. Often this means writing a return expression which is a "hardwired" instance of the type of message a method will return once its more complex computation is written.*
- *Use the Encyclopedia of Classes in Part 4 of the Smalltalk/V manual. And use the Methods Index in the Appendices to direct your cross-referencing exploration of Smalltalk/V classes and their methods.*

- *Keep moving.* Keep your productivity high by writing the "easy ones" first. Very often the act of writing other methods or incubating on a tough one will result in insights to conquer almost any Smalltalk/V programming problem.
- *Know how to bounce back from errors,* like using `Scheduler reinitialize` to kill hard-to-find instances.

Keep these tips in mind as you tackle your first Smalltalk/V programming project. The remaining three sections of this manual are provided as reference tools. Part 3, **The Smalltalk/V 286 Reference**, covers in greater detail all the material explored in these tutorials. Part 4, **The Encyclopedia of Classes**, exhaustively describes the classes in the Smalltalk/V environment, including each classes' instance and class variables and methods. The **Appendices** contain a variety of sections on advanced and special topics as well as that ever-so-important **Methods Index**.

If you want to review any of the topics covered in this tutorial, you can either repeat the corresponding section of the tutorial, or refer to the detailed description in Part 3.



Part 3

Smalltalk/V 286 Reference

13 THE SMALLTALK LANGUAGE

Smalltalk is easy to learn and use because it has simple syntaxes and semantics, and few concepts. The concepts *object*, *class*, *message*, and *method* form the basis of programming in Smalltalk. The methodology for using Smalltalk consists of:

- Identifying the objects appearing in the problem and its solution.
- Classifying the objects according to their similarities and differences.
- Designing messages which make up the language of interaction among the objects.
- Implementing methods which are the algorithms that carry out the interaction among objects.

Objects

Objects are self-describing data structures. Every object is an instance of a class. The class of an object is determined by sending it the message **class**. This message is understood by all objects.

Objects are protected data structures. The data stored inside of an object is accessible only through messages. Objects can also be shared.

The word **self** in a method refers to the receiver object of the message that invokes the method.

Variables

All Smalltalk *variables* are containers for objects. A variable contains a single object pointer.

The variable name can be used in an expression to refer to the object whose pointer it contains. A variable may contain different object pointers at different times. The object pointer contained in a variable changes when an assignment expression is evaluated. An assignment makes a copy of a pointer to an object, not a copy of the object itself.

Variables are either private or *shared*. Private variables are accessible only to a single object. Shared variables are accessible to multiple objects. A private variable has a lower-case first letter, while a shared variable has an upper-case first letter.

There are three kinds of variables:

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

2. *Temporary variables* are created during the activation of a method. They exist for the lifetime of the method activation.
3. *Shared variables* are shared by many objects. They exist until explicitly deleted.

Instance Variables

Each object maintains its own internal state. The private memory of an object consists of its individually accessible components called *instance variables*. Instance variables are similar to fields of a record structure in other languages. Instance variables either have a name or are referred to with an integer index. *Named* instance variables are accessed by using their name. *Indexed* instance variables are accessed only through messages (usually using `at:put:` with integer indices). Each member of a class has its own separate instance variables.

Instance variables have a type—they contain either *pointers* or *bytes*. All instance variables for objects belonging to the same class are the same type. Most objects' instance variables contain pointers. The pointers refer to objects. If an object contains bytes, then its instance variables contain eight-bit values representing elementary data values.

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

Only the instance variables of the receiver of the message that invoked the method can be referred to by name.

Temporary Variables

Temporary variables include *method arguments* and *method temporaries*, and contained *block arguments*.

Method arguments are assigned the associated message arguments for the message which caused method invocation. Method temporaries are initialized to nil upon method invocation. Block arguments are assigned the associated message arguments for the

value: message at the time the block is activated. When a block is invoked while its containing method is still active, the block and the containing method share the same temporary variables.

Shared Variables

Shared variables are defined in dictionaries called *pools*. Different kinds of shared variables are defined in different kinds of pools. All shared variable names start with an upper-case letter. The variable name and the variable value are bound together into an object that is an instance of class **Association**. This association object is placed in the pool.

The System Dictionary **Smalltalk** is a pool which contains all the global variables.

Global variables are accessible from every object.

Class variables for each class are implicitly collected into a pool for the class. Class variables are defined as part of the class specification. Class variables are accessible only to the class, subclasses, instances of the class, and instances of the subclasses.

Pool variables are contained in named pool dictionaries that you explicitly construct. Pool dictionaries are global variables. To make pool variables accessible to a class and its instances, you must modify the class specification.

Classes

Classes are the program modules of Smalltalk because they describe data structures (objects), algorithms (methods), and external interfaces (message protocol). Classes provide complete capabilities to solve a particular problem.

Every object is an instance of some class. All objects which are instances of a class are similar because they have the same structure (i.e., the same instance variables), the same messages to which they respond, and the same available methods.

Classes are also objects contained in global variables which are maintained in the System Dictionary **Smalltalk**. As such, class names begin with a capital letter. This allows classes to be referred to in an expression.

The Class Hierarchy

Classes form a hierarchy, consisting of a root class, called **Object**, and many subclasses. Each class inherits the functionality of all its superclasses in the hierarchy. Class **Object** provides the common behavior for all objects. It includes methods for printing an object symbolically, for testing the class of an object, and for making a copy of an object. Each subclass builds on its superclasses by adding its own methods and instance variables to complete the implementation of the subclass's behavior.

The complete **Smalltalk/V** class hierarchy is shown on the next page. Indentation is used to show subclass relationships.

Object	TopDispatcher
Behavior	DispatchManager
Class	DisplayObject
MetaClass	DisplayMedium
BitBlt	Form
CharacterScanner	BiColorForm
Pen	ColorForm
Animation	DisplayScreen
Commander	ColorScreen
Boolean	DOS
False	File
True	Font
ClassBrowser	Icon
ClassHierarchyBrowser	InputEvent
ClassReader	Inspector
Collection	Debugger
Bag	DictionaryInspector
IndexedCollection	Magnitude
FixedSizeCollection	Association
Array	Character
CompiledMethod	Date
Bitmap	Number
ByteArray	Float
FileHandle	Fraction
Interval	Integer
String	LargeNegativeInteger
Symbol	LargePositiveInteger
OrderedCollection	SmallInteger
Process	Time
SortedCollection	Menu
Set	Message
Dictionary	Pane
IdentityDictionary	SubPane
MethodDictionary	GraphPane
SystemDictionary	ListPane
SymbolSet	TextPane
Compiler	TopPane
LCompiler	Pattern
Context	WildPattern
HomeContext	Point
CursorManager	ProcessScheduler
NoMouseCursor	Prompter
DeletedClass	Rectangle
DemoClass	Semaphore
Directory	Stream
DiskBrowser	ReadStream
Dispatcher	WriteStream
GraphDispatcher	ReadWriteStream
PointDispatcher	FileStream
ScreenDispatcher	TerminalStream
ScrollDispatcher	StringModel
ListSelector	TextSelection
TextEditor	UndefinedObject
PromptEditor	

Inheritance

A class inherits all of its superclasses' instance variables, class variables, and methods. Inheritance of class variables allows the methods of a class to refer to the class variables defined in its superclasses.

Inheritance of instance variables allows the methods of a class to refer to the instance variables defined in its superclasses, but it also means that superclass instance variables are included in objects which are instances of the class.

Determining what method to perform starts with two pieces of information: the message selector and the class of the receiver of the message.

First, the available methods for the class of the receiver are examined to see if there is a method which matches the message selector. If so, that method is performed. If not, the superclass of the class of the receiver is used, and the check for a method matching the selector is performed again. This checking for a matching method and advancing to the superclass is repeated until the method is found or until the end of the superclass chain is reached. In the latter case, a programming error occurs, and a message which describes the error is sent to the receiver of the original message.

There is a special syntax form for a receiver, **super**, which changes the initial class used for message lookup. The word **super** has two implications.

1. It represents the same object as **self** does, the receiver of the message which caused the method containing **super** to be performed.
2. It causes message lookup to start in the superclass of the class containing the method in which **super** appears, rather than starting in the class of the receiver.

The major purpose of a message to **super** is to be able to use a method in a superclass which is redefined in a subclass.

Class Messages

Messages to class objects are used for creating instances of the class and for initializing class variables. the most common messages for creating new instances are **new** and **new:**. Some classes define their own messages for creating instances.

Like all objects, classes know to which messages they can respond. For other objects, the methods available are determined by the object's class. Class objects, too, belong to a "class," called a *metaclass*, which determines the messages to which the class can respond.

There are three important classes relating to metaclasses:

1. **Metaclass**—the class of all metaclasses.
2. **Class**—the superclass of all instances of **Metaclass**.
3. Every metaclass has exactly one instance: the class of which it is the metaclass.

Specifying a New Class

In order for you to add a new class, you first choose a superclass on which you will build. Make the new class a subclass of the chosen superclass, then add the instance variables and methods necessary to complete the new class's functionality. Classes are normally specified using a Class Hierarchy Browser. The following describes the information which defines a class.

Classes are defined by sending a message to the new or modified class's superclass with class specification information as arguments. The class information that can be specified is the following:

- The class name
- Whether objects of the class contain pointers or bytes
- Whether objects of the class can contain indexed instance variables
- The names of the named instance variables for objects of the class
- The names of the class variables available to all objects of the class
- The names of the pool dictionaries which define shared variables available to objects of the class and possibly other classes

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

- 1) **subclass: subclassSymbol
instanceVariableNames:
instanceVariableNameString
classVariableNames: classVariableNameString
poolDictionaries: poolDictionaryNameString**
- 2) **variableSubclass: subclassSymbol
instanceVariableNames:
instanceVariableNameString
classVariableNames: classVariableNameString
poolDictionaries: poolDictionaryNameString**
- 3) **variableByteSubclass: subclassSymbol
classVariableNames: classVariableNameString
poolDictionaries: poolDictionaryNameString**

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

The third message defines classes whose member objects contain bytes. Objects with bytes contain only indexed instance variables, so there is no instance variable name string argument. Objects with bytes define elementary data values such as strings of characters.

Messages and Methods

All processing in a Smalltalk system involves sending messages to objects. Messages are the language of interaction which you use in order to express your computing requirements to objects. Messages request services from an object in terms of its external interface.

Methods are the algorithms which are performed by an object in response to receiving a message. Methods represent the internal details of the implementation of an object.

Protocol definitions for a class always have two parts—*class* methods and *instance* methods.

Class methods implement the messages sent to the class. The receiver of a class message is always the class object, not an instance of the class. All classes are global variables and can be referred to by their names.

Instance methods implement messages sent to instances of the class. The receiver of an instance message is always an object that is an instance of the class.

A method contains a sequence of Smalltalk expressions. There are four types of expressions:

1. Literals:

#aSymbol #(1 2 4 16) 'magic'

2. Variable names:

Smalltalk x replacementCollection

3. Message expressions:

bag add: stream next
100 factorial
array at: index + 10 put: Bag new

4. Blocks of code:

[:x :y | x name < y name]

The beginning of a method defines its name, arguments, and any temporary variables that it uses.

Sending a message involves:

1. Identifying the object to which the message is sent (the *receiver* of the message).
2. Identifying the additional objects that are included in the message (the *message arguments*).
3. Specifying the desired operation to be performed (the *message selector*).
4. Accepting the single object that is returned as the message *answer*.

The following sections present the syntax of methods and messages both informally with examples and more precisely using a syntax metalanguage. The metalanguage definition appears in Appendix 1. If you find that the informal presentation is sufficient, you can skip over the syntax rules. A complete syntax summary and cross reference are also presented in Appendix 1.

The Syntax of Variable Names and Literals

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

Variable Names

A variable name identifies a variable in an object. A variable is a container for an object pointer. A variable name is a sequence of letters and digits, beginning with a letter. Example variable names are:

OrderedCollection aString elements x2

Variable names beginning with an upper case letter represent shared variables, while those beginning with a lower-case letter represent private variables. The rules for variable names are:

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

Literals

A literal defines an object of class **Number**, **String**, **Character**, **Symbol**, or **Array**. Examples are given below where each of the possible literal forms is defined. The syntax rule for literals is:

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

Numbers

Numbers are objects of class **Float**, **Fraction**, or **Integer**. If a number contains a decimal point, it is an object of class **Float**. If it contains a negative exponent and no decimal point, it belongs to class **Fraction**. All other numbers belong to class **Integer**. If the number includes **r**, the digits preceding **r** define the number radix. In this case, capital letters are used to represent digit values greater than 9, with **A** = 10, **B** = 11, etc. Example numbers are:

```
15      16rFF     3.1416     1e-3     -100
```

The rules for numbers are:

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

Strings

A string is a sequence of characters enclosed in apostrophes. It is an object of class **String** which is a sequence of objects of class **Character** that can be indexed. Strings are not necessarily constant; their characters may be changed by sending a message to the string. Paired apostrophes within a string reduce to a single apostrophe in the resultant string object. Example strings are:

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

The rules for strings are:

```

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

```

Comments

A comment is a sequence of characters enclosed in double quotes. A comment is ignored anywhere within a method, except when occurring within a string. Example comments are:

"Answer the size of the receiver" "goodBye"

The rule for comments is:

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

Character Constants

A character constant is an object of class **Character**. A character constant appears as a dollar sign followed by any character. Example character constants are:

\$\$ \$a \$' \$ \$.

The rule for character constants is:

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

Symbols

A symbol is an object of class **Symbol**, a sequence of objects of class **Character** which can be indexed. Symbols differ from strings in that their characters may not be changed. A symbol constant identifies the associated symbol object. The form of a symbol constant is a number sign, #, followed by the characters of the symbol. Example symbol constants are:

#+ #asOrderedCollection #at:put: #==

The rules for symbols and symbol constants are:

```

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

```

Arrays

An array is an object of class **Array** which may be indexed by an integer from one through the size of the array. An array is a series of literals enclosed in parentheses. An array constant identifies the associated array object. It consists of an array preceded by a number sign. Example array constants and arrays are:

```

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

```

The rules for arrays and array constants are:

```

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

```

Expression Syntax

The actions in a method are specified by a series of expressions separated by periods. A period is optional after the last expression of the series. Each expression computes a single object as its result. The expression may also include assignment of its result to one or more variables.

The final expression in an expression series may be preceded by a caret, \wedge . The caret means that method execution terminates and answers the object computed by the expression.

The rules for expressions and expression series are:

```

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

```

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

A unary expression sends a series of unary messages which are evaluated from left to right. A unary message has no arguments.

A binary expression sends a series of binary messages which are evaluated from left to right. A binary message has a single argument following the binary selector. The traditional arithmetic operators are implemented in Smalltalk using binary expressions. This gives all arithmetic operators the same precedence. Parentheses may be used to specify other than left-to-right evaluation.

A keyword expression sends a single keyword message with one or more arguments. The arguments to a keyword message are evaluated from left to right.

The selector of a keyword message is the concatenation of all the keywords in the message.

Unary expressions have highest precedence, followed by binary and then keyword. Parentheses may be used to specify a different evaluation order.

Cascaded messages are a series of messages to the same receiver. Each message after the first is preceded by a semicolon.

The rules for message expressions are:

```

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

```

Blocks

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

A block may have arguments. These are specified between the left bracket and vertical bar by preceding each block argument variable name with a colon.

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

A block with one argument is executed by sending it the message `value:.` The argument to the `value:` message is assigned to the block argument upon block execution.

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

A block may contain an expression preceded by a caret, `^`. Evaluation of such an expression causes termination of execution for both the block and the method in which the block appears.

Blocks are the basis for control structures in Smalltalk. Since control structures conform to keyword message syntax, control structures have no special syntax.

The rule for blocks is:

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

Method Syntax

A complete *method specification* includes a message pattern, optional primitive number, optional temporaries, and an expression series. The message pattern specifies how to send a message to request method execution. It includes the method selector and the variable names used to refer to arguments within the method.

The rules for method syntax are:

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

Control Structures

Control structures are invoked by sending messages with blocks as arguments. Three forms, with several variations, are predefined in the Smalltalk language. You may define additional forms in Smalltalk using these predefined ones.

Conditional Execution

The following predefined conditional execution messages are available:

```
ifTrue: aBlock
ifFalse: aBlock
ifTrue: trueBlock ifFalse: falseBlock
ifFalse: falseBlock ifTrue: trueBlock
```

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

Iterative Execution

The following predefined iterative execution messages are available:

```
whileTrue: aBlock
whileFalse: aBlock
```

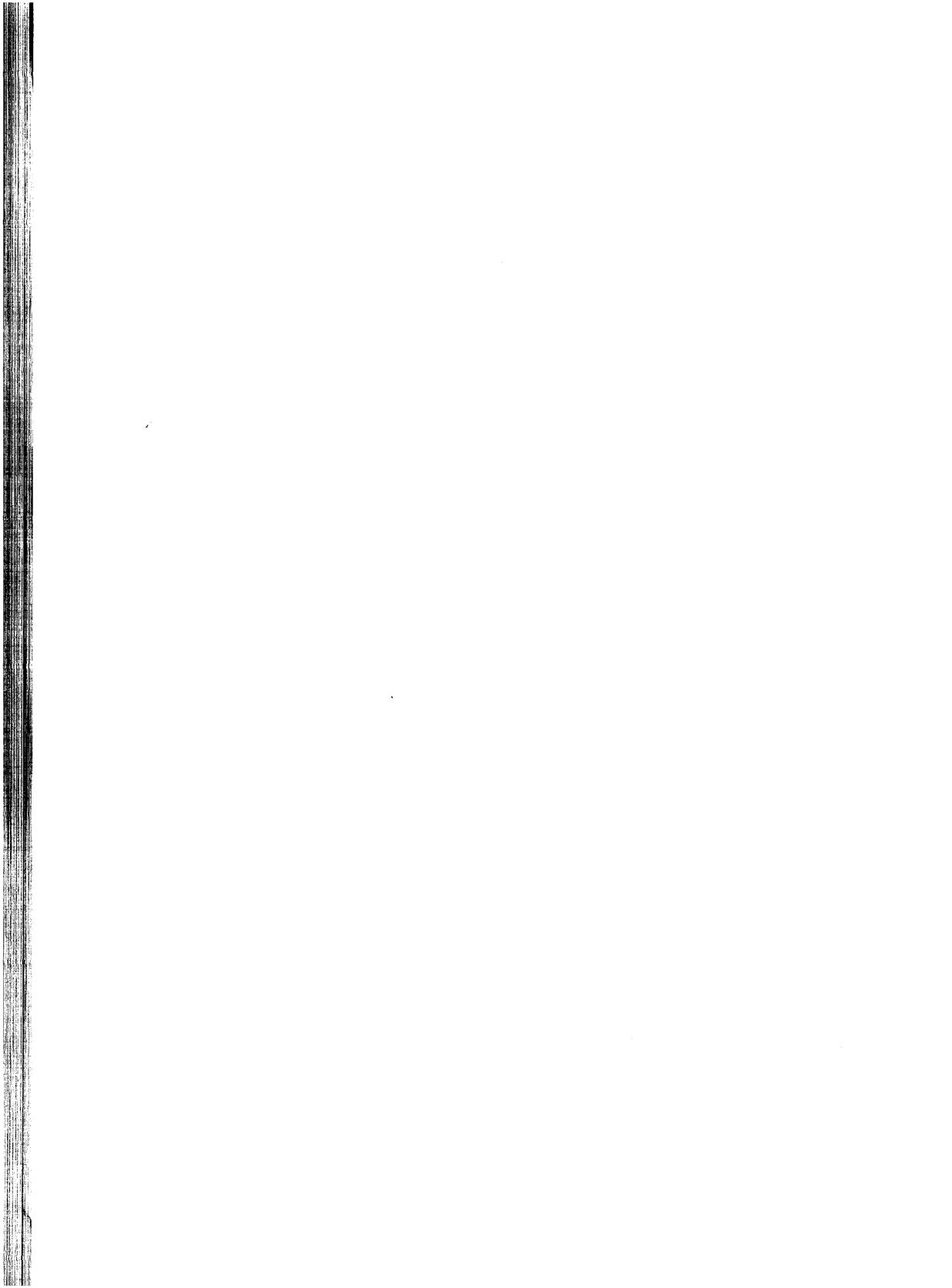
Both the receiver and argument of these messages must be no-argument blocks. For **whileTrue:**, the receiver block is sent the message **value**. If it answers true, the argument block is sent the message **value**. The iteration continues until the answer of the first block evaluation is false. For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

Short Circuit Boolean Evaluation

The following predefined boolean operators are available:

```
and: aBlock
or: aBlock
```

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



14 SMALLTALK/V 286 CLASSES

This section describes the major Smalltalk/V classes. These classes serve as the basic building blocks for your applications.

Magnitudes

The **magnitude** classes are the easiest to understand and the most frequently used. They define objects that can be compared, measured, ordered, and counted. These include characters, numbers, dates, and times. Many useful messages for comparing, testing, and ordering these objects are defined. The arithmetic operators and many useful numerical functions are also defined as messages understood by the numerical magnitude objects.

This chapter presents a quick overview of each of the magnitude classes provided in Smalltalk/V. Examples are used to demonstrate some of the functionality provided. Part 4: The Encyclopedia of Classes gives a detailed specification of each of the magnitude classes.

The **Magnitude** class hierarchy shown below lists all of the magnitude classes.

```
Magnitude
  Association
  Character
  Date
  Number
    Float
    Fraction
    Integer
      LargeNegativeInteger
      LargePositiveInteger
      SmallInteger
  Time
```

Magnitude

All of the magnitude classes are subclasses of the abstract class **Magnitude**. Class **Magnitude** provides the comparing and ordering protocol inherited by its subclasses. All magnitudes support comparing, ordering, and interval testing. **Magnitude** assumes its subclasses implement the ordering relation and comparison methods: `=`, `<=`, `>=`, `<`, `>`, `~`. Based on these methods, **Magnitude** provides generic methods for interval testing and max/min computation inherited by all **magnitude** classes. Some numerical examples are:

<u>Expression</u>	<u>Answer</u>
<code>46 > 33</code>	true
<code>46 min: 33</code>	33
<code>46 max: 33</code>	46
<code>5/4 between: 0.5 and: 1</code>	false

Character

The instances of class **Character** are the extended ASCII character set from ASCII value 0 to ASCII value 255. Characters are pre-existing objects in Smalltalk, hence they do not have to be created. References to characters are made in two ways: as literals or by converting integers into the corresponding ASCII character. There are two conversion messages. The message `asCharacter` can be sent to an integer, or the message `value:` with an integer argument can be sent to class **Character**. For example:

<u>Character</u>	<u>Literal</u>	<u>Equivalent Expression</u>
A	\$A	65 asCharacter
B	\$B	66 asCharacter
C	\$C	Character value: 67
space	\$	32 asCharacter
line feed		10 asCharacter
tab		Character value: 9

Like all subclasses of **Magnitude**, the class **Character** must define how characters are compared and ordered. The methods `<`, `<=`, `=`, `>=`, `>`, and `~` compare characters by comparing their ASCII values. For example:

<u>Expression</u>	<u>Answer</u>
<code>\$a = \$A</code>	false
<code>\$A < \$B</code>	true

The interval testing and min/max methods are inherited from class **Magnitude** automatically:

<u>Expression</u>	<u>Answer</u>
69 asCharacter max: \$A	\$E
\$x between: \$a and: \$t	false

Class **Character** has many testing and conversion methods. Some examples follow:

<u>Expression</u>	<u>Answer</u>
\$a isUpperCase	false
\$a isLowerCase	true
\$a asUpperCase	\$A
\$? asLowerCase	\$?
\$e isVowel	true
\$+ isLetter	false
\$9 isDigit	true
\$A asciiValue	65

Date and Time

Instances of class **Date** represent specific dates such as January 1, 1980 or September 15, 1876. Instances of class **Time** represent specific times of the day such as 10 am or 12:15 pm.

Dates and Times are created by evaluating expressions. The detailed descriptions of classes **Date** and **Time** in Part 4 give a complete list of the messages supported by **Date** and **Time**. Some examples are:

```
Time now
Date today
'20 January 1950' asDate
Date newDay: 20 month: #Jan year: 1950.
```

The following code makes an instance of class **Date** and puts it in the global variable **Birthday**:

```
Smalltalk at: #Birthday put: '4 August 1976' asDate
```

Ordering and comparing of dates and times are supported. Some examples of messages supported by dates are:

<u>Expression</u>	<u>Answer</u>
Birthday year	1976
Birthday dayName	Wednesday
Birthday > Date today	false
Birthday min: Date today	'August 4, 1976'
Birthday previousWeekday: #Saturday	'July 31, 1976'
Birthday daysLeftInYear	149
Birthday daysInYear	366

You can add new ways of creating objects by defining new methods. If you add the following method as a class method in class Time:

```
hour: hours minute: minutes seconds: seconds
      "Answer an instance of class Time as specified"
      ^self fromSeconds:
        (((hours * 60) + minutes) * 60) + seconds
```

then you can create instances of class Time using expressions like the following:

```
Smalltalk at: #LunchTime
      put: (Time hour: 12 minute: 0 second: 0).
Smalltalk at: #DinnerTime
      put: (Time hour: 18 minute: 45 second: 0).
Smalltalk at: #BreakfastTime
      put: (Time hour: 7 minute: 30 second: 0).
```

Some examples using these new global variables are:

<u>Expression</u>	<u>Answer</u>
LunchTime	true
between: BreakfastTime and: DinnerTime	
LunchTime min: DinnerTime	12:00:00
DinnerTime hour	s18
LunchTime < BreakfastTime	false

Number

Smalltalk supports three kinds of numbers: floating point (class **Float**), rational (class **Fraction**), and integer (class **Integer** and its subclasses). The methods of class **Number** define the general behavior of its subclasses, support mixed mode arithmetic, and provide many useful numeric, testing, and iteration functions.

Number defines the arithmetic protocol that its subclasses must implement. These are the usual binary arithmetic operators: **+**, **-**, *****, **/**. There is equal precedence between all binary operators, so evaluation is left to right. Some examples:

<u>Expression</u>	<u>Answer</u>
3 + 4	7
3 + 4 * 2	14
2 + 4 / 12	1/2

Number implements many numerical methods that its subclasses can inherit such as: **exp**, **cos**, **arcSin**, **tan**, **ln**, **sqrt**, **floor**, **reciprocal**. Some examples:

<u>Expression</u>	<u>Answer</u>
7.5 floor	7
4 reciprocal	1/4
-2.3 abs	2.3

Number implements many testing methods inherited by its subclasses such as: **even**, **positive**, **strictlyPositive**. Some examples:

<u>Expression</u>	<u>Answer</u>
4 even	true
0.1 positive	true
0 strictlyPositive	false

Number implements methods for creating other kinds of objects, such as:

<u>Expression</u>	<u>Answer</u>
2 @ 7	A Point with x coordinate of 2 and y coordinate of 7
1/4 to: 3/4 by: 1/8	An Interval containing the fractions 1/4, 3/8, 1/2, 5/8, 3/4

Number also implements iteration methods, such as:

```
1/4 to: 1.5 by: 1 do: [:i|
  Transcript space; nextPutAll: i printString; cr]
```

which prints the numbers 1/4 and 5/4 in the Transcript window.

Smalltalk/V supports mixed mode arithmetic so that arithmetic expressions can be composed of different kinds of numbers. Executing sample expressions is the best way to understand the conversion rules.

<u>Expression</u>	<u>Answer</u>	<u>Comment</u>
1 + 2	3	
5.1 - 3	2.1	Note space between - and 3
2 * -4.0	-8.0	Mixed mode gives a Float
2/4	1/2	A fraction
1/2 + 1	3/2	Mixed mode gives a Fraction
1/2 + 1.0	1.5	Mixed mode gives a Float
4/2	2	Fraction reduces to an integer

The following examples explain many of the messages that can be used with numbers.

<u>Expression</u>	<u>Answer</u>	<u>Comment</u>
<code>4 // 3</code>	1	Integer quotient
<code>-4 // 3</code>	-2	Truncate toward minus infinity
<code>4 \\\ 3</code>	1	Integer remainder
<code>-4 \\\ 3</code>	2	Integer remainder, truncates as //
<code>-4 quo: 3</code>	-1	Integer quotient, truncate toward zero
<code>-4 rem: 3</code>	-1	Integer remainder, truncate toward zero
<code>-2.3 abs</code>	2.3	Absolute value
<code>10 negated</code>	-10	
<code>11 reciprocal</code>	1/11	A fraction
<code>2 + 3 * 4</code>	20	Evaluation is left to right
<code>3 - (2 * 2)</code>	-1	Parentheses change evaluation order
<code>2 + 3 negated</code>	-1	Unary operator (negated) done first
<code>6 quo: 2 + 1</code>	2	Keyword operator (quo:) done last
<code>2 sqrt</code>	1.414236	Square root
<code>4 sqrt</code>	2.0	Answer always float
<code>2.1 squared</code>	4.41	Receiver times itself
<code>2.3 even</code>	true	
<code>2 odd</code>	false	
<code>10 negative</code>	false	
<code>0 positive</code>	true	True if ≥ 0
<code>0 strictlyPositive</code>	false	True if > 0
<code>-0.1 sign</code>	-1	
<code>0 sign</code>	0	
<code>100 sign</code>	1	
<code>5.1 ceiling</code>	6	Nearest integer greater than or equal
<code>-5.1 ceiling</code>	-5	
<code>5.1 floor</code>	5	Nearest integer less than or equal
<code>-5.1 floor</code>	-6	
<code>5.1 truncated</code>	5	Nearest integer toward zero
<code>-5.1 truncated</code>	-5	
<code>5.1 rounded</code>	5	Nearest integer
<code>5.1 truncateTo: 2</code>	4	Nearest argument multiple toward zero
<code>5.1 truncateTo: 2.3</code>	4.6	
<code>5.1 roundTo: 2</code>	6	Nearest argument multiple
<code>5.1 roundTo: 2.3</code>	4.6	
<code>5 exp</code>	148.41316	Exponential
<code>2.7182819 ln</code>	1.0	Natural logarithm
<code>4 log: 2</code>	2.0	The logarithm in the base of the argument
<code>3 raisedTo: 1.1</code>	3.3483695	The receiver to the power of the argument

<u>Expression</u>	<u>Answer</u>	<u>Comment</u>
4 raisedToInteger: 3	64	The receiver to the power of the integer argument
30 degreesToRadians	0.52359878	Convert degrees to radians
2 radiansToDegrees	114.59156	Convert radians to degrees
0.52359878 sin	0.5	Angle in radians
0.72273425 cos	0.75	
0.24497866 tan	0.25	
0.5 arcSin	0.52359878	Angle in radians
0.75 arcCos	0.72273425	
0.25 arcTan	0.24497866	

Float

An 8-byte IEEE format is used for instances of class **Float** to approximate real numbers. This gives approximately 18 digits of precision and represents values in the range $(+/-)4.19e-307$ to $(+/-)1.67e308$. The 8087 or 80287 arithmetic co-processor, depending upon your computer, must be present to perform arithmetic operations on floating operands.

Fraction

Instances of class **Fraction** are exact representations of rational numbers. A pair of integers (instance variables numerator and denominator) describes the fraction. Fractions are created by sending the slash (/) message to an integer with an integer argument (provided that the answer does not reduce to an integer).

Integer

Integers are frequently used in counting and indexing. Three subclasses of class **Integer** are defined: **LargeNegativeInteger**, **LargePositiveInteger** and **SmallInteger**. Instances of class **SmallInteger** are in the range -32767 to 32767. These are highly efficient in both computing speed and memory occupation. Small integers are encoded in the reference to the object (the object pointer); they are not represented as objects in memory. The large integer classes can represent numbers with up to 64K bytes of precision. Conversion between integer classes is automatic.

Streams

The stream classes are used for accessing files, devices, and internal objects as sequences of characters or other objects. Streams have an internal record of their current position. Streams also have access messages which get or put the next object at the current position and advance the stream's position by one. Messages are defined for changing the stream position so that random access is possible.

This chapter presents the purposes of and the protocol shared among the stream hierarchy classes. For a complete specification of each class, refer to Part 4: Encyclopedia of Classes.

The Stream class hierarchy is as follows:

```

Stream
 ReadStream
 WriteStream
  ReadWriteStream
    FileStream
    TerminalStream
  
```

Streams are frequently used for scanning input and writing edited output. The example which follows sends the message `printString` to an instance of class `String`. The answer to this message is a new string composed of the initial string (the receiver of `printString`) surrounded by quotes with any internal quotes doubled. For example:

<u>Expression</u>	<u>Answer</u>
'hello' <code>printString</code>	'hello'
'hello' <code>printString</code> <code>printString</code>	""hello""

The key to the following implementation of `printString` in class `String` is that an instance of class `WriteStream` automatically grows to contain all the characters written to it and responds to the message `contents` by returning a string containing all of its characters.

```

printString
| inputStream outputStream |
inputStream := ReadStream on: self.
outputStream := WriteStream on:
    (String new: self size + 2).
outputStream nextPut: $'.
[inputStream atEnd]
whileFalse: [
    character := inputStream next.
    outputStream nextPut: character.
    character == $
        ifTrue: [outputStream nextPut: $']].
outputStream nextPut: $'.
^outputStream contents

```

This example illustrates several stream messages.

Instances of classes **ReadStream** and **WriteStream** are created with the **on:** message with a string as the argument. Both streams are positioned at the first character. Note that in creating the **WriteStream** instance, space is provided for the containing quotes but not for interior paired quotes. If interior quotes exist, the string object affected by the **WriteStream** will automatically be enlarged.

Characters are written to the **WriteStream** with the message **nextPut:.** The character to write is the argument.

The end of a **ReadStream** is detected with the **atEnd** message. If there is a character at the current position, **atEnd** answers false; otherwise it answers true.

A character is read from the **ReadStream** with the message **next.** Note that you cannot send the message **next** to a **ReadStream** that is positioned at the end.

All of the characters in a **WriteStream** are returned as a string in answer to the **contents** message.

Accessing Protocol

We summarize the above information in the following protocol:

<u>Protocol</u>	<u>Explanation</u>
atEnd	Answer true if stream is at the end else answer false.
contents	Answer the collection of objects that is being streamed over.
next	Answer the next object in the receiver stream and advance the position by one.
nextPut: anObject	Write anObject at the current position. Answer anObject .

Positioning and Reading Protocol

Some of the stream positioning protocol is as follows:

<u>Protocol</u>	<u>Explanation</u>
position	Answer an integer representing the stream's position. The position at the beginning of the stream is zero.
position: anInteger	Set the stream position to anInteger . Report an error if anInteger is beyond the end of the stream.
reset	Set the stream's position to zero.
skip: anInteger	Add anInteger (which may be negative) to the stream's position.

Some stream reading protocol follows:

<u>Protocol</u>	<u>Explanation</u>
do: aBlock	Proceed through the stream from the current position to the end evaluating aBlock with each element of the stream as the block argument.

<u>Protocol</u>	<u>Explanation</u>
isEmpty	Answer true if the stream contains no elements; otherwise answer false.
next: anInteger	Answer a collection of the next anInteger elements of the stream. Advance the stream position by anInteger .
peek	Answer the next element in the stream without advancing the stream position. Answer nil if at end of stream.
peekFor: anObject	Answer true and advance the stream position if the next object in the stream equals anObject . Otherwise, answer false and leave the stream position unchanged.
skipTo: anObject	Set the stream position beyond the next occurrence of anObject in the stream or, if none, at the end of the stream. Answer true if there was an occurrence; otherwise answer false.
upTo: anObject	Answer a collection of objects starting at the current stream position and up to but not including the next object that equals anObject and advance the stream position beyond the object that equals anObject . If anObject is not in the stream, answer up to the end of the stream and set the stream position to the end.

The following example illustrates positioning and reading protocol using a stream on an array of symbols. First the stream is created and assigned to the variable **Colors**. Then a series of messages are sent to the stream **Colors**. The result of each message is shown below.

```
Colors := ReadStream on:  
#(red blue green yellow pink cyan magenta brown).
```

<u>Expression</u>	<u>Answer</u>
Colors isEmpty	false
Colors next	red
Colors next: 3	(blue green yellow)
Colors peek	pink
Colors peekFor: #blue	false
Colors upTo: #magenta	(pink cyan)
Colors skip: -4	
Colors position	3
Colors skipTo: #pink	true
Colors upTo: #red	(cyan magenta brown)

Writing Protocol

Some additional stream writing protocol follows.

<u>Protocol</u>	<u>Explanation</u>
nextPutAll: aCollection	Write the elements of aCollection to the stream. Answer aCollection.
next: anInteger put: anObject	Write anObject to the stream anInteger times. Answer anObject.
cr	Write a line-terminating character to the stream.
tab	Write a tab character to the stream.
space	Write a space character to the stream.

All objects understand the message **printOn:** with a stream as the argument. This message produces a character description of the receiver object on the argument stream. For example, the implementation of **printOn:** for class **Rectangle** is:

```
printOn: aStream
    "Print the origin and corner points"
    origin printOn: aStream.
    aStream nextPutAll: ' corner: '.
    corner printOn: aStream
```

where the **printOn:** message is sent to the origin and corner points and the message **nextPutAll:** to its stream argument. The implementation for class **Point** is:

```
printOn: aStream
    "Print the x and y coordinates"
    x printOn: aStream.
    aStream nextPutAll: ' @ '.
    y printOn: aStream
```

An example of printing a Rectangle is:

Display boundingBox printOn: Transcript

which writes the following in the System Transcript window if you are running in EGA color mode:

0 @ 0 corner: 640 @ 350

Interface to DOS File System

Class **FileStream**, a subclass of **ReadWriteStream**, provides the primary interface to the DOS file system. File streams respond to all of the stream protocol presented earlier. File streams use an instance of the class **File** to provide random page access to DOS files. Files use an instance of class **FileHandle** to read and write DOS file pages. The class **Directory** provides access to DOS disk directories.

In this section we present an overview of these file system classes. For detailed information on all of the messages that they provide, please see the descriptions in Part 4.

File Streams

File streams are usually created with either a message to class **File** specifying a partial or complete path name or a message to an instance of class **Directory** specifying a particular file to access in that directory. Here are some examples of messages to class **File**.

File pathName: 'c:\smalstalk\chapter.1'

File pathName: 'chapter.1'

File pathName: '\smalstalk\chapter.1'

The first expression above has a complete path name. The second example above is a partial path name. The directory object **Disk**, a global variable, is used to complete the path name. In this case the file '**chapter.1**' in the directory **Disk** is accessed. The final example is a complete path name without a disk drive specifier. The drive specifier used is the same as that used by the directory **Disk**.

The other way to create a file stream is by sending one of the following messages to a directory object.

Disk file: 'chapter.1'

Disk newFile: 'junk.fil'

The above two messages cannot have path names as arguments, only a file name. The difference between the two messages is that the second message will erase an existing file of the same name if one exists. They both will create the file if it does not already exist.

A word of caution about DOS files. DOS does not automatically update the directory entry on disk as you write to a file. There are two messages that you can send to file streams to cause the directory entry to be updated on the disk. These are:

stream close

stream flush

The difference between the two messages is that the first closes the file stream making further access to the DOS file using this object impossible. The second message also causes the directory entry to be updated but keeps the file stream object open for further access to the file. For consistency, all other streams support these two messages as well, but they have no effect.

File streams are buffered for efficiency. In addition, file streams recognize two different formats for end of line, the DOS cr-lf pair, and the UNIX single lf. When a file stream is opened, the beginning of the file is scanned to determine which format applies. New files are created using the DOS format. The following three messages let you test and change the line ending format for a file.

```

stream lineDelimiter      "Answers Lf or Cr"
stream lineDelimiter: Lf   "Change to Unix format"
stream lineDelimiter: Cr  "Change to DOS format"

```

The fastest way to read a file stream is with the **upTo:** or the **nextLine** message. The fast way to write a file stream is with the **nextPutAll:** message.

Putting all of the above together, here is a faster version of the program given in Chapter 7 of the tutorials which converts text files from DOS format to UNIX format.

```

"Convert a file from DOS format to UNIX format"
| input output |
input := Disk file: 'chapter.7'.
output := Disk newFile: 'stripped.7'.
output lineDelimiter: Lf.
[input atEnd]
  whileFalse: [output nextPutAll: input nextLine; cr].
output close

```

Directories

The class **Directory** provides access to DOS file system directories. Smalltalk/V as delivered has the following global variables which contain directories.

```

Disk
DiskA
DiskB

```

The variable **Disk** contains the directory in which you started Smalltalk/V. The variables **DiskA** and **DiskB** contain the root directories for the disk devices A: and B: respectively. You can create new directory objects using the following messages:

```

SampleDir := Directory pathName: 'a:\dirname'
DiskC := Directory new drive: $c; pathName: '\'

```

Note that creating a directory object is not the same as creating a directory on the disk drive itself. To create a new directory on the disk, send the message **create** to a directory object with the proper drive and path name, as in:

```
SampleDir create
```

Directories understand messages for listing their subdirectories and files, for creating new files and subdirectories, and much more. See Part 4 for more details.

Files and FileHandles

Class **File** provides the logical support to file streams necessary for random page access to DOS files. Class **FileHandle** provides the low level access to DOS files. Part 4 provides a detailed list of the messages implemented by these classes. Unless you are building some sort of new file access protocol separate from file streams, you will rarely have to deal with these classes. A few of the class messages for **File** are important: the ones for copying, renaming, and removing files.

FileHandle is a subclass of **ByteArray**. A **FileHandle** instance has a size of exactly two bytes long which contains the DOS file handle number when the file is opened. DOS allows only a limited number of file handles. When you try to open a file and no more file handles are available, **Smalltalk/V** automatically checks that all file handles are indeed used by some objects. For example, you might have opened a file and forgot to close it. As long as this handle is not pointed to by any object, **Smalltalk/V** will automatically reuse it to open a new file.

Terminal Input and Output

Terminal input and output is accomplished through the cooperation of two classes: **InputEvent** and **TerminalStream**. **InputEvent** is a lower level interface whose method **nextEvent** uses a primitive to read a keyboard or mouse event and return the type of the event read. **InputEvent** is interrupt driven. It usually waits on the **KeyboardSemaphore** (a global variable containing an instance of class **Semaphore**) until the semaphore is signaled by a key stroke or mouse operation.

CurrentEvent is the global variable used to read events. It contains an instance of **InputEvent**. The read primitive modifies the instance variables of **CurrentEvent**: **type**, **value**, **x**, and **y**. The location of the mouse at the time of the event is placed in **x** and **y**. The following table describes the events returned from the read primitive (left column), the types returned by the **InputEvent** (middle column), and the values associated with the types (right column).

<u>Primitive Type</u>	<u>nextEvent Type</u>	<u>Value</u>
#characterInput	#characterInput	ASCII code
#functionInput	#functionInput	scan code
#mouseMove	#mouseMove	
	#mouseStillDown	mouse button value
#mouseButton	#mouseButtonDown	1 = left button down 2 = right button down 3 = middle button down 5 = left button down shifted 6 = right button down shifted 7 = middle button down shifted
	#mouseButtonUp	1 = left button up 2 = right button up 3 = middle button up 5 = left button up shifted 6 = right button up shifted 7 = middle button up shifted
#nullEvent	#nullEvent	(none)

The scan code associated with #functionInput is defined by DOS. When a #mouseMove event is returned from the primitive, an InputEvent can generate either #mouseMove or #mouseStillDown depending on whether a mouse button is being held down (if yes, the latter one is generated). When a #mouseButton event is returned, an InputEvent further differentiates it into two types, #mouseButtonDown or #mouseButtonUp depending on whether it is a down or up action that has caused the event.

TerminalStream is defined as a subclass of **ReadWriteStream**. It uses **InputEvent** to read from the keyboard or mouse and sends messages to **CharacterScanner** to write characters to the terminal screen. The global variable **Terminal**, which contains an instance of class **TerminalStream**, is used throughout **Smalltalk/V** to handle terminal I/O.

TerminalStream reimplements the method **next** defined in **Stream** with the method **read**. Method **read** is the major user interface for reading a single character (or function key) from the keyboard or an event from the mouse. Two global variables, **FunctionKey** and **MouseEvent**, are set to true or false to indicate the source of the input before the input is returned. If a mouse event is read, the variable **MouseEvent** is set to true. If a function key is pressed, the variable **FunctionKey** is set to true. And non-function keys set both variables to false.

Method **nextPut:** outputs a character to the terminal screen at the current cursor position. It is seldom used in the system since writing characters to the screen is usually done through instances of **CharacterScanner** which has more sophisticated masking and clipping capabilities.

Programming Function Keys

When a function key is processed outside of class **Terminal-Stream**, the **read** method in class **TerminalStream** is usually invoked by method **processInput** in class **Dispatcher**. As soon as a function key is pressed, the **read** method returns the function code and sets the global variable **FunctionKey** to true and the global variable **MouseEvent** to false. Method **processInput** then calls method **processKey:** in the receiver dispatcher class (any of the dispatcher classes). The latter method checks the state of the two global Boolean variables, **MouseEvent** and **FunctionKey**. If either of the variables is true, then a function code must be processed. To do this, method **processFunctionKey:** in the receiver dispatcher is invoked to compare the function code to the values defined by the pool dictionary **FunctionKeys**. When a match is found, the appropriate action is taken.

Suppose you want to assign the **move** and **frame** window operations to F1 and F2. First, define names for them, for example **MoveWindowFunction** and **FrameWindowFunction** respectively. Then, enter these names into the pool dictionary **FunctionKeys** and associate them with the desired function key codes. You can do this by executing the following expression:

```
FunctionKeys at: 'MoveWindowFunction'  
    put: 59 asCharacter.  
FunctionKeys at: 'FrameWindowFunction'  
    put: 60 asCharacter
```

This creates two new variables in pool dictionary **FunctionKeys**. Note that 59 and 60 are the scan codes generated by F1 and F2 respectively.

The next step is to add the following code to method **processFunctionKey:** in class **Dispatcher** so that the desired functions occur when F1 and F2 are pressed.

```

MoveWindowFunction == aCharacter
ifTrue: [
    pane topPane hasCursor
    ifTrue: [ ^pane topPane dispatcher move ]
    ifFalse: [ ^Terminal bell ]].
FrameWindowFunction == aCharacter
ifTrue: [
    pane topPane hasCursor
    ifTrue: [ ^pane topPane dispatcher frame ]
    ifFalse: [ ^Terminal bell ]].

```

Collections

A **collection** is a group of related objects. The Smalltalk collection classes define several different data structures which serve as containers for arbitrary objects. For example, a *String* is a sequence of characters while a *Set* is an unordered collection of non-duplicated objects of any kind. The collection classes are useful because they provide similar protocol for:

1. Iterating over the elements of a collection.
2. Searching a collection for a particular element.
3. Adding and removing elements.
4. Accessing and changing elements.

The following is the Collection class hierarchy:

```

Collection
Bag
IndexedCollection
FixedSizeCollection
    Array
    ByteArray
    Interval
    String
    Symbol
OrderedCollection
SortedCollection
Set
Dictionary
IdentityDictionary

```

The attributes, conversions, and common protocol among various collections are discussed next with a description of each kind of collection following.

Attributes of the Collection Class

In general, each kind of collection can be characterized by four attributes:

1. Whether the collection has a well defined order associated with its elements. This order can be defined either externally by a key or internally by the contents of elements.
2. Whether the collection's size is fixed or expandable.
3. Whether or not duplicates of the collection's elements are allowed.
4. Whether the collection is accessible by a set of keys. Keys can be either integer indices or lookup keys.

The following table shows the attributes of each class:

	Fixed				Element
	<u>Ordered</u>	<u>size</u>	<u>Dup's</u>	<u>Keys</u>	<u>Class</u>
Bag	No	No	Yes	None	any
IndexedCollection*	Yes	N.A.	N.A.	Integer	N.A.
FixedSizeCollection*	Yes	Yes	N.A.	Integer	N.A.
Array	Yes	Yes	Yes	Integer	any
ByteArray	Yes	Yes	Yes	Integer	SmallInteger
Interval	Internal	Yes	No	Integer	Number
String	Yes	Yes	Yes	Integer	Character
Symbol	Yes	Yes	Yes	Integer	Character
OrderedCollection	Yes	No	Yes	Integer	any
SortedCollection	Internal	No	Yes	Integer	any
Set	No	No	No	None	any
Dictionary	No	No	No	Lookup	any
IdentityDictionary	No	No	No	Lookup	any

Notes: * — abstract classes, there are no instances

Internal — ordered by the internal contents of the collection

N.A. — not applicable (determined by subclasses)

In the table, the only collections that have the same attribute values are the String Symbol pair and Dictionary IdentityDictionary pair. The difference between a String and a Symbol is that a Symbol is guaranteed to be unique while a String can have many copies. The difference between a Dictionary and an IdentityDictionary is that during the key lookup comparison, the former uses the = message while the latter uses ==.

Conversions

Because the various collection classes have different attributes, being able to convert from one kind of collection to another is useful. Smalltalk/V provides the following conversion protocol in class **Collection**.

<u>Methods</u>	<u>Comments</u>
asArray	Ordering is possibly arbitrary.
asBag	Duplicates are kept.
asSet	Duplicates are eliminated.
asOrderedCollection	Ordering is possibly arbitrary.
asSortedCollection	Each element is \leq its successor.
asSortedCollection: sortBlock	Ordering is specified by sortBlock .

Thus any collection can be converted into an **Array**, a **Bag**, a **Set**, an **OrderedCollection**, or a **SortedCollection**.

Instance Creation

Like other classes, message **new** can be used to create an instance of any collection. Message **new:** can be used to create a fixed-size collection with a specified size and a variable size collection with a specified initial allocation size.

Some collections may be expressed in literal form:

<u>Class</u>	<u>Instance in literal form</u>
String	'John Mary'
Symbol	# John
Array	#(\$J 'John' John (John 3))

A literal string is enclosed in a pair of quotes, a literal symbol is preceded by a number sign (#), and a literal array is enclosed in paired parentheses and preceded by a number sign. The **Array** example contains four elements: a character, a string, a symbol, and another array which has two elements — a symbol and a small integer. Notice that within a literal array, a symbol or another array element must not be prefixed with a number sign.

In addition, there is protocol in every collection class to create instances with one, two, three, or four elements which are not necessarily constants. For example,

Array with: 'Daughters of John'
with: #('Ann' 'Mary')

creates an array with two elements, a string and another array of two elements.

Common Protocol

Smalltalk/V provides common protocol to manipulate collections in a uniform way. These can be categorized as adding new elements, removing elements, testing the occurrences of elements, and enumerating elements. These are all described in Part 4: Encyclopedia of Classes under class **Collection**.

Suppose you have two global variables, **Customer** and **Supplier**, initialized as:

```
Customer := Bag with: #John.  
Supplier := #(John Peter).
```

Then you send adding, removing, and testing messages to **Customer**:

<u>Expression</u>	<u>Answer</u>	<u>Customer value if changed</u>
Customer add: #Bob	Bob	Bag(John Bob)
Customer addAll: Supplier	(John Peter)	Bag(John John Peter Bob)
Customer removeAll: Supplier	(John Peter)	Bag(John Bob)
Customer removeAll: Supplier	error	Bag(Bob)
Customer remove: #Bob	Bob	Bag()
Customer isEmpty	true	
Customer occurrencesOf: #John	0	
Customer includes: #John	false	
Customer addAll: #(John John)	(John John)	Bag(John John)
Customer addAll: Supplier	(John Peter)	Bag(John John John Peter)
Customer occurrencesOf: #John	3	

Enumerating messages allow you to process all the elements of a collection. Enumerating messages usually take a one-argument block as an argument and evaluate it with each element in the receiver collection. Assume **Customer** and **Supplier** have the same values as at the end of the last example.

```
| count |
count := 0.
Customer do: [ :aName | count := count + aName size].
^count
```

produces 17.

Customer select: [:aName | aName == #John]

produces Bag(John John John).

Customer reject: [:aName | aName == #John]

produces Bag(Peter).

Customer collect: [:aName | aName asArray]

produces Bag((\$J \$o \$h \$n) (\$J \$o \$h \$n) (\$J \$o \$h \$n) (\$P \$e \$t \$e \$r)).

Customer detect:

```
[ :aName | aName includes: $P]
```

produces Peter.

Customer detect:

```
[ :aName | aName = #Mary] ifNone: ['Not found']
```

produces 'Not found'.

Customer inject: 0 into:

```
[ :count :aName | count + aName size]
```

produces 17.

Class Bag

A **Bag** contains a collection of arbitrary objects. Duplicates are allowed and ordering is arbitrary. A **Bag** does not have external keys; therefore it cannot respond to the messages **at:** and **at:put:**. In addition to the common protocol, it has a message, **add:withOccurrences:** to add an element a specified number of times. **Bags** are hashed for efficient lookup.

As an example, here is an expression that computes the frequency of occurrence of words in a file.

```
| input frequency output word |
input:=File pathName: 'in.fil'.
output:=File pathName: 'out.fil'.
frequency:=Bag new.
[(word:=input nextWord) isNil]
    whileFalse: [frequency add: word asLowerCase].
frequency asSet asSortedCollection do:[:word|
    output
        nextPutAll: word;
        tab;
        nextPutAll: (frequency occurrencesOf: word) printString;
        cr].
output close.
```

Class Set

A Set is like a Bag except that it cannot have duplicate elements. Sets are hashed for efficient lookup.

As an example, here is an expression that computes a sorted list of words in a file.

```
| input words word |
input:= File pathName: 'in.fil'.
words= Set new.
[(word:= input nextWord) isNil]
    whileFalse: [words add: word asLowerCase].
^ words asSortedCollection.
```

Class Dictionary

Class Dictionary represents a set of objects with external lookup keys. Dictionaries are hashed for efficient lookup. A dictionary's elements are instances of class Association which contain a lookup key and its corresponding value. Because the key is only for lookup purposes, the messages includes:, do:, and other inherited enumeration messages are applied to the values rather than to the keys or to the associations themselves. Class Dictionary provides other messages to deal with keys and associations. Refer to Part 4 for all the messages implemented by class Dictionary.

Class IdentityDictionary

Class **IdentityDictionary** is similar to **Dictionary** except that it uses equivalence (`==`) instead of equality (`=`) during a key lookup. Its implementation also makes better storage utilization than a **Dictionary**. Because its key lookup matches object pointers instead of object contents, the only sensible classes for its keys (except for special situations) are **Symbol** and **SmallInteger**.

Class IndexedCollection

Class **IndexedCollection** represents collections with elements ordered externally by integer indices. It is an abstract class to contain common protocol for its subclasses and therefore should not have any instance of its own created.

Because of its well-defined ordering, all of its subclasses implement the equality (`=`) message in such a way that the answer is true if two **IndexedCollections** have the same class and size, and their corresponding elements answer true for the equality message.

Class FixedSizeCollection

Class **FixedSizeCollection** is a subclass of class **IndexedCollection**. It is an abstract class to provide common protocol for its subclasses: **Array**, **ByteArray**, **Interval**, **String**, and **Symbol**. These subclasses represent collections with a fixed range of integer indices as external keys. Because these subclasses have fixed sizes, they cannot respond to the `add:` message.

The instance creation message `new:` is subtly different when applied to a fixed size collection than to a variable one. The following message:

`(Array new: 5) size`

evaluates to 5, while

`(OrderedCollection new: 5) size`

evaluates to 0. When message `new:` is sent to class **Array**, the new instance is created with elements initialized to nil. When the message is sent to a variable size collection like **OrderedCollection**, the new instance is created with space allocated, but is logically empty.

The elements of an **Array** can be any objects. An element of a **ByteArray** must be a **SmallInteger** in the range of 0 to 255. The elements of a **String** or **Symbol** are characters. **Symbols** are guaranteed to be unique.

An **Interval** represents a finite arithmetic progression. Its elements can be any kind of number: integer, float, or fraction. Although **Interval** contains all the numbers within a specified range and with a specified increment between each number, it is represented concisely with only three instance variables: beginning, end, and increment. Its elements are regenerated upon access rather than stored in the instance. To create an instance, the two **Interval** class messages, **from:to:** and **from:to:by:**, are used. Class **Number** also provides some shorthand messages, **to:** and **to:by:**, to create new **Intervals**.

Class OrderedCollection

OrderedCollections are ordered by the sequence in which objects are added to and removed from them. They are like dynamic arrays, except that they can be expanded on both ends. To facilitate this feature, messages are provided to add, remove, and access both the beginning and end.

The **add:** message defined in class **Collection** is implemented to be like **addLast:**. Other messages enable you to access, add, or remove an object in the middle by specifying its preceding or succeeding object.

OrderedCollections can act as stacks or queues. Operations to a stack are typically "last-in, first-out." Following is a comparison of terminology:

<u>Typical Stack Vocabulary</u>	<u>OrderedCollection Message</u>
push newElement	addLast: newObject
pop	removeLast
top	last
empty	isEmpty

Operations to a queue are typically "first-in, first-out":

<u>Typical Queue Vocabulary</u>	<u>OrderedCollection Message</u>
add newElement	addLast: newObject
delete	removeFirst
front	first
empty	isEmpty

Queues grow on one end and shrink on the other. When space is exhausted on the growing end, an **OrderedCollection** always checks the shrinking end. If there is enough space, it shifts the entire collection towards the shrinking end to make room for growing at the other end. If there is not enough space, it will allocate a larger space and copy the original collection to the new space.

Class SortedCollection

SortedCollections are ordered according to a two-argument block called the *sort block*. The sort block is used to determine whether two elements are correctly sorted relative to each other. Because the position of each element is dictated by the sort block, messages such as `addLast:` are disallowed. Message `add: newObject`, however, will insert the `newObject` into the sorted position according to the sort block.

There are five ways to create a new instance:

```
SortedCollection new
SortedCollection new: 10
SortedCollection sortBlock: [ :a :b | a > b ]
anyCollection asSortedCollection
anyCollection asSortedCollection: [ :a :b | a > b ]
```

A sort block can be as complex as desired, but the last expression in the block must evaluate to either true or false. For example, the following sort block assumes that strings are being compared. It sorts the strings based on the number of unique vowels.

```
[ :a :b |
  (a asLowerCase select: [ :c | c isVowel ]) asSet size
  <=
  (b asLowerCase select: [ :c | c isVowel ]) asSet size]
```

When the sort block is not specified at creation time, the following default sort block is used:

```
[ :a :b | a <= b ]
```

The sort block can also be changed any time by sending message `sortBlock: newBlock` to a **SortedCollection** which automatically resorts the whole collection according to the `newBlock`.

Window Classes

To write an interactive application in **Smalltalk/V**, you need to understand **Smalltalk/V** window technology. In **Smalltalk/V**, a window typically involves three major kinds of classes (and their subclasses): the application classes (such as `ClassBrowser`) which synchronize panes, the `Pane` classes which display on the screen, and the `Dispatcher` classes which process keyboard and mouse inputs.

The application class is also referred to as the **model** class. It has to be written for each new application, although you can use an existing model as a template. The **Pane** and **Dispatcher** classes and their subclasses are complete building blocks in the system, and you rarely need to modify them.

The relationship among these classes is depicted in *Figure 14.1* using the **ClassBrowser** window as an example.

You can look at the **model** class as the skeleton of a window which organizes all the window panes and is responsible for the communication and synchronization among the panes.

The **Pane** class has two immediate subclasses: **TopPane** and **SubPane**. The main function of an instance of class **TopPane** is to coordinate all of the subpanes in the *model* window. Thus there is one and only one instance of class **TopPane** in each window. It holds the window label located at the top of the window.

The **SubPane** class has three subclasses: **GraphPane**, **ListPane** and **TextPane**. An instance of class **ListPane** provides a view of a list of strings. You can browse through the list (scrolling if necessary), and select the string that you wish. When you make a selection, the **model** instance will be notified and act accordingly.

An instance of class **TextPane** lets you view and edit the text that it contains. The text is usually represented as an instance of class **StringModel**. When you save a piece of modified text, the **model** instance is again notified to act according to your request.

Every pane has a unique dispatcher associated with it. The type of the pane determines the type of the associated dispatcher. An instance of a **Dispatcher** subclass serves as a messenger that collects input from the keyboard and mouse. It sends messages to its pane to take actions according to the input events.

Keep in mind that for an application with window panes that involve only classes **GraphPane**, **ListPane** and **TextPane**, you only need to know about the application model. If you want to define new kinds of panes, then you also need to define corresponding new **Dispatcher** subclasses, which means you need to learn about the entire trilogy. Or if you want to re-assign the meaning of a function key, then you need to know how to modify the existing **Dispatchers**. For these more advanced problems refer to the Smalltalk source code and **Part 4: Encyclopedia of Classes** for the relevant classes.

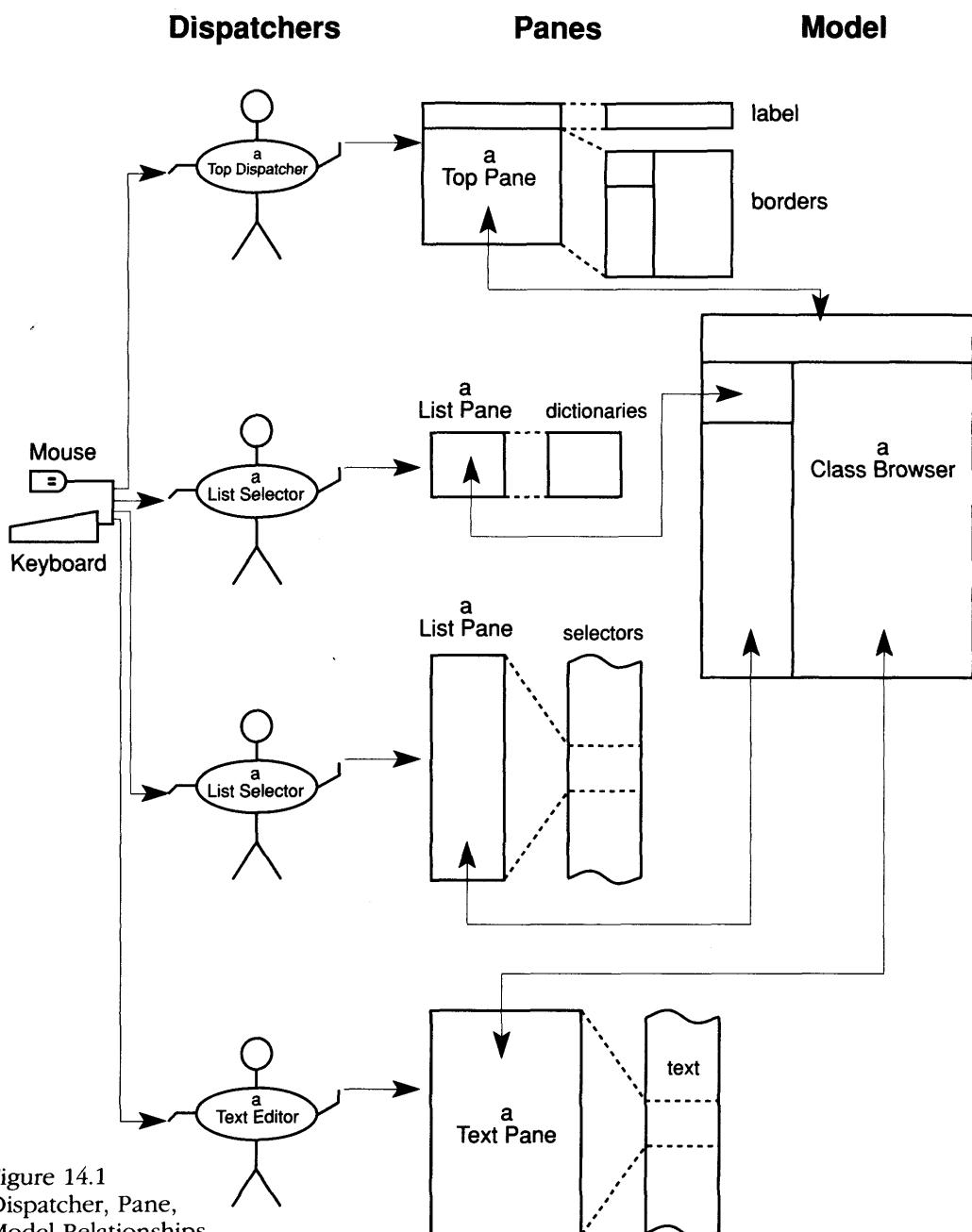


Figure 14.1
Dispatcher, Pane,
Model Relationships

Application Model

An application model has five major functions:

- Remember the Current State
- Create Panes
- Initialize Contents of Panes
- Carry Out Communication and Synchronization
- Define Menus for Panes

Remember the Current State

This is normally accomplished by assigning application states to instance variables in the **model** class. For example, class **ClassBrowser** has the following instance variables:

browsedClass

The class object you are currently browsing.

selectedDictionary

The current message dictionary (either class or instance) of the class you are browsing.

selectedMethod

The currently selected method within the currently selected message dictionary.

The contents of these variables are normally initialized during pane creation and changed from time to time by the **change** methods mentioned below.

Create Panes

The creation of panes is usually accomplished by sending the message **open** or **openOn:**, depending on whether an argument is needed, to a new instance of the **model** class. It initializes the following items:

- The label and minimum size of the window
- The name of each pane
- A menu creation message associated with each pane (optional)
- A framing block for calculating the area of each pane relative to the window
- A change message for each pane to send when the change in a pane has global effects
- The model of each pane (usually the application model itself but can be another model. This can also be changed dynamically.)

Following is the `openOn:` method defined in `ClassBrowser`:

```

openOn: aClass
    "Create a class browser window on aClass.
    Define the type, behavior and relative size
    of each pane and schedule the window."
| topPane twoLineHeight |
(aClass isKindOf: Class)
    ifFalse: [ ^ nil].
browsedClass := aClass.
topPane := TopPane new
    label: aClass name, ' | ClassBrowser';
    minimumSize: SysFontWidth * 20
        @ (SysFontHeight * 8);
    yourself.
twoLineHeight := ListFont height * 2 + 4.
topPane addSubpane:
    (ListPane new
        model: self;
        name: #dictionaries;
        change: #dictionary:;
        selection: 2;
        framingBlock: [box|
            box origin extent:
                box width // 5 @ twoLineHeight]).
selectedDictionary := browsedClass.
topPane addSubpane:
    (ListPane new
        model: self;
        name: #selectors;
        menu: #selectorMenu;
        change: #selector:;
        framingBlock: [:box|
            box origin + (0 @ twoLineHeight)
            extent: box width // 5 @
                (box height - twoLineHeight)]).
topPane addSubpane:
    (TextPane new
        model: self;
        name: #text;
        change: #accept:from:;
        framingBlock: [:box|
            box origin + ((box width // 5) @ 0)
            corner: box corner]).
topPane dispatcher open scheduleWindow

```

Invoking this method will create a window with three subpanes:

```
dictionaries (a ListPane)
selectors (a ListPane)
text (a TextPane)
```

In order for the window to work properly, the messages **dictionaries**, **dictionary:**, **selectors**, **selector:**, **selectorMenu**, **text**, and **accept:from:** must be defined as instance methods in the model.

Initialize Contents of Panes

The application model must provide for each subpane a method with the same name as the pane name which, when invoked, answers the data of the pane. For example, the **ClassBrowser** has three methods to initialize its three subpanes:

dictionaries

"Answer the array of dictionaries"

^ #(class instance)

selectors

"Answer a sorted list of selectors for the selected dictionary"

^ selectedDictionary selectors asSortedCollection

text

"Answer the source text for the selected method"

^ selectedDictionary sourceCodeAt: selectedMethod

Note that the first two methods answer an instance of a subclass of class **IndexedCollection** whose elements must be printable like Strings or Symbols. The third one answers a string (lines within the string are separated by line feeds). In case there is more data than a string can hold, the method **fileInFrom:** in **TextPane** can be used to initialize its data from an external file (refer to source code of the **file** method in the **DiskBrowser** class).

Carry Out Communication and Synchronization

When you make a selection or change the contents of pane data, the effect can be either *local* or *global*. Global effects the model or other panes. Anything else is local. For example, in the Class Browser, when you make a selection in the dictionaries pane, both the selectors pane and text pane need to be synchronized. Thus the effect is global. If you make editing changes in the Class Browser's text pane, the change is local because it does not effect other panes or the model. When you save these changes, however, the text needs to be compiled into the selected class and logged to the **change log** file. This can only be done by the model, so the effect of saving the text is global.

Specifying change messages in the `open` or `openOn:` method provides each pane with a message to send when these global effects occur. The argument of a change message is usually a piece of data passed from the pane to the model. For example, when you select a method in the selectors pane, the following method in `ClassBrowser` is invoked by the pane:

```
selector: aSymbol
    "Display the selected method in the text pane"
    selectedMethod := aSymbol.
    self changed: #text
```

where the first statement changes the application state by assigning `aSymbol` to the instance variable `selectedMethod`, and the second statement informs the `text` pane that the state has changed and it needs to update its contents.

When the global effect calls for one or more updates in another pane, the `changed:` or `changed:with:` method defined in class `Object` can be used to broadcast the effect to all subpanes of the model. In the previous example, selecting a method in the selectors pane displays the source of the method in the text pane. The `changed:` message is used to notify the text pane that the `selectedMethod` had been changed. The other message, `changed:with:`, in addition to notifying subpanes, also passes an object as the argument of the `with:` keyword to provide communication from the model to the panes.

When the subpane receives the update message (sent by method `changed:`) and its name matches the argument of the `changed:` keyword, the pane name is sent as a message to the model retrieving the new pane contents. To continue the previous example, after the text pane receives the update message, it updates its own data by sending the `text` message to the `ClassBrowser` to perform the `text` method which in turn answers a string of the new text pane contents. This concludes the update, and the `Dispatcher` regains control and waits for the next keyboard or mouse event.

Define Menus for Panes

If the `menu:` message is sent during the creation of a pane, a method with the same name as the `message` argument must be defined in the model. This method answers an instance of class `Menu` which contains the desired menu items for the pane. In the `openOn:` method of the `ClassBrowser`, the message `menu: #selectorMenu` is sent to the selectors pane. Thus a corresponding method is defined in the `ClassBrowser`:

```
selectorMenu
    "Answer the selector pane menu"
    ^Menu
        labels: 'remove\senders\implementors' withCrs
        lines: #()
        selectors: #(removeSelector senders implementors)
```

The string argument to **labels:** contains the items to be shown in the menu. Message **withCrs** replaces backslashes (\) with line feeds in its receiver string. The argument to **selectors:** is an array of messages to send when you select the corresponding item in the menu. The methods carrying out these messages can be optionally defined in the model. If you do not define them, the ones in the **Dispatcher** class (associated with the pane) are used as defaults. If they are not defined in either class, an error results. The **ClassBrowser** defines all three methods needed in the menu:

```

removeSelector
    "Remove the selected method"
    selectedMethod isNil
        ifTrue: [ ^ nil ].
    selectedDictionary
        removeSelector: selectedMethod.
    Smalltalk logEvaluate:
        selectedDictionary name,
        'removeSelector: #' ,
        selectedMethod.
    selectedMethod := nil.
    self
        changed: #selectors with: #restore;
        changed: #text

senders
    "Popup a window with the senders of the selectedMethod"
    selectedMethod == nil
        ifFalse: [ Smalltalk sendersOf: selectedMethod ]

implementors
    "Popup a window with the implementors of the selectedMethod"
    selectedMethod == nil
        ifFalse: [
            Smalltalk implementorsOf: selectedMethod ]

```

The **removeSelector** method provides another example of changing the current state and using message **changed:** to inform the selectors pane to update.

Pane

An instance of class **TopPane** is responsible for all the functions pertaining to the whole window:

- Display the window frame and invoke each **SubPane** to display its pane contents.
- Save, display, and highlight the window label.
- Activate the window and all subpanes.
- Answer whether the window contains a certain point.
- Close the **TopPane** and invoke each **SubPane** to close itself.

Class **SubPane** and its three subclasses, **GraphPane**, **ListPane** and **TextPane** are responsible for functions that are specific to subpanes:

- Display the pane frame.
- Activate itself.
- Answer whether the pane contains a certain point.
- Display a portion of its data in the pane.
- Scroll data in four directions.
- Make a selection on a piece of its data.
- Close itself.

In addition, a **TextPane** provides the capabilities to cut, paste, copy, and execute portions of its data.

Dispatcher

The main function of an instance of class **Dispatcher** and its subclasses is to interpret the input from the keyboard or mouse and send an appropriate message to the corresponding pane. It also has the following functions:

- Activate or de-activate the corresponding pane.
- Return the cursor to the top-left corner of its pane.
- Open or close the window.
- Cycle windows or panes in the window.

A **TopDispatcher** has the following additional functions:

- Provide methods to execute items in the window menu.
- Set or change the content of the window label bar.

Other **Dispatchers** have the following additional functions:

- Tell the pane to scroll its data by a specified amount.
- Tell the pane to handle selections.

- Provide methods to execute items in the pane menu.

Prompter

Class **Prompter** gives an application writer a simple mechanism to pose a question and solicit an answer. A **Prompter** is a window with its label showing the intended question and a single text pane for editing the answer. It is a window application itself, but is often used by other window applications as a building block. For example, when you create a file using the Disk Browser, the first thing you see is a **Prompter** asking you to respond with the file name.

To open a **Prompter**, you can send one of the following two messages to class **Prompter**:

Prompter prompt: question default: answer
Prompter prompt: question defaultExpression: answer

where both **question** and **answer** are strings. After the **Prompter** window is opened, the **answer** string will be shown in its text pane as a default. The first message returns a string as answered by the application user, while the second message returns an object resulting from evaluating the answer. For instance,

Prompter prompt: 'Give me a string please'
default: '2 + 3'

returns '2 + 3', and

Prompter prompt: 'Give me an expression please'
defaultExpression: '2 + 3'

returns 5 after the default answer is accepted. If you cancel the **Prompter**, an answer of nil will be returned by both messages.

Notice that when a **Prompter** is accepted or canceled, the program flow control is given back to the caller of the **Prompter**. When most other kinds of windows are closed, control is given to the **Scheduler** (described below) to cause another window to become active.

Dispatch Manager

A **DispatchManager** schedules all the windows under its control. Normally only one such instance exists in the system which is contained in the global variable **Scheduler**. The **Scheduler** maintains an ordered collection of **TopDispatchers** and schedules windows by sending messages to these **TopDispatchers**. It performs the following functions:

- Add and remove dispatchers (thus windows).

- Answer the **TopDispatcher** associated with the active window.
- Display all the windows.
- Cycle the ordering of windows.
- Search for the window containing the cursor and make it the active window.
- Re-initialize the system by removing all windows and then drawing the System Transcript window.

Applications with Multiple Windows

One way to coordinate windows within a multi-windowed application is to have one application model for all the windows in the application. You make each subpane of these windows a dependent of the model. When a change is made to a subpane which will have global effects, one of the **change** methods in the model will be invoked. It will then decide what other subpanes will be affected and updated.

In **Smalltalk**, subpanes in one window can have non-unique names. Subpanes with the same name are updated simultaneously in one broadcasting of changes. Since this is also true for the one model multi-window approach, take care in giving the same name to subpanes in different windows.

Another way to coordinate windows is to leave the current scheme of one model per window as it is. In this case, one super model is created with all other models as its dependents. In the super model, one **change** message is defined for each dependent. When the lower-level model receives a **change** message, it updates its own panes as in the single window application. In addition, the lower-level model sends an appropriate **change** message to the super model which then broadcasts the change to other dependent models. Each lower-level model should implement an **update** method to accept the broadcasting from the super model.

A window can also be associated with the super model which can, for example, allow you to select different functions in the application. Each selection triggers a sequence of windows being activated. This approach provides more modularity. For instance, subpane names are related to a particular lower-level model, so you do not have to be concerned about name collisions among windows. This gives you an opportunity to use single-window applications as building blocks. The disadvantage is that more levels of message sends will be generated.

When an application wants to update a subpane which is outside of the current active window, it should make the subpane's window the active window first. If it does not, the portions of the subpane obscured by other windows will be overwritten.

Graphic Classes

Smalltalk/V graphics are generated by bitmapped operations. In fact, Smalltalk/V uses bitmapped graphics to generate the entire user interface.

A **Bitmap** is simply a linear array of bits. Each bit has a value of 1 or 0, with 1 representing white and 0 black. Since a monochrome display screen is a two dimensional plane of pixels, instances of class **Form** provide a two dimensional view of the bitmap to represent the monochrome screen. A Form has a **bitmap**, a **width** and a **height**, which allows you to manipulate the bitmap as if it were a two dimensional array of bits. The **DisplayScreen** represents a monochrome screen which is a special kind of Form, and hence a subclass of class **Form**.

A **ColorScreen** or a **ColorForm** (both are subclasses of **Form**) consists of an array of bitmaps. This additional dimension allows colors other than black and white to be recorded. All bits at the same location of each bitmap collectively represent the color of the pixel at that location. For example, if the bit at location $0@0$ in the first bitmap is 1, and the bits at the same location in the second, third, and fourth bitmap are 1, 0, and 0, then the color code for the top left pixel is the binary number 0011 (the bit in the first bitmap is least significant) or 3 in decimal. The EGA and VGA graphics controllers allow a maximum of four bitmaps, thus a maximum of 16 colors are available to be displayed at any one time.

Almost all bitmapped operations involve moving bits from one place to another. Naturally, you might think that you would implement this by simply sending messages to a **Form**. Due to the complexity of these operations, however, another class, called **BitBlt**, is created to handle all bitmapped operations.

Bitmapped operations address a bit or an area of bits in a Form. A **Point** addresses an individual bit, while a **Rectangle** refers to a block of bits.

Bitmapped graphics, then, is centered around four classes and their subclasses: **Point**, **Rectangle**, **Form**, and **BitBlt**. The rest of the chapter describes each of these classes in detail.

Point

A **Point** addresses a bit within the bitmap of a two dimensional **Form**. It has two instance variables: **x**, representing the column (horizontal) coordinate, and **y**, representing the row (vertical) coordinate. The value of **x** increases to the right and **y** to the bottom.

The most efficient way to create a **Point** is by sending the **@** message to an **Integer**. For example, the top left corner of a **Form** can be addressed by the **Point**:

0 @ 0

where the first integer (receiver) is the x coordinate and the second integer (argument) the y coordinate. The **x:** and **y:** messages alter the coordinates of a Point, while the **x** and **y** messages retrieve these coordinates. A Point can also be compared with another Point. Following are some examples:

<u>Expression</u>	<u>Result</u>
(1 @ 100) x	1
(1 @ 100) y	100
(1 @ 100) x: 50	50 @ 100
(1 @ 100) y: 50	1 @ 50
(-2 @ 10) < (-1 @ 11)	true
(-2 @ 10) < (-1 @ 10)	false
(-2 @ 10) > (-3 @ 11)	false
(-2 @ 10) max: (-3 @ 11)	-2 @ 11
(-2 @ 10) min: (-3 @ 11)	-3 @ 10
1 @ 2 between: 0 @ 2 and: 2 @ 2	true

Arithmetic can be performed on a Point with either a Point or a Number (as a scalar) argument. The message **transpose** creates a new Point by swapping the two coordinates of the receiver Point, while **dotProduct** gives the sum of the x product and y product of two Points:

<u>Expression</u>	<u>Result</u>
(1 @ 10) + (2 @ 12)	3 @ 22
(3 @ 22) - 10	-7 @ 12
(1 @ 10) * (3 @ 2)	3 @ 20
(1 @ 10) // 2	0 @ 5
(-2 @ -3) abs	2 @ 3
(2 @ 3) negated	-2 @ -3
(2 @ 4) dotProduct: (5 @ 6)	34
(2 @ 4) transpose	4 @ 2

Rectangle

A Rectangle references a rectangular block of bits contained in a Form. It is represented by two Points: the top left Point, called **origin**, and the bottom right Point, called **corner**. Its width and height can then be calculated by:

```
width := corner x - origin x
height := corner y - origin y
```

which represent the number of columns and the number of rows of bits contained in the Rectangle. The Point represented by width @ height is called the *extent* of the Rectangle. A simpler way to calculate the extent is:

extent := corner - origin.

A Rectangle is usually created by sending the **corner:** or **extent:** message to a Point. For example, the following two expressions create two Rectangles covering the same area:

1 @ 1 corner: 100 @ 100
1 @ 1 extent: 99 @ 99

To illustrate the Rectangle instance messages, consider these rectangles, **Box1** and **Box2**:

Box1 := 20 @ 0 corner: 150 @ 100.
Box2 := 70 @ 80 corner: 170 @ 120.

<u>Expression</u>	<u>Result</u>
Box1 top	0
Box1 bottom	100
Box2 left	70
Box2 right	170
Box1 center	85 @ 50
Box1 width	130
Box1 height	100
Box1 origin	20 @ 0
Box1 corner	150 @ 100
Box1 containsPoint: 50 @ 50	true
Box1 expandBy: 10	10 @ -10 corner: 160 @ 110
Box2 insetBy: 10	80 @ 90 corner: 160 @ 110
Box1 intersects: Box2	true
Box1 intersect: Box2	70 @ 80 corner: 150 @ 100
Box1 merge: Box2	20 @ 0 corner: 170 @ 120
Box1 translateBy: 10 @ 10	30 @ 10 corner: 160 @ 110
Box1 moveBy: 10 @ 10	30 @ 10 corner: 160 @ 110
Box2 moveTo: Box1 origin	30 @ 10 corner: 130 @ 50

Notice that the last two expressions modify **Box1** and **Box2** themselves, while others create new rectangles.

Form

Class **Form** is a subclass of **DisplayMedium**, which in turn is a subclass of **DisplayObject**. **DisplayObject** and **DisplayMedium** are abstract classes; they do not hold any data. Their purpose is to group related methods together to be inherited by their subclasses. (Part 4 lists their protocols.)

As we said earlier, the main purpose of a **Form** is to provide a two dimensional view for a bitmap. It therefore has three instance variables: **bits** (which contains the bitmap), **width** (which contains the number of pixels horizontally), and **height** (containing the number of pixels vertically).

A **Form** also has three additional instance variables:

- **offset** is the Point on the **DisplayScreen** from which this **Form** was originally copied by sending message **fromDisplay:** to class **Form**.
- **byteWidth** is the number of bytes (8 bits) horizontally. For efficiency reasons, each row of a **Form** is allocated in an integral number of words (16 bits). Therefore, if its **width** is not an integral of the word size, the remainder bits in the last word of each row (the right hand side of the **Form**) are not used. This, however, is transparent to the user.
- **deviceType** denotes the type of device to which this **Form** belongs. When an instance of class **Form** is created, its **deviceType** is automatically set to 0, indicating that the **Form** resides in regular memory. When an instance of class **DisplayScreen** (a subclass of **Form**) is created, it is set to 1, indicating that the **Form** serves as the buffer for the display screen, whose address and size are dictated by the graphics adapter and graphics mode being used. The values of **deviceType** for instances of class **ColorScreen**, **ColorForm**, and **BiColorForm** are 1, 3, and 2, respectively.

A global variable, called **Display**, contains an instance of either **ColorScreen** or **DisplayScreen**. Every time you change the graphics mode, this variable is reinitialized to the exact size of the screen. It can be used as any other **Form**; when pixels are moved to **Display**, however, they are automatically shown on the screen. In other words, the contents of **Display** reflect your monitor's display screen.

The following messages can be sent to class **Form**, **BiColorForm**, or **ColorForm** to create new instances:

fromDisplay: aRectangle

Answer a new **Form** whose extent equals **aRectangle**'s extent and whose content is copied from **aRectangle** area of the display screen.

width: wInteger height: hInteger

Answer a white **Form** whose width is **wInteger** and height is **hInteger**.

You can also use the message `Form new` to create an instance of `Form`, and then use one of the following messages to initialize its variables:

extent: aPoint

Change the receiver width and height to the coordinates of `aPoint`.

width: w height: h

Change the receiver width to `w` and height to `h`, and allocate its bitmap with the appropriate size.

width: w height: h initialByte: aByte

Change the receiver width to `w` and height to `h`, and initialize every byte in the bitmap to `aByte`.

Other useful messages are `copy:from:to:rule:`, which copies the contents of one `Form` to another `Form`; `displayAt:` and `displayOn:at:clipping Box:rule:mask:`, which display the contents of a `Form` on the screen; and `outputToPrinter`, which shows the contents of a `Form` on a printer. Refer to the description of `Form` in Part 4, **The Encyclopedia of Classes**, for the definition of these messages.

BitBlt

Class `BitBlt` ("bit block transfer") describes all the parameters in a basic bitmapped operation. This basic bitmapped operation is to move a block of bits from one place to another. More complicated operations, such as drawing a line, involve a sequence of such basic moves. This basic operation can sometimes become rather complicated when all the parameters are involved.

This basic operation works as follows. You first define a source rectangle on the source form. The bits from the source rectangle are combined with the bits from the mask form with a logical AND operation. The resulting rectangle of bits are combined into the destination `Form` rectangle with a specified combination rule and clipping rectangle.

The following figure shows how the parameters of a `BitBlt` determine which bits are involved in the bit transfer:

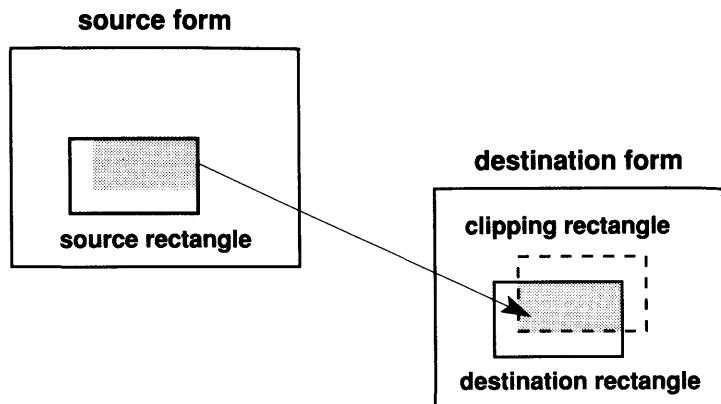


Figure 14.2
BitBlt Clipping

And the following figure shows how the resultant bits are produced by combining the source bits, mask bits, and destination bits with a combination rule:

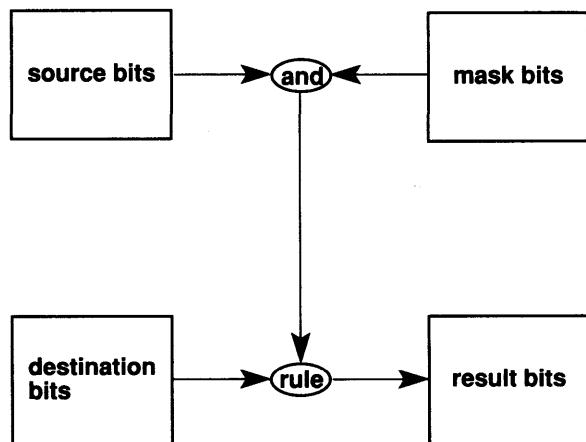


Figure 14.3
BitBlt Operation

For example, suppose

source bits = 11110000
mask bits = 10101010
dest bits = 00001111
rule = logical OR

the resultant bits would be 10101111. These resultant bits are then stored back to the destination Form.

A mask form has a fixed width and height of 16. If its extent is smaller than the source rectangle extent, its bits are repeated both horizontally and vertically up to the extent of the source form rectangle so that they can be ANDed with each bit in the source rectangle. The effect of this is to provide a halftone texture for the 1 bits in the source rectangle. The mask Form, therefore, is also called *halftone* Form. The following expressions obtain a prebuilt mask Form whose halftone is the same as the message name:

- Form white**
- Form black**
- Form gray**
- Form darkGray**
- Form lightGray**

When the destination is a **ColorForm** or **ColorScreen** and the source is a **Form** or **BiColorForm**, a **BiColorForm** mask can be used to obtain colors other than black and white. This is not a rare case. For example, both **Pen** and **CharacterScanner** instances are in this category. **BiColorForm** has two additional instance variables: **foreColor** and **backColor**. When a **BiColorForm** is used as a mask form, its **foreColor** specifies the color for 1 bits and the **backColor** the color of 0 bits after source bits are ANDed with the mask form bits. To obtain a **BiColorForm** mask, the following messages can be used:

- BiColorForm color: 1.**
- BiColorForm gray foreColor: 2 backColor: 4.**

The first message returns a 16 by 16 **BiColorForm** whose bitmap consists of all 1 bits and whose foreground color is 1 (blue) and background color is 0. The second message returns a 16 by 16 **BiColorForm** whose bitmap consists of alternating 0 and 1 bits and whose foreground color is 2 and background color is 4.

When the ANDed source bits are combined into the destination form rectangle, two additional parameters must be specified. One is the *combination rule*, which indicates how the source bits are to be combined with the destination bits. The other is the *clipping rectangle*, which limits the affected area on the destination form. To access the combination rules supported by Smalltalk/V, send a message to class **Form**:

Form over	destination becomes source
Form orRule	source OR into destination
Form andRule	source AND into destination
Form under	source AND into destination
Form erase	if source is 1 then destination becomes 0
Form reverse	source XOR into destination
Form orThru	first erase without specifying mask Form, then OR with mask Form specified

The final affected area on the destination Form can be simulated as follows:

```

AffectedRect := sourceRect intersect:
    sourceForm boundingBox.
AffectedRect moveBy: destRect origin - sourceRect origin.
AffectedRect :=
    ((AffectedRect intersect: destRect)
        intersect: destForm boundingBox)
        intersect: clippingRect

```

The instance variables of BitBlt are:

```

sourceForm destForm halftone rule width height
sourceX sourceY destX destY clipX clipY clipWidth clipHeight

```

Refer to BitBlt in Part 4, The Encyclopedia of Classes for a description of all instance variables.

In terms of instance variables, the source rectangle is defined as:

```
sourceX @ sourceY extent: width @ height
```

The destination rectangle is defined as:

```
destX @ destY extent: width @ height
```

And the clipping rectangle is defined as:

```
clipX @ clipY extent: clipWidth @ clipHeight
```

The **sourceForm** must be either nil or an instance of **Form** or its subclasses. The **halftone** must be either nil or an instance of **Form** or **BiColorForm** with width and height equal to 16. The **destForm** must be an instance of **Form** or its subclasses. All other instance variables must be a **SmallInteger**. When **sourceForm** or **halftone** is nil, it implies an infinitely large form with contents of all 1 bits.

Note that the source form and destination form can be the same form. In this case, the only complication is that the source rectangle may overlay the destination rectangle. When such overlay occurs, the operation is carried out as if the source bits were first copied to an intermediate Form, and then copied to the destination rectangle. The actual implementation still amounts to a single move and is as efficient as moving between different Forms.

To create an instance of class BitBlt, use the class message:

```
destForm:sourceForm:
```

with only destination and source forms specified. Other instance variables are supplied with the following default values:

```

sourceX := sourceY := 0.
destX := destY := 0.
clipX := clipY := 0.
width := destForm width.
height := destForm height.
clipWidth := destForm width.
clipHeight := destForm height.
rule := Form over

```

If you want to specify all parameters, use **BitBlt new** to create an instance, and then send the following message to the new instance to set the instance variables:

```

destForm:
  sourceForm:
  halftone:
  combinationRule:
  destOrigin:
  sourceOrigin:
  extent:
  clipRect:

```

Class **BitBlt** supplies numerous messages to change a small number of parameters. Refer to **BitBlt** in Part 4, **The Encyclopedia of Classes** for a list of all messages.

After the parameters have been correctly set up, two messages in class **BitBlt** actually move the bits:

copyBits

does exactly what it says: moves bits from one place to another.

drawLoopX: xDelta Y: yDelta

draws a line from the destination origin to:

(destX @ destY) + (xDelta @ yDelta).

The line is drawn by performing a sequence of **copyBits**, starting from the destination origin up to the other end of the line. For each **copyBits**, the destination origin is moved by one pixel horizontally and/or vertically towards the other end.

CharacterScanner

CharacterScanner is a subclass of **BitBlit**. Its main function is to convert a String of ASCII characters to displayable bitmapped shapes. It therefore adds an instance variable, **curFont**, which contains the current Font being used for conversion. A Font provides a Form containing all the bitmap images of characters. It also has an index table which gives the source origin of each character within the Form. (Class **Font** is discussed in Chapter 15.) To display a Character, you use its ASCII value to obtain the source origin from the index table and then move the bitmap image to the destination form.

To create a **CharacterScanner**, use either the expression:

```
CharacterScanner new
    initialize: clipRect
    font: aFont
```

or

```
CharacterScanner new
    initialize: clipRect
    font: aFont
    dest: aForm
```

The former method uses **Display** as the destination form, while the latter specifies **aForm** as its destination. Both specify **aFont** as the current font and **clipRect** as the clipping rectangle.

CharacterScanner has an instance variable, **frame**, which is initialized by both of the above messages to contain **clipRect**. This allows you to temporarily change the clipping rectangle and later restore it back to **frame**.

To specify colors, **CharacterScanner** has another two variables: **foreColor** specifies the color of the character, and **backColor** specifies the color of the background. They default to black and white, respectively, with both of the above messages.

To display a string, send the following message to a **CharacterScanner**:

```
display: aString
    at: startPoint
```

where **startPoint** is the top left corner, relative to the origin of the frame, of the displayed string. To display a portion of a string, send the message:

```
display: aString
    from: startChar
    to: endChar
    at: startPoint
```

where **startChar** and **endChar** bracket the portion of the string to be displayed. Another message,

```
display: aString
from: startChar
at: startPoint
```

displays **aString** from **startChar** up to its last character at **startPoint** in the frame. In addition, it blanks out the rest of the line after the last character. This is often used when replacing a portion of the line with another string. Blanking out the rest of the line is needed since the length of the line may change after the replacement.

If you want to display a string without bothering its background, use the message:

```
show: aString from: start at: aPoint
```

which displays **aString** from **start** character up to its last character at **aPoint** in the frame without changing the background.

Other useful messages are:

```
displayForm: aForm at: aPoint rule: aRule
Display aForm at aPoint in the frame using aRule.
```

```
gray: aRect
Color aRect in frame with gray tone.
```

```
reverse: aRect
Reverse the color of aRect in frame.
```

```
setFont: aFont
Change the font to aFont.
```

```
setForeColor: fColor backColor: bColor
Set the foreground color to fColor and background color to bColor.
```

Pen

Class **Pen** provides a graphics interface similar to turtle graphics. It adds the following parameters to **BitBlt**:

- **location** consists of **destX** and **destY**, which represent the integer coordinates of a Pen's current location, and **fractionX** and **fractionY**, which represent the hundredth fractional part of the coordinates.
- **direction** is an instance variable containing a number between 0 and 359 degrees, which specifies where the Pen is heading. 0 makes the Pen go east, 90 makes it go south, and 270 north.

- **downState** is an instance variable containing true or false, indicating whether the Pen draws while it moves.

The source form of a Pen represents its nib. Thus, to draw a thicker line, simply expand the extent of the source form. The mask form provides the color of the Pen. The destination form is the form on which the Pen is going to draw. The clipping rectangle limits the area on which the drawing can have any effect.

The expression:

Pen new

creates a Pen with a nib of size 1 by 1, black color, **Display** as its destination form, the entire **Display** area as its clipping rectangle, the center of the **Display** as its **location**, a direction of north, and its **downState** equal to true. The expression:

Pen new: aForm

creates a similar Pen except, that it draws on **aForm** rather than **Display**.

The following messages (in addition to the ones provided in **BitBlt**) can be used to modify a Pen's parameters:

black

Change the Pen color to black.

gray

Change the Pen color to gray.

white

Change the Pen color to white.

changeNib: aForm

Change the source Form (nib) to **aForm**.

defaultNib: shape

Change the size of the nib to **shape** which can be either a number or a point.

direction: aNumber

Set the direction to **aNumber** degrees.

turn: anInteger

Turn the receiver Pen **anInteger** number of degrees which can be either positive or negative.

down

Set down the pen.

up

Lift up the pen.

home

Center the pen on the destination form.

north

Set the direction to 270 degrees.

place: aPoint

Position the receiver Pen at aPoint.

The following messages move the Pen by a certain pattern. If the Pen is down, the pattern will be drawn:

grid: scale

Draw a grid within the clipping rectangle where **scale** is the space between lines.

bounce: increment

If the Pen touches the clipping rectangle after going for **increment**, change its direction so that it looks like bouncing off the wall.

ellipse: r aspect: asp

Draw an ellipse with pen position as its center, **r** as half of the width, **asp** as the ratio of ellipse height to width. The height will be adjusted by **Aspect**.

fillAt: aPoint

Color all pixels that are connected to aPoint and have the same color as that of aPoint by the pattern contained in mask form.

go: distance

Move the receiver Pen for length **distance** in the current direction. The y-axis will be adjusted by **Aspect**.

goto: aPoint

Move the receiver Pen to aPoint.

polygon: length sides: n

Draw a polygon with **n** sides, each **length** long.

Notice that the **ellipse:aspect:** and **go:** messages both adjust their y-axis by the ratio contained in the global variable **Aspect**. With this adjustment, you can draw a real circle or a real square. To find out the correct **Aspect** ratio for your screen, first display a Form with the same width and height, measure with a ruler the width and height of the displayed rectangle, and evaluate:

Aspect := measuredWidth / measuredHeight.

Class **Pen** has a subclass called **Commander**. An instance of **Commander** contains an Array of **Pens**. Many messages implemented by **Pen** are reimplemented in **Commander** so that every **Pen** in the **Commander** receives the same message. This creates the illusion that all pens are moving at the same time. To create a **Commander**, evaluate the following expression:

Commander new: numberOfPens.

A **Commander** has two unique messages:

fanOut

Turn each **Pen**'s direction by an increment of $360 / (\text{number of Pens})$.

lineUpFrom: startPoint to: endPoint

Place all pens on a line specified by **startPoint** and **endPoint** with pens equally spaced between the two points.

Animation

Class **Pen** also has another subclass, called **Animation**. Each instance of **Animation** contains an object kept as a **Pen** with a name, a color, and an Array of pictures. When an object moves, the **Animation** erases its old image before displaying it at the new location. To create an **Animation**, evaluate:

Animation new initialize: aRectangle

where **aRectangle** is the clipping rectangle on Display. To add an object to an **Animation**, send the message:

add: formArray name: aName color: aSymbol

The **formArray** is an Array of **Forms** simulating the continuous movement of the object. The argument **aName** is normally a **String** or a **Symbol** to identify the object. The argument **aSymbol** is the selector for a unary message which can be sent to class **Form** to return a mask form (such as **#black**, **#white**, etc.).

Animation has the following messages to change the behavior of its objects:

tell: name bounce: increment

Tell the object with the **name** to bounce for **increment** distance.

tell: name direction: anInteger

Tell the object with the **name** to change its direction to **anInteger**.

tell: name go: distance

Tell the object with the **name** to go for **distance**.

tell: name place: aPoint

Tell the object with the **name** to be placed at **aPoint**.

tell: name turn: anInteger

Tell the object with the **name** to turn by **anInteger** degrees.

speed: anInteger

Change the distance between the consecutive copies to **anInteger**. The larger the distance, the faster the object moves.

shiftRate: anInteger

Specify how many times the current picture will be copied before shifting to the next one. By increasing the value of **anInteger**, the motion within the object will appear slower even though it is traveling the same distance.

Integrating Color and Monochrome Displays

Many applications expect to run on both monochrome and color displays with one Smalltalk/V image. Following are some guidelines for developing such applications.

When you want to create a Form that contains a portion of the display, use the message:

Display compatibleForm

to obtain the class of the new Form. It returns either class **Form** or class **ColorForm** depending on whether the class of **Display** is **DisplayScreen** or **ColorScreen**, respectively. For example:

Display compatibleForm fromDisplay: (0 @ 0 extent: 100 @ 100)

creates a **Form** or **ColorForm** containing the top left 100 by 100 area of the screen without worrying about whether the screen has color or not.

When you want to create a mask that is compatible with the destination, use the message **compatibleMask**. It returns class **Form** if the receiver of the message is a **Form**, **BiColorForm**, or **DisplayScreen**; or returns class **BiColorForm** if the receiver is a **ColorForm** or **ColorScreen**. This message is often used in conjunction with the message **color:** to obtain the desired color. For example:

Display compatibleMask color: 1

returns either a **Form** with all 0 bits if **Display** is a **DisplayScreen**, or a **BiColorForm** with all 1 bits and a foreground color 1 if **Display** is a **ColorScreen**. The way the **color:** message works is as follows. If the receiver is class **BiColorForm**, then the **color:** message returns a 16 by 16 **BiColorForm** with all 1 bits and its foreground color set to the argument and background color set to 0. If the receiver is class **Form**, the message returns a 16 by 16 **Form** with all bits set to 0 if the argument of the message is between

0 and 7, or set to 1 if the argument is between 8 and 15. In other words, for a Form mask, the **color:** message maps the first eight colors into 0 bits and the remaining 8 colors into 1 bits. Because of this mapping, when you use a **Pen** or a **CharacterScanner**, you had better choose the foreground colors from one set of eight colors and background colors from the other set of eight colors. Otherwise, when you run on a monochrome display, you will end up drawing white on white, or black on black, making the result invisible.

Multiprocessing Classes

Object-oriented computing in Smalltalk involves communicating objects which send messages to each other to perform useful work. Although this suggests parallel computation, it actually is not. An object always waits to receive a response after sending a message. The situation corresponds to that of using a procedural language, where at any point in time there is a stack of incomplete procedure calls, and there is a single procedure which is active. In Smalltalk, there is a stack of incomplete message sends, and there is a single method which is active.

Smalltalk/V provides parallel processing by defining multiple stacks of incomplete message sends, where each stack is represented by a separate object of class **Process**. Since there is a single processor, the parallelism is simulated. At any time, only a single process is executing. However there may be many processes ready to execute and there are well defined conditions under which **Smalltalk/V** switches to a new current process. **Semaphore** objects are provided for synchronization among processes.

A new process is created by sending the message **fork** to a block. For example:

```
[Transcript show: 'Hello'; cr] fork.  
Transcript show: 'Goodbye'; cr
```

This example creates a separate process to execute the code within the block and continues execution of the current process in the code following the block. The result displayed in the System Transcript is:

```
Hello  
Goodbye
```

The output shows that the new process is initiated before the current process is continued, although both processes operate at the same priority. Processes can be given different priorities by sending the **forkAt:** message to a block. For example:

```
[[Transcript show: 'world!'; cr] forkAt: 2.  
Transcript show: 'Hello '] forkAt: 3
```

The example above creates two new processes, one at priority two and the other at priority three. Since higher priority processes are scheduled first, the output on the System Transcript is:

Hello world!

Multiprocessing is especially useful in discrete event simulation because it allows each simulation object to carry out its behavior as a separate process, using semaphores to synchronize processes. Multiprocessing is implemented in **Smalltalk/V** by classes **Process**, **ProcessScheduler** and **Semaphore**.

For some interesting examples of multiprocessing applications in Smalltalk, see the following publications:

A Little Smalltalk, Timothy Budd, Addison-Wesley 1987, page 116, "The Dining Philosophers Problem".

Smalltalk-80, the Language and its Implementation, Adele Goldberg and David Robson, Addison-Wesley 1983, page 262, "Resource Sharing".

Semaphore

Semaphore is a subclass of **Object**. A **Semaphore** is an object used to synchronize multiple processes. A process waits for an event to occur by sending the message **wait** to a semaphore. A process signals that an event has occurred by sending the message **signal** to a semaphore.

A semaphore has two instance variables:

signalCount

Contains an **Integer** representing the number of **signal** messages minus the number of **wait** messages sent to the semaphore during its entire lifetime.

waitingProcesses

Contains an **OrderedCollection** of processes that have sent the message **wait** to the semaphore without a corresponding **signal** message. New waiting processes are added at the end of the collection.

An example of the use of semaphores is the following:

```
| s |
s := Semaphore new.
[Transcript show: '1'] fork.
[Transcript show: '2'.s wait. Transcript show: '3']
    forkAt: 3.
[Transcript show: '4'.s signal. Transcript show: '5'; cr]
    forkAt: 2
```

This example creates three new processes. The output displayed on the System Transcript is:

1 2 4 3 5

This output is created as follows. The `fork` message creates a process which shows '1'. The `forkAt: 3` message creates a process which shows '2' and then is blocked waiting on the semaphore. The `forkAt: 2` message creates a process which shows '4' and signals the semaphore. This allows the higher priority process to resume, show '3' and terminate. Then the process at priority 2 resumes, shows '5' and terminates. The initiating process, the user interface, is running concurrently with these processes.

Process

Process is a subclass of **OrderedCollection**. A process is a sequence of computations in Smalltalk carried out by objects sending messages to other objects and waiting for the results. An object of class **Process** describes such a computation sequence. A process has a **name** and a **priority**. A process exists in one of several states. Figure 14.4 shows the state transitions a process can make.

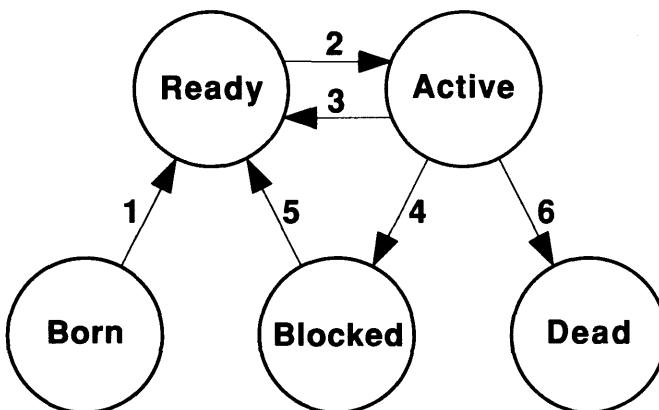


Figure 14.4
Process State Transitions

The process state transitions occur for the following reasons:

- A new process is created and becomes **ready** as a result of sending the **fork** and **forkAt:** messages to a block.

- A **ready** process becomes **active** if it is the longest waiting at its priority and there are no higher priority ready processes and: (1) the active process becomes blocked or dead, (2) the ready process has higher priority than the active process, or (3) the ready process has the same priority as the active process and the following expression is executed:

Processor yield

- The **active** process becomes **ready** when it is replaced by a **ready** process under the conditions described above for transition 2.
- An **active** process becomes **blocked** when the message **wait** is sent to a semaphore which has no excess signals.
- A **blocked** process becomes **ready** when it is the first in the **waiting** queue of a semaphore and the message **signal** is sent to the semaphore.
- The **active** process becomes **dead** when it reaches the end of the block which caused process creation as a result of the **fork** or **forkAt:** messages, or when the following expression is executed:

Processor schedule

The User Interface Process

The **Smalltalk/V** user interface is driven by a single process which responds to all keyboard and mouse input events for all windows. The user interface process alternates between (1) responding to an input event, and (2) waiting for the next input by sending the message **wait** to global variable **KeyboardSemaphore**. When there is no input activity, other lower-priority processes can run. The process scheduler guarantees that there is always a lowest priority *idle* process to run when there is no other system activity.

Errors are handled by a debugger running under the user interface process whether or not the error occurs in the user interface process. If the **error:** message is sent under the user interface process, the current process is suspended and a new user interface process is created. This allows the process with the error to be debugged with the debugger. If **error:** is sent by a non-user-interface process, an entry describing the error is placed in the **PendingEvents** queue (a global variable). **PendingEvents** is polled for activity by the user interface process when there is no other input activity.

ProcessScheduler

Class **ProcessScheduler** is a subclass of **Object**. A **ProcessScheduler** controls process execution. There is a single instance of class **ProcessScheduler** maintained in global variable **Processor**. The process scheduler determines which ready process is the active process and maintains a queue of ready but inactive processes. The highest priority ready

process is selected as the active process. If there is more than one process at the highest priority, the process that has been ready the longest is chosen.

Process priorities may range between 1 and Processor topPriority.

Interrupts

Interrupts are the mechanism used for communicating asynchronous external events to Smalltalk/V. Examples of external events are keyboard inputs, mouse movements and clock ticks. The set of interrupt events can be extended by you.

The Smalltalk/V interrupt model corresponds to typical computer hardware interrupt architectures. Interrupts may be explicitly enabled and disabled. When an interrupt event occurs and interrupts are enabled, interrupts are disabled and a `vmInterrupt:` message is sent to the object at the top of the execution stack for the current process. The argument to `vmInterrupt:` is the selector of the method defined in `Process` class which services the interrupt. An interrupt routine concludes by enabling interrupts and returning to `vmInterrupt:`, which answers `self`, leaving the execution stack exactly as before the interrupt.

When an interrupt event occurs and interrupts are disabled, the Smalltalk/V places the interrupt event in a pending interrupts queue. Each time interrupts are enabled, the pending interrupts queue is examined to see if there are additional interrupts to be serviced.

The typical interrupt processing method merely signals a semaphore to resume a process to handle the event. For example, the keyboard interrupt handler in class `Process` is as follows:

```
keyboardInterrupt
  'Handle keyboard interrupt.'
  KeyboardSemaphore signal
```

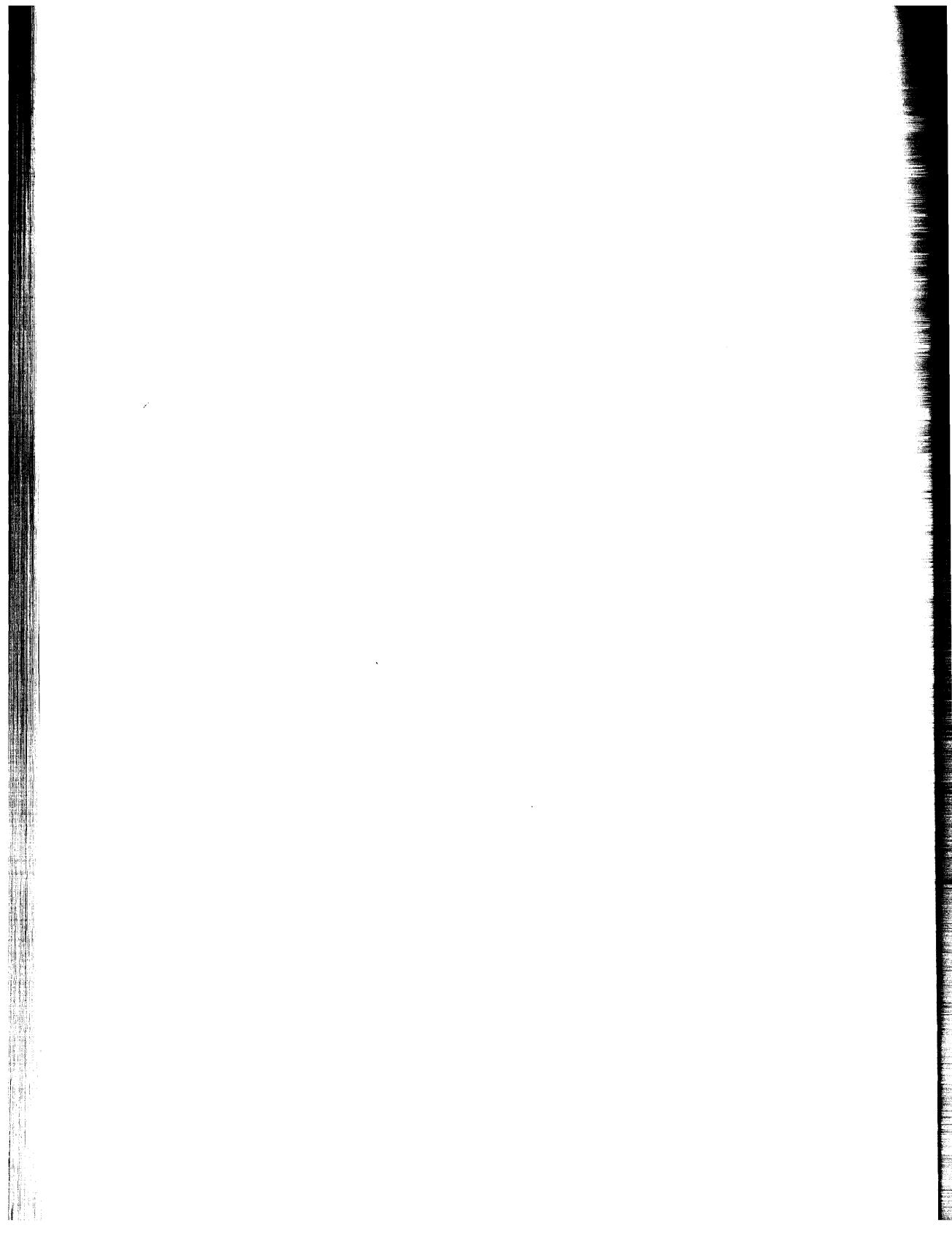
Note that the semaphore `signal` message enables interrupts if they are disabled. Interrupts are explicitly enabled and disabled by sending the message `enableInterrups:` to `Process` class. Disabling interrupts should be used with extreme caution, because Smalltalk/V cannot respond to external events while interrupts are disabled. The interrupt state may be preserved around a critical code section as follows:

```
| oldState |
oldState := Process enableInterrups: false. "disable and save state"
" ... critical code ... "
Process enableInterrups: oldState "restore interrupt state"
```

Global variable **InterruptSelectors** contains an array of selectors corresponding to interrupt events defined as follows:

<u>Interrupt Number</u>	<u>Selector</u>	<u>Event</u>
1	<code>undefinedInterrupt</code>	Interrupt number outside array bounds
2	<code>controlBreakInterrupt</code>	Control and break keys struck
3	<code>timerInterrupt</code>	Millisecond clock tick
4	<code>ioErrorInterrupt</code>	DOS critical error
5	<code>traceInterrupt</code>	Debugger hop and skip completion
6	<code>breakpointInterrupt</code>	Debugger breakpoint reached
7	<code>lowMemoryInterrupt</code>	Not much memory left after garbage collect
8	<code>keyboardInterrupt</code>	Mouse events and keyboard inputs
9	<code>overrunInterrupt</code>	Interrupts lost because queue full
10	<code>commInterrupt</code>	Communication port event

Interrupts to **Smalltalk/V** can be generated from device drivers (see **Appendix 2, Primitive Methods**). Add the selectors for the new interrupt handling methods to the end of the **InterruptSelectors** array.



15 SMALLTALK/V 286 ENVIRONMENT

This chapter describes the Smalltalk/V environment, including

- how to use windows, panes and menus;
- how to use the Smalltalk/V text editor;
- how to evaluate Smalltalk expressions; and
- how to maintain your Smalltalk/V system

Figure 15.1 shows a typical Smalltalk/V screen with three *windows*: A System Transcript, a Workspace, and a Class Browser.

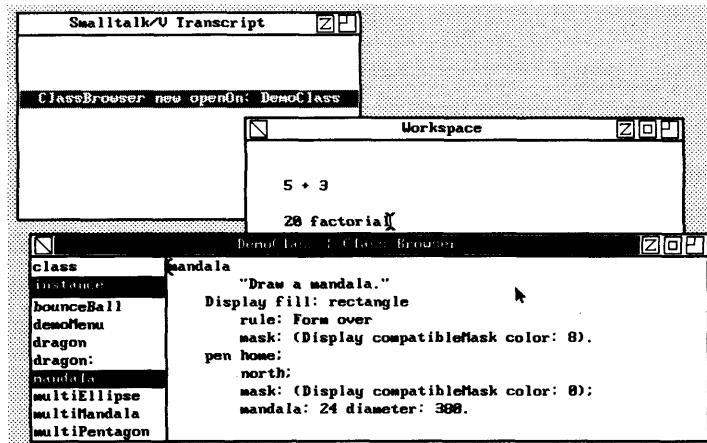


Figure 15.1
Smalltalk/V Screen

Every window in Smalltalk/V has a *label bar* and is surrounded by a *border*. The area inside the border is divided into one or more *panes*. In our example, the Workspace and the System Transcript each have a single pane, and the Class Browser has three panes.

The Keypad

The arrow that moves around the screen is the *cursor*. You move the cursor by pressing the **cursor arrow** keys on the keypad, as shown in Figure 15.2. Pressing an **arrow** key moves the cursor one space in the direction of the arrow. Pressing the **arrow** key while holding down the **shift** key makes the cursor jump several spaces in the direction of the arrow.

If you have a mouse, the left button serves as the **select** key and the right button for other functions. These keys are described throughout this chapter. You can reassign these keys, and any other keyboard keys if you wish.

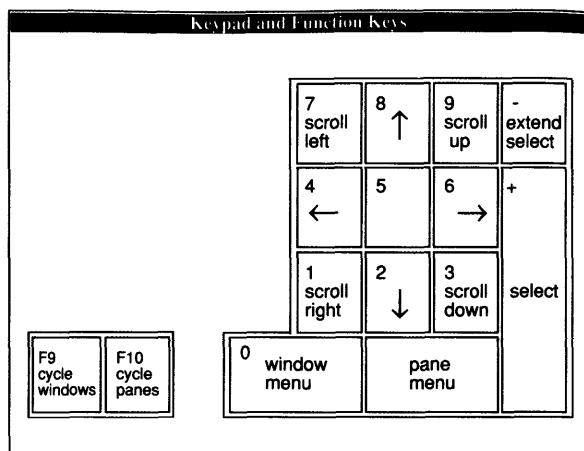


Figure 15.2
Cursor Movement Keys

If your mouse has three buttons, pressing the middle button produces a beep. You can, however, modify the behavior of the mouse, as well as anything else in the environment. Soon you will feel comfortable enough with Smalltalk/V to customize your system. If you want to program the third mouse button, start with the method `initialState` in class `TerminalStream`. Or, if you find that the scroll speed is too fast or too slow you may want to modify the method `scrollDelay`: in the class `ScrollDispatcher`.

Active Window

To *select* a window, move the cursor into the window and press the `select` key, or, if you have a mouse, position the cursor within the window and click (quickly press and release) the left mouse button.

The window that you have selected is called the *active* window. The active window is always on top of all other windows and its label bar is reversed. A window remains active regardless of whether or not the cursor is inside it. To deactivate a window, move the cursor outside of the window, and press the `select` key or the left mouse button.

The *screen* is considered the active window if you last pressed the `select` key when the cursor was outside of all windows. When no window is active, all of the label bars have a normal appearance.

Cycling

Smalltalk/V provides a fast way to move the cursor about the screen. Pressing the **cycle windows** key moves the cursor to the next window in a clockwise direction and automatically activates it. This is called *window cycling*.

Similarly, pressing the **cycle panes** key moves the cursor from pane to pane in a clockwise direction within a window. This is called *pane cycling*.

Using Menus

System Menu

Menus are the standard way of giving commands to Smalltalk/V. The *system* menu has commands for creating new windows, saving your work, exiting the environment, and redrawing the screen.

To pop up the system menu, move the cursor outside all windows on the screen, and press either the **window menu** key or the **pane menu** key, or, if you have a mouse, click the right button.

To *select a menu function*, move the cursor to the desired function and press the **select** key (the line on which the cursor sits becomes reversed to show that it is selected). Smalltalk/V immediately performs the action, and the menu disappears.

To *leave a menu* without performing any function, move the cursor outside the menu in any direction. If you have a mouse, then click the left mouse button. The menu disappears.

The other functions of the system menu are described in detail in subsequent chapters.

Window and Pane Menus

Smalltalk/V includes several menus, some of which are shown in Figure 15.3. You must leave a menu before you can pop up a different menu. The menu that you get at any time is determined by the cursor location and which **menu** key you use.

To pop up a *window menu*, move the cursor into a window and press the **window menu** key or, if you have a mouse, click the right button. If you have a mouse, the cursor must be on the label bar, any border line of the window, or any pane border line within the window. The window menu contains functions specific to that window.

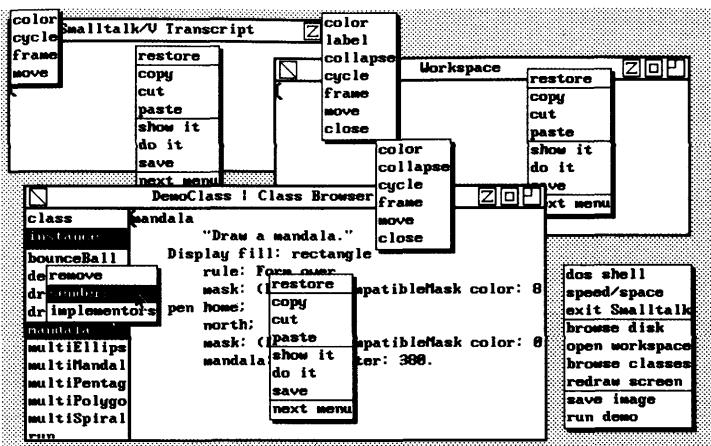


Figure 15.3
Smalltalk/V Menus

To pop up a *pane menu*, move the cursor into a pane of the window and press the *pane menu* key or, if you have a mouse, click the right button. The pane menu contains functions specific to that pane.

If you press either menu key or the right mouse button when the cursor is outside of every window, the system menu is popped up.

Manipulating Windows

Opening a Workspace

Windows are usually opened from menus. To *open a new workspace*, select *open workspace* from the system menu. An active *workspace* window opens up. This window can be closed, collapsed or resized using the label bar buttons and manipulated like other windows using the window menu.

Label Bar

The label bar, shown in Figure 15.4, displays the window title along with two or more small buttons, depending on which Smalltalk/V window is open. The buttons provide quick access to specific window activities.

To select a button, place the cursor on the button and click the left mouse button or use the numeric keypad + key.

Close Button: When selected, the window closes and disappears from the screen.

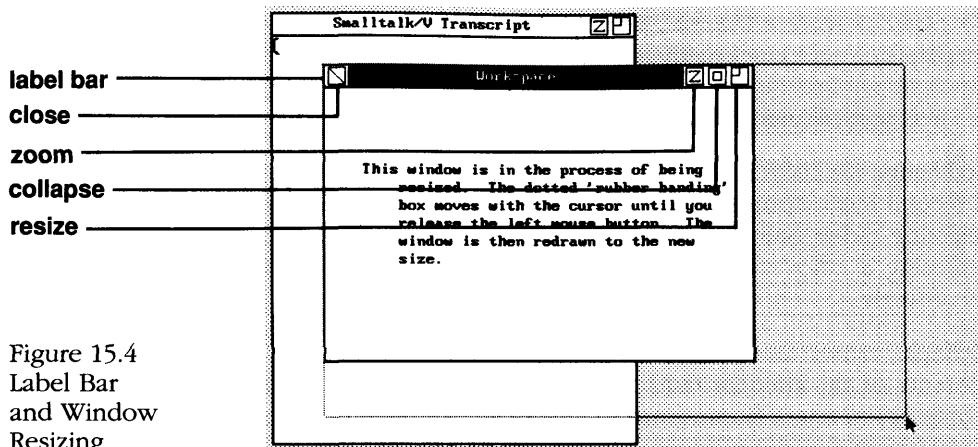


Figure 15.4
Label Bar
and Window
Resizing

Zoom Button: When selected, Smalltalk/V zooms in on the text pane so that it fills the whole screen. To unzoom the text pane, click on the label bar and the window redraws to its original configuration. You can also use the F8 function key to zoom a text pane.

Collapse Button: When selected, the window collapses to show only the label bar. If the window is already collapsed, selecting this button expands the window to its original size and position on the screen.

Resize Button: Select this button and the system responds with a rectangle outline for resizing the window, shown in Figure 15.4.

With a mouse. With the cursor on the resize button, press and hold down the left mouse button while you move the mouse to drag the cursor and resize the rectangle outline. Release the mouse button to redraw the window.

With the keypad. Select the **resize button** with the numeric keypad + key; then use the keyboard cursor keys to move the cursor and resize the rectangle outline. Press and release the numeric keypad + key to redraw the window.

Common Window Menu Functions

The window menu of every window has functions that apply to the window as a whole.

To *move a window*, select **move** from the window menu of the window you want to move or move the cursor over the window label and hold down the left mouse button. You'll then see a rectangle outline that is the size of the window being moved. Move the cursor until the rectangle outline is in the desired location and press the **select** key or release the left mouse button. Smalltalk/V moves the window to its new location.

To *frame or resize a window*, pop up the window menu and select the **frame** function. **Smalltalk/V** responds with a rectangle outline, as shown in Figure 15.4. Move the rectangle until the lower right corner is in the location you want, and press the **select** key or click the left mouse button. **Smalltalk/V** changes the window's size.

To change the color of a window, pop up the window menu and select the **color** option. A second menu comes up from which you can choose **text color** or **background color**. Selecting either one causes a third menu containing a palette of 16 colors to appear. Choosing a color makes either the text or background color of the window change accordingly.

Collapse causes the window to be collapsed so that only the label bar is visible. The contents of the window remain intact; you just can't see them. To make the contents visible again, select **frame** from the menu and reframe the window or select **collapse** a second time and the window will open to its previous size and position.

Cycle causes a window underneath other windows to pop up on top and become the active window. This is the same as pressing the **window cycle** (F9) key.

Label lets you type in a new label for the window. When **label** is selected, **Smalltalk/V** pops up a prompter window where you can type a new label. Just type the new label, and press the **return** key when finished.

Close causes the window and its contents to disappear.

Panes

Windows are composed of one or more panes. There are three kinds of panes: *text* panes, *list* panes, and *graph* panes.

Many of the windows in the **Smalltalk/V** environment have panes that allow you to modify or edit text. These are called *text* panes. The System Transcript is a window with only a single *text* pane, as is any Workspace. These and all other *text* panes use the same *text editor*.

Other windows in the **Smalltalk/V** environment have panes that allow you to select from a list of items. These are called *list* panes. Windows that have *list* panes are usually referred to as *browsers*. The Class Browser is an example of a window with two *list* panes and one *text* pane.

The third kind of pane, *graph* pane, allows you to draw pictures. Since *graph* panes are not used by any of the standard windows in the **Smalltalk/V** environment, they are not explained in this chapter. Refer to the graphics and windows tutorials and reference sections for more information.

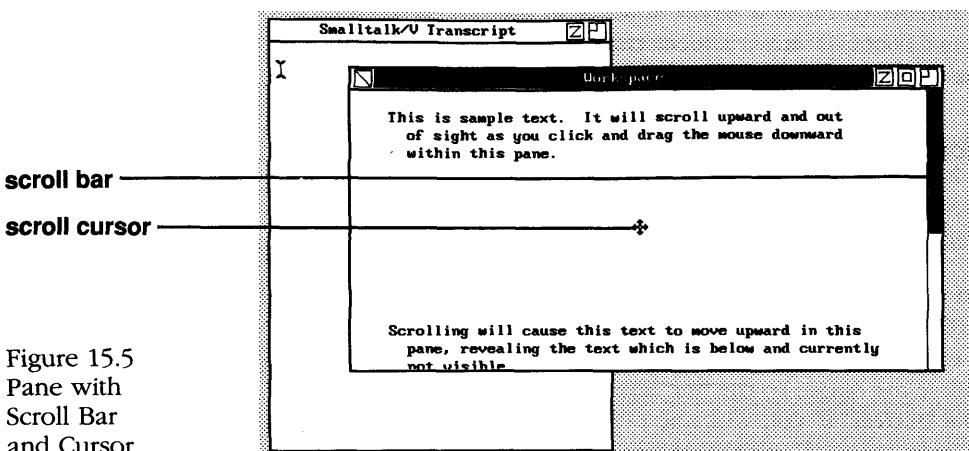


Figure 15.5
Pane with
Scroll Bar
and Cursor

Scrolling

There are many instances when the contents of a pane are larger than the pane itself. You can think of a pane as a screen which lets you see only a portion of a pane's contents, as shown in Figure 15.5. *Scrolling* lets you move this screen around the text so that you can see a different portion of it.

You can scroll through a file using either the mouse or keypad keys. However, when you use the mouse, you have access to the scroll bar.

The scroll bar displays the relative location of the visible contents of the file to the file as a whole. By positioning the scroll cursor on the scroll bar and releasing the right mouse button, you *quick jump* around the file.

Experiment with this often missed feature. It is particularly useful for use with large files where scrolling would be time consuming.

Scrolling With the Keyboard

To scroll *vertically*, use the PgUp and PgDn keys. Pressing PgUp moves the pane contents up one line, while pressing PgDn moves the pane contents down one line.

To scroll *horizontally*, use the Home and End keys. Pressing Home moves the pane contents four characters to the right. Pressing End moves the pane contents four characters to the left.

Holding down the **shift** key when pressing any of the four scrolling keys causes the pane contents to move a larger amount in the indicated direction. For vertical scrolling, the pane contents move almost the entire height of the pane. For horizontal scrolling, the pane contents move one-half of the pane width.

Scrolling With a Mouse

There are two ways to scroll a pane using a mouse. One way is to grab a character and pull it to the position in the pane where you want it. Continuous scrolling is also available.

In the *grab and pull* technique, the grab location is the position of the cursor when you press the right mouse button. Position the cursor at a character in the last line of a text pane in the active window. Press the right mouse button. Keeping the button down, move the cursor to the first line of the pane and a few characters over to the left. Release the button. If you did not move the cursor out of the pane before you released the button, you will see that the text has been pulled to the location of the cursor when the button was released.

For a *horizontal continuous scroll*, press the right button while the cursor is in the pane you want to scroll. Holding the button down, move the cursor to the right of the pane, then back inside the pane, then move it outside to the left of the pane. Do not allow the cursor to go above or below the pane. Now release the button. While the cursor is back inside the pane, scrolling pauses. When the cursor is moved back out of the pane, scrolling is resumed. When the button is released, scrolling ends.

For a *vertical continuous scroll*, press the right button down while the cursor is in the pane you want to scroll. Holding the button down, move the cursor above the pane or on the top border of the pane. Notice that the pane scrolls down. Similarly, if you move the cursor below the pane or on the bottom border, the pane scrolls up.

Vertical scrolling speed is controlled by the horizontal position of the cursor in relation to the width of the pane. As you move the cursor to the left, the speed slowly decreases until you move the cursor past the left border of the pane. At this point, scrolling speed is at its slowest. As you move the cursor to the right, the speed gradually increases until you move past the right border of the pane. Now scrolling is at its maximum speed, almost a page at a time.

To *pause* during the scroll, move the cursor back into the pane. To *resume* scrolling, move the cursor out of the pane as you did to start the scroll. To *terminate* scrolling, release the button.

To *quick jump* through large chunks of text without scrolling, position the scroll cursor on the scroll bar to locate the section of the file you want to see, and release the right mouse button. The pane will now show the area of the file you located on the scroll bar.

Text Editor

Inserting Text

Whenever you press the **select** key, an I-beam is drawn between two characters to mark the *text insertion point*. As you type characters, they appear in front of the text insertion point, which moves to the right with each keystroke. If you press the **backspace** key, the characters in front of the insertion point is deleted, and the insertion point moves a space to the left.

To *move the insertion point*, place the cursor at the new position and press the **select** key. The insertion point jumps to the cursor position.

Whenever the cursor is outside the text editing pane, if you try to type a character, your system beeps, and the character does not appear. You can only edit the contents of a text pane if the cursor is inside the pane.

Return, Tab, and Backspace Keys

Pressing the **return** key moves the insertion point to the beginning of a new line. If the insertion point was in the middle of a line, the line is split into two lines at the insertion point.

If you press the **tab** key, blanks will be inserted at the insertion point to make it move to the next tab stop. In this editor, tab stops occur every four spaces.

If you press the **backspace** key, the character in front of the insertion point is deleted. Notice that if the insertion point is at the beginning of a line and backspace is pressed, the line is joined to the line above.

Selecting, Replacing, and Deleting

To *select text*, move the cursor to either the beginning or end of the text to be selected. This position can be at any character, even in the middle of a word. Press the **select** key to bring the insertion point to the cursor. Now move the cursor to the other end of the text to be selected (again, at any character position) and press the **extend selection** key. The entire selected area is reversed. Notice what happens if you move the cursor and press the **extend selection** key again. The selected (reversed) area of text now ends at the new cursor position. You can continue to move the cursor and adjust the selected text by pressing the **extend selection** key until you've selected the text you want as shown in Figure 15.6.

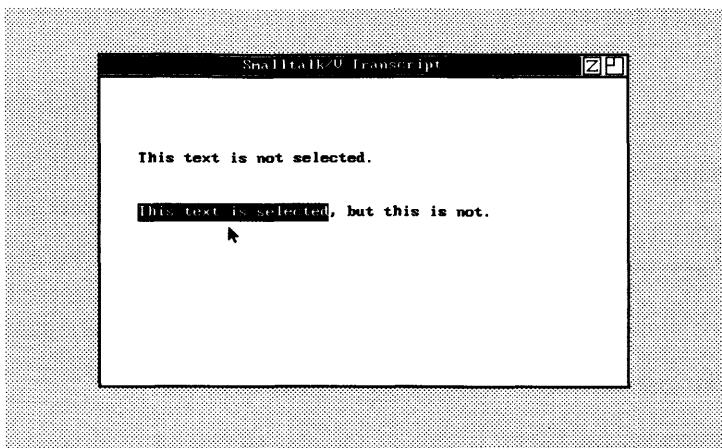


Figure 15.6
Selected Text

If you press the **select** key, the selected text is deselected (no longer reversed) and the insertion point appears at the cursor.

Selecting text with a mouse works in a similar fashion.

To *insert text* in a *text pane* of the active window, position the cursor where you want the insertion point to be, then click the left mouse button. Now the I-beam appears and you can type.

To *select text* in the active window, you can use a method similar to using the **extend selection** key. First move the cursor to the point where you want to begin or end your selection, and click the left button. To extend the selection, move the cursor to the other end of the text then hold down the **shift** key, and press the left mouse button. The text selection will now extend to the current cursor location.

You can also select text using the *draw through* method. Position the cursor at one end of the text you want selected. Now press the left mouse button. Hold the button down as you move the cursor to the other end of the selection. Notice that as you move the cursor, the selection (the reversed text) follows and includes the current cursor location. Draw through terminates when you release the button. Note that if you move the cursor outside the pane while holding the left mouse button down, the pane will scroll while extending the selection.

To *select a single character*, position the cursor at the character you want to select. Then press down the left button and hold it until the character is reversed. This takes only a fraction of a second.

To *select a whole word*, position the cursor anywhere on the word to be selected and double click the left mouse button. The word is reversed.

To *select a line*, place the cursor just inside the window border to the left of the line you wish to select and click the left mouse button twice.

To *replace* text, select the text to replace and then type the new text. As you type, **Smalltalk/V** replaces the selected text with the new text.

To *delete* large amounts of text, select the text and press the **backspace** key. To delete text one character at a time, place the insertion point at the end of the text to be deleted and press the **backspace** key.

Saving and Restoring

The text editor in **Smalltalk/V** is always working with a text copy of some underlying object in the system. Among other things, this can be a file, a string, or some **Smalltalk** code. For example, the System Transcript and any Workspace edit strings of characters, and the text pane in the Class Hierarchy Browser edits **Smalltalk** code. Because a copy is being edited, you must tell the environment when editing is complete. **Smalltalk/V** can then store the edited text back in the original place. You can also restore the text in the pane to its previous state, if you wish.

Every pane that allows text editing has editing functions on its pane menu.

The *restore* function replaces the text in the pane with the text representation of the underlying object being edited. If you are editing a string, then the string replaces the text in the pane. If you are editing a file, the text in the file is reread into the pane. If you are editing **Smalltalk** code, then the source code currently being used by the system is put in the pane.

The *save* function in the pane menu tells **Smalltalk/V** that editing is complete. **Smalltalk/V** responds by updating the underlying object being edited with the text currently in the pane. For example, if a file is being edited, then the file is rewritten with the text in the pane.

Cutting, Copying, and Pasting

The text editor has a buffer that can be used to transfer text from place to place. Text is placed in this buffer by either *cutting* or *copying* it from the pane. After text has been placed in the edit buffer, it can be *pasted* anywhere inside of a text pane. This is shown in Figure 15.7.

To place text in the edit buffer you must first select the text. Then, pop up the pane menu and select either *cut* or *copy*. If the *cut* function is selected, **Smalltalk/V** responds by replacing the contents of the edit buffer with the selected text, and then deleting the selected text from the pane. If the *copy* function is selected, the selected text is not deleted from the pane. Instead a copy of the text replaces the contents of the edit buffer.

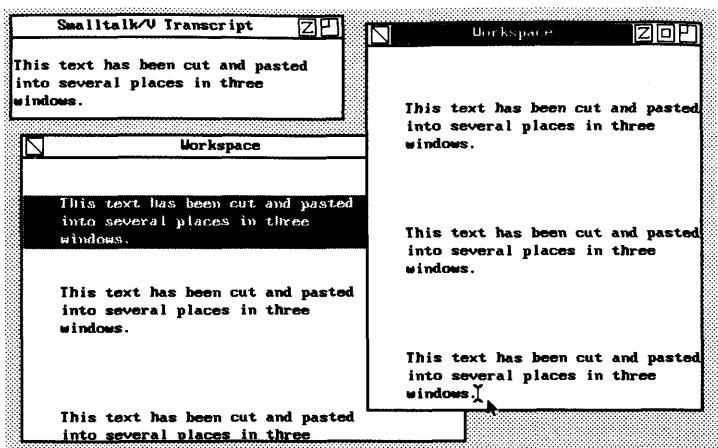


Figure 15.7
Cutting and Pasting

Text in the edit buffer can be pasted into the pane by either inserting it into a new place or replacing some existing text. To insert the contents of the edit buffer, place the insertion point at the desired position and then select the **paste** function from the pane menu. To replace some text with the contents of the edit buffer, first select the text to be replaced and then select the **paste** function from the pane menu.

The **paste** function leaves the contents of the edit buffer unchanged. This means that the same text can be pasted several times and in different places.

Since the same edit buffer is shared by all of the text panes, you can easily transfer text between windows. First select the text to be transferred. Next, place this text into the edit buffer by using the pane menu to either cut it or copy it. Then move the cursor into the new window and either place the insertion point where the text is to be inserted or select some existing text that is to be replaced. Now, use the **paste** function from the pane menu. The text appears in the new window.

Saving the Image

The **image** is all of the Smalltalk objects, both code and data, that make up the Smalltalk/V environment. The image is read from the **image** file when the system starts up. In this way, the objects are loaded into memory. These objects include the windows that appear on the screen.

Since Smalltalk/V is an interactive and modifiable environment, the image is constantly being changed as you use and modify Smalltalk/V. These modifications are not written on the disk until you ask them to be written. This can be done at any time by selecting

the *save image* function from the system menu. You can also save the image while exiting as described below. The next time you start up the system, it resumes exactly as it was when the image file was last written.

The **image** file on the disk represents the last saved version; therefore it also becomes the starting point in the event of a system crash or a major mistake. For these reasons, you should maintain a recent backup of the **image** file along with the **source** file and the **change log** that is associated with these files. **Maintaining Smalltalk/V**, later in this chapter, explains the relationships between these files in more detail, and gives instructions on maintaining them. It also gives advice on how to recover from a system crash or other problem.

Exiting Smalltalk/V

To exit **Smalltalk/V**, select **exit Smalltalk** from the system menu. You'll then see another menu, asking whether to remember or throw away the changes made since the environment was started.

Selecting the *save image* function causes the state of the system, including the location and contents of all the windows, to be saved. When you start **Smalltalk/V** the next time, it will be restored to the saved state.

Selecting the *forget image* function causes all of the changes made since **Smalltalk/V** was started to be forgotten. When **Smalltalk/V** is started the next time, it will be restored to the previous saved state (the way it was when it was started this time).

Selecting the *continue* function returns you back to the **Smalltalk/V** environment. This has the same effect as leaving the menu without selecting anything.

Evaluating Smalltalk Expressions

The Smalltalk language includes expressions which are similar to expressions in other programming languages. In the **Smalltalk/V** environment, you can enter the text for an expression in any text pane, evaluate it, and display the result. You specify what you want to do in the environment by using pop-up menus and by evaluating Smalltalk expressions, which serve as the **Smalltalk/V** command language.

Doing and Showing

All text panes in the system support immediate expression evaluation via their pane menus. To *evaluate an expression*, you must first select it, then pop up the pane menu and select either the *do it* or the *show it* function. If you choose *show it*, the expression is evaluated and a character representation of the expression value is inserted in the pane after the evaluated text. If you choose *do it*, the expression is evaluated and the expression value is thrown away. Notice that only the selected text is evaluated; the other text in the pane is ignored. Extra blanks at the beginning or the end of the selected text are similarly ignored.

Any legal Smalltalk expression or expression series can be selected in a text pane and evaluated. Temporary variables can be declared as needed.

Compilation Errors

When you select and evaluate an expression, Smalltalk/V compiles and then evaluates it. If it detects a compilation error, Smalltalk/V inserts an error message in the source code at the point of error. To fix the error, simply edit the text in the window, and evaluate the expression again.

Making Command Templates

You do not need to type the expression that you want to evaluate. You can edit some existing text in the pane and then select and evaluate it. This feature makes it possible to build a window or even a file of useful expressions that you can edit as command templates to construct new expressions for evaluation. File `sample.sml` contains several useful expressions. You can use the Disk Browser to look at and evaluate the expressions contained in this file. These expressions all have comments explaining them.

You can add to the expressions in this file (or make files of expressions of your own) using the Disk Browser. You can include comments with your expressions by merely enclosing the comment in double quotes like this: "this is a comment".

Prompters

Prompters are a special kind of text editing window that appear on the screen when you request something that requires additional information. The label of a Prompter is a message called a prompt, which tells you what information Smalltalk/V needs to finish the operation. You enter your response into a single text pane of the Prompter. Often a default response appears in the text pane as shown in Figure 15.8.

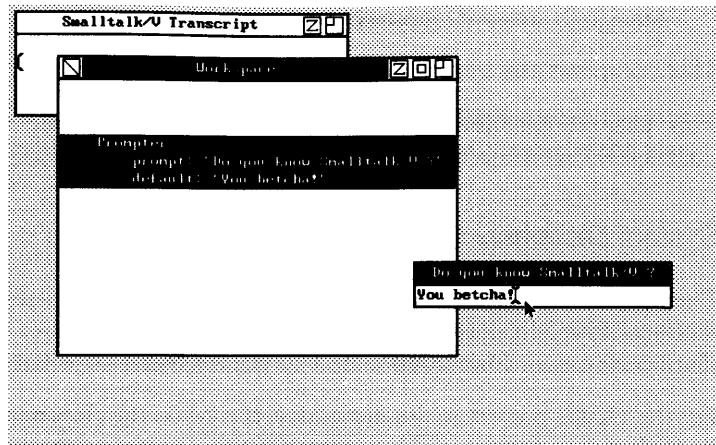


Figure 15.8
Prompter

The text editor used by a Prompter is the same as the text editor described under **Text Editor** above, except for the following differences:

Smalltalk/V uses Prompters to request a single piece of information from you. You must answer the request or tell **Smalltalk/V** that you want to cancel whatever operation is requesting the information. Since you must respond to the request, **Smalltalk/V** will not deactivate the prompter window until you do so. In addition, a Prompter has no window menu. **Smalltalk/V** will beep if you press the **window menu** key.

The pane menu of a Prompter (which you pop up by pressing the **pane menu** key) gives you two choices which are specific to Prompters. When you select the *accept* function, the Prompter sends the text in the pane to the requestor of the information. When you select the *cancel* function, the Prompter sends nil to the requestor. The requestor will cancel the operation. Pressing the **return** key is the same as selecting the *accept* function from the pane menu. After the request is answered, either by accepting or canceling, the prompter window disappears.

The System Dictionary

There is a class in **Smalltalk/V** known as **SystemDictionary**. There is only one instance of this class, the **System Dictionary**. The System Dictionary defines methods for system-oriented functions such as compressing the **change log** and determining available memory. It also contains all of the names known globally in the system. This includes the names of the classes, global variables, and pool dictionaries. The System Dictionary is referred to in Smalltalk code with the name **Smalltalk** which identifies a global variable. All global variable names must begin with a capital letter. An example of a message to the System Dictionary is the following expression:

Smalltalk unusedMemory

This computes the number of bytes of available memory in your system.

This chapter describes the System Dictionary and some of the system commands (messages) that are sent to it.

System Dictionary Contents

The System Dictionary contains all of the names of the classes, global variables and pool dictionaries in the system. In addition, as you define classes, global variables, and pool dictionaries, their names are added to the System Dictionary.

For the classes present in the dictionary, the key is the class name and the value is the class itself. For the global variables present in the dictionary, the key is the variable name and the value is an object. If the variable has been defined but not initialized, its value is nil. Some of the more important global variables present in the system are listed below, along with their values and a brief description. The pool dictionaries are explained in more detail at the end of this chapter.

<u>Variable</u>	<u>Description</u>
Aspect	A Fraction which contains the aspect ratio of your display.
CharacterConstants	A pool dictionary. CharacterConstants associate names with special character values such as: carriage return, line feed, escape, form feed, etc.
CurrentEvent	An InputEvent used by the environment for reading all keyboard and mouse events.
CurrentProcess	The Process that is currently running.
Cursor	A CursorManager used to store the location of the cursor.
Disk	A Directory . The current directory when the system is booted.
Display	A DisplayScreen which is used to access the video screen as a form.
FunctionKeys	A pool dictionary. This dictionary defines names for keyboard function key values and for mouse input codes.

<u>Variable</u>	<u>Description</u>
KeyboardSemaphore	A Semaphore that is signaled every time there is a keyboard or mouse interrupt. CurrentEvent waits on this Semaphore .
Processor	The ProcessScheduler of the system.
Scheduler	A DispatchManager which manages the scheduling of windows. Scheduler should be the only instance of class DispatchManager in the system.
Smalltalk	The SystemDictionary .
Sources	An Array containing two file streams for accessing the source code of Smalltalk/V . The first entry is the source file. The second entry is the change log file.
SysFont	The default Font used for the current graphics resolution.
Terminal	A TerminalStream which is used for reading the keyboard and mouse, if any.
Transcript	A TextEditor which is the System Transcript. Transcript is used primarily for system messages. The user can also send his messages to it (e.g., for debugging traces).

System Dictionary Methods

There are many useful functions or methods which operate on the System Dictionary. They include:

<u>Method</u>	<u>Description</u>
at: <i>key</i> put: <i>anObject</i>	Creates a new global variable, <i>key</i> , with the value <i>anObject</i> . If the global variable exists, changes its value to <i>anObject</i> .

<u>Method</u>	<u>Description</u>
compressChanges	Compresses the change log file, removing all but the latest version of the methods.
compressSources	Compresses the source file after updating it with the latest version of the methods in the change log file.
implementorsOf: aSymbol	Returns a Method Browser containing all of the classes which define a method with the name <i>aSymbol</i> . This is equivalent to choosing the implementors function on the method list pane menu of the Class Hierarchy Browser.
keys	Returns all of the keys currently defined in the System Dictionary.
removeKey: key	Removes the specified key from the System Dictionary.
sendersOf: aSymbol	Returns a Method Browser containing all of the methods, and their corresponding class names, which send a message with name <i>aSymbol</i> . This is equivalent to choosing the senders function on the method list pane menu of the Class Hierarchy Browser.
unusedMemory	Returns the number of bytes free in the system.

Pool Dictionaries

A dictionary contained in a global variable can be used as a pool dictionary.

CharacterConstants

CharacterConstants is a pool dictionary associating names to special ASCII character values such as line feed and escape. The keys of this dictionary are descriptive strings and the values are the associated ASCII characters as shown below:

<u>Key String</u>	<u>ASCII Character</u>
Bell	Bell character (ASCII value 7)
Bs	Back space character (ASCII value 8)

<u>Key String</u>	<u>ASCII Character</u>
Cr	Carriage return (ASCII value 13)
Del	Delete character (ASCII value 127)
Esc	Escape character (ASCII value 27)
Ff	Form feed character (ASCII value 12)
FunctionPrefix	First character of a function key two character sequence (ASCII value 0)
Lf	Line feed character (ASCII value 10)
MouseButton	Character sent when any mouse button changes state (ASCII value 254)
SetLoc	Character sent when the mouse moves (ASCII value 255)
Space	Space character (ASCII value 32)
Tab	Horizontal tab character (ASCII value 9)
UpperToLower	SmallInteger of value 32, the numeric difference between upper and lower-case ASCII value characters

FunctionKeys

FunctionKeys is a pool dictionary associating names to the function key values such as cursor movement keys and menu keys. By assigning a new value to a function key name, a different key on the key board can be associated to the named function.

Maintaining Smalltalk/V

In this section, the procedures for maintaining the system are discussed. These include:

- How to keep the files that the system uses from getting too big.
- Some simple steps that you can take to protect yourself from losing all your work in the event of a total failure (crash) of the system.
- What to do if the system does crash.

Automatic Logging of Changes

As you define and modify classes or methods, **Smalltalk/V** is logging all of these changes to the **change log**. **Smalltalk/V** maintains pointers from the compiled methods in the environment to the latest version of the source code. If the source code has never been changed, the pointer goes to the **source file**. Otherwise, the pointer goes to the **change log**. As a result, the disk is accessed as you browse the methods in a class. Furthermore, since the **image** file contains pointers to the **source file** and the **change log**, you must maintain these three files as a group.

For these reasons, *you must not edit or modify the change log!* You can view it with the Disk Browser and you can reinstall methods and class definitions from it into your system. This capability can be used to facilitate recovery from a system crash and is explained in more detail in the section on **Surviving a System Crash** in this chapter.

Other important events are automatically logged by **Smalltalk/V**. Every time the **image** is saved, a message with the date and time is written to the **change log**. Every **Smalltalk** expression that you evaluate with either *do it* or *show it* is also logged. In addition, every time you remove a method from a class, a message is logged.

The format of the **change log** is very similar to the one described by Glenn Krasner in Chapter 3 of *Smalltalk-80: Bits of History and Words of Advice*, Addison-Wesley, 1983. A brief description of this format using the EBNF notation described in **Appendix 1** follows.

```

<rule> changeLog =
    {textChunk}.
<rule> textChunk =
    sourceCode | classDefinition | expression | imageComment.
<rule> sourceCode =
    '!" className' methods !' { method '!" } '!".
<rule> classDefinition =
    ' "define class" ' classDefinitionMessage '!".
<rule> expression =
    ' "evaluate" ' evaluatedText '!".
<rule> imageComment =
    ' "*** saved image on: ' dateAndTime '***" !'.

```

where: *className* is

the name of a class.

method is

a valid method as defined in Part 2 of this manual with occurrences of "!" replaced by "!!!".

classDefinitionMessage is

a valid class definition message to the class's superclass as defined in Part 2 of this manual.

evaluatedText is

the text evaluated by *do it* or *show it* with all occurrences of "!" replaced by "!!!".

dateAndTime is

the character representation of the date and time the image was saved.

The **change log** is an ASCII text file which means that you can print it at any time. A simple examination of the file should remove any questions regarding its format.

Importance of Saving the Image

Smalltalk/V is written entirely in Smalltalk. The image that is referred to in this manual is the collection of all Smalltalk objects that make up the environment.

When you save an image, descriptions of all of these objects are written out on the **image** file. When you start the system at a later time, all of the objects described in the **image** file are recreated exactly as they were at the time the image was saved. Since these objects contain everything that makes up the Smalltalk/V environment, the system starts up exactly as you saved it.

As you use the system by writing Smalltalk code or using the environment for other purposes, new objects are created and existing objects have their contents changed. You must save the image to make your changes permanent. You can save the image by either

selecting the *save image* function from the system menu or by selecting the *save image* function as you are exiting the environment. If you do not save the image, the environment will not have any of your changes the next time it is started.

If you forget to save the image, you can recover these changes. Items that were logged in the **change log** are still present, and you can install them again using the Disk Browser (see **Surviving a System Crash** later in this chapter). Not saving the image is useful when you are developing something new and you decide that you have made a major mistake that you would like to discard.

Compressing the Change Log

Compressing the **change log** reduces it to only the latest copy of each new or changed method. Class definitions are removed. A new **image** file is written automatically.

You should compress the **change log** when it starts to get large. There must be enough space on the disk for both the new and the old **change log** at the same time, so make sure you compress the **change log** *before* the disk gets too full.

To *compress the change log*, evaluate the following expression in any text pane (the System Transcript for example):

Smalltalk compressChanges

If there is not enough room on the disk, delete some files and then try it again.

Compressing the Source File

Compressing the **source** file creates a new **source** file for all of the methods currently in the system by taking the latest copy of the source code for each method from the old **source** file or **change log**. A new **image** file and an empty **change log** are automatically written.

The **source** file is written in a compressed format to preserve space on the disk. There are pointers from the compiled methods in the **image** to the **source** file and to the **change log**. For these reasons, it is *imperative* that the **source** file not be edited or altered except by **Smalltalk/V**. Copying the **source** is okay. You should always use the same **image** file with the same **source** file (or a copy of it) and the same **change log**.

After you compress the sources, you will have a new **image** file and a new **source** file. The **change log** will be empty. Therefore, you should make backup copies of the **source** file, the **change log** file, and the **image** file before compressing the sources so if anything goes wrong, you can recover.

To compress the sources, evaluate the following expression in any text pane (the System Transcript, for example):

Smalltalk compressSources

After compressing the sources, you have a new base system. You should make a backup copy of the new source, image, and change log files. Maintaining backup copies will allow you to recover from most, if not all, problems that might occur.

Retaining your old change logs gives you a record of all the changes you have made to the system. This will prove invaluable when you receive a new release of Smalltalk/V.

Surviving a System Crash

Smalltalk/V is very resilient to errors. Unfortunately, disasters can and will happen, especially if you are making modifications to Smalltalk/V. The automatic logging feature of Smalltalk/V makes it possible to recover from most disasters. Disasters happen not because Smalltalk/V has bugs in it, but because it is a modifiable program development environment. Smalltalk/V will let you change anything, even if it destroys the environment.

For example, if you make a mistake changing the method for the walkback window, or if the hardware fails, you may see the error message: `doesNotUnderstand: is missing`. This means that Smalltalk/V cannot find the code that displays the appropriate walkback message and has terminated.

This presents no problem since you can recover most, if not all, of your work very quickly. In fact, experimenting with the system is a good way to learn about it, but make sure you take the following precautions:

1. Always have a backup copy of the source file that you are currently using. Smalltalk/V never modifies the source file and neither should you. When you compress the sources, as explained in Compressing the Source File, a new source file is made. Make a backup copy of it immediately.
2. Always have a reliable backup copy of the image file and change log file. Remember that the image and the change log work together. If you use an image file with an older version of the change log, you may not be able to access the source code for some methods. After you compress the changes (see Compressing the Change Log), the image and change log files are both replaced. You cannot use an old image file with a new compressed change log file. You can use an old image with a later version of the same change log if the change log has not been compressed between the time when the image was made and when you use the change log. *In short, back up the image and change log together, and you shouldn't have any problems.*

3. You should save the **image** onto the disk (you can do this from the system menu) before trying something that might crash the system. Doing so will preserve most of your work. If the system crashes, you can restart Smalltalk/V using this saved image. If you fail to save the **image** and the system crashes, you can use the Disk Browser to examine the **change log**.
4. If you have been experimenting with Smalltalk/V and you have any reason to believe you have damaged the system or if there are changes you do not want to make permanent, *do not* save the **image**. Exit the Smalltalk/V environment using the **forget image** function. Remember, most of the changes you have made are in the **change log** and they can be reinstalled selectively.
5. If Smalltalk/V crashes, don't panic. Make a copy of your **image** and **change.log** file. Usually, you can recover your work, and if you make a copy, you can try repeatedly. Read the next section for information on how to recover.

Recovering From a System Crash

If the system crashes you should make a copy of your **image** and **change.log** file before proceeding. To recover your work, follow these steps:

1. Check if you have a good **image** file. To find out, try to start Smalltalk/V in the usual way using this file. If it does not start up, then you should get your most recent backup of the **image** and **change log** and start Smalltalk/V with that version.
2. Now that you have Smalltalk/V running, use the Disk Browser to look at the **change.log** file that you were using when the system crashed. The **change log** has all of the modifications that you made and all of the expressions that you evaluated.
3. Find the point in the **change log** where you saved the image that you are currently running. Every time that the **image** is saved, Smalltalk/V writes a comment into the **change log** giving the time and date when the **image** was saved. Scan the **change log** backwards, if you were using a recently saved **image**.
4. Now that you have found the point at which you saved the image, you know that your lost work is from this point to the end of the **change log**. Remember that the **change log** is formatted as a series of chunks. The exact format is given in the section **Automatic Logging of Changes** in this chapter. To restore your work, select one or more consecutive chunks of text. You must select complete chunks—watch the exclamation points (!). Next,

pop up the *pane* menu and select the *install* function. If a chunk is an expression, it will be evaluated. If a chunk is a method definition, it will be recompiled and installed into the system. Choose the chunks that you install carefully—one of them represents the error, and you don't want to put it back into the system.

When you have finished recovering what you want, save the **image**.

Purging Unused Symbols

Unused symbols tend to accumulate slowly over time in **Smalltalk/V** because instances of class **Symbol** are not collected as garbage. Evaluate the following expression to remove all symbols that are no longer referenced.

Smalltalk purgeUnusedSymbols

The number of symbols recovered and the amount of space reclaimed is usually quite small.

Cloning

Cloning is a process which builds a new **image** file from the current **image**. The Cloner does a complete memory walk of all of the objects in **Smalltalk/V**. Cloning does not in and of itself reclaim any space or objects. The **image** file is fully compacted every time the **image** is saved. Cloning does not remove unused symbols. Cloning the system is a good way of verifying that all of the objects in the **image** are undamaged.

The Cloner can also be used to build a version of the system that has many of the classes removed. In order to specify these classes and objects, you must edit the method **initializeClones** in **cloner**.

Note that when you clone, the **image** file will be re-written. Therefore, you should back up your **image** file before cloning.

The Cloner classes are not resident in **Smalltalk/V**. To clone **Smalltalk/V**, an expression must be evaluated to read and execute a file. The file installs several classes into **Smalltalk/V** and then uses them to clone the system. The cloned system does not contain the cloning classes.

If the file **cloner** is in your disk directory, (the same directory as the **image** file), then evaluate the following expression in any *text* pane, (e.g. the System Transcript):

(Disk file: 'cloner.st') **fileIn**.

When cloning is finished, **Smalltalk/V** will exit to DOS. You should make a backup of your **image** file.

DOS Shell

The DOS Shell capability allows you to quickly exit Smalltalk/V to execute DOS commands and programs, and then return to Smalltalk/V when finished. For example, you could use this capability to format a floppy disk or to run a word processing program, and then return to Smalltalk/V.

Choosing dos shell from the system menu pops up another menu with several choices of single DOS commands you can execute. In addition, a to dos choice exits you to the DOS command processor COMMAND.COM from which you can run several DOS commands and programs. To resume Smalltalk/V use the DOS EXIT command.

Reserving Space for DOS

In order to use this feature you must reserve space for DOS when starting up Smalltalk/V. The memory you reserve is not used by Smalltalk/V. You specify the amount of space to reserve for DOS as the argument to the /d: parameter on the command line that invokes Smalltalk/V. For example, the following command invokes Smalltalk/V and reserves 200K for DOS to use.

```
v /d:200
```

Notice that the argument is a decimal number and specifies the amount of memory to reserve in multiples of 1024 bytes. Appendix 3, Configuring Smalltalk/V 286, explains all of the command line options in detail.

How It Works

The method `executeCommands:` in class `ScreenDispatcher` does all of the work. For example, this expression displays text on the display screen using the DOS echo command:

```
Scheduler systemDispatcher
  executeCommands: #'('ECHO hello from Smalltalk/V' 'PAUSE')
```

The argument is a collection of strings containing the DOS commands to execute, in this case an array with two strings containing the echo command and the pause command. These strings are written to the file `smalstalk.bat`, along with a few DOS commands to return to the appropriate directory so Smalltalk/V can resume properly. Smalltalk/V runs the DOS commands by executing this batch file.

As a final example, here is the method executed in `ScreenDispatcher` when you select go to DOS from the DOS menu; it exits to the DOS command processor COMMAND.COM:

```

gotoDOS
    "Exit to COMMAND.COM"
    self executeCommands:
        #'ECHO OFF
        'ECHO TO resume Smalltalk/V, enter EXIT'
        'CD \
        'COMMAND.COM')

```

Font and Cursor Shapes

Font

Smalltalk/V represents characters in strings using their ASCII codes. In order to display them on the screen or printer in bitmapped graphics mode, these ASCII values must be converted into bitmap images. Class **CharacterScanner** (refer to Chapter 14) performs the conversion while instances of class **Font** provide the character bitmap image needed for conversion, and information about how to retrieve the image.

Smalltalk/V includes two fonts. This section provides you with the information about how to install your own fonts.

The expression **Font eightLine** returns a **Font** whose characters have a size of 8 by 8 pixels, while the expression **Font fourteenLine** returns a **Font** with characters of 8 by 14 pixels.

The following global variables govern the fonts used by different parts of the system:

<u>Global Variable</u>	<u>Used by</u>
LabelFont	window labels
ListFont	ListPane
TextFont	TextPane
SysFont	all others

You can change the first three global variables to contain different fonts; when you open new windows, you'll see the new fonts being used. To change the **SysFont** to **aFont**, use the expression

Font setSysFont: aFont.

The instance variables of **Font** are the following:

charSize

Contains a Point representing the width and height of each character.

glyphs

Contains a Form of bitmap images of all characters. The images are attached horizontally. For example, if the font contains 100 characters and charSize is 8 by 14, then its **glyphs** Form will have a width of 800 (100 * 8) and height of 14. Each Font can contain a maximum number of 256 characters.

xTable

Contains an Array of the **x** coordinate of the origin of each character image in the **glyphs** Form. It contains one entry more than the number of characters contained in the Font.

startChar, endChar

Contain the ASCII value of the first and last character, respectively, in the Font. There cannot be any missing values between **startChar** and **endChar**. For characters with an ASCII value outside of the range **startChar** to **endChar**, the glyph for **endChar** is used.

basePoint

Contains a Point whose **x** value is always 0 and whose **y** value indicates the base line of the font.

To install a fixed width new font, evaluate the following expression:

Font new

```
installFixedSize: glyphForm
charSize: sizePoint
startChar: x
endChar: y
basePoint: bPoint
```

This initializes all the instance variables and automatically creates the **xTable**.

Cursor Shapes

Smalltalk/V uses different cursor shapes to visually indicate system status. For example, the hourglass cursor shape is used to indicate a computation is in progress. Cursor shapes are also a good way to convey status information about your applications.

Smalltalk/V supports the Microsoft Mouse and its compatibles (e.g. PC Mouse, Logitech Mouse). It therefore has two interfaces to deal with cursors. When you have a mouse, the cursor is managed by class **CursorManager**. When you do not have a mouse, it is managed by class **NoMouseCursor**. **CursorManager** merely provides an interface between **Smalltalk/V** and the mouse driver supplied by the mouse manufacturer. **NoMouseCursor**, on the other hand, handles with **Smalltalk** code everything related to

the cursor. Smalltalk/V automatically detects whether or not there is a mouse and sets a global variable, **Cursor**, to be an instance of the appropriate class. The user interface to **Cursor** is identical for either class.

Normally when you display something that covers the cursor, you should first hide the cursor; otherwise, cursor writing by the mouse driver may interfere. To simplify this problem, all system primitives that can alter the contents of the screen currently make sure that the cursor is hidden before the altering, and restored after.

For the advanced user, the expression **Cursor hide** hides the cursor and **Cursor display** displays it. These two messages work like parentheses: they must be balanced. For example, suppose the cursor is currently being shown; if you hide it twice and then display it once, you must still display it again to show it. If you are not sure which level the cursor is at, and it is not displayed, evaluate the following expression:

Cursor reset

This forces a balance of **hide** and **display** and shows the cursor.

To change the cursor shape, simply send the message **change** to the cursor instance to which you want to change. For example:

CursorManager execute change

changes the cursor shape to an hour glass (indicated by **execute**). Smalltalk/V includes 11 prebuilt cursor shapes:

<u>Message to</u> CursorManager	<u>Shape</u>
normal	an arrow pointing north west
execute	an hour glass
origin	an angle bracket pointing north west
corner	an angle bracket pointing south east
downArrow	an arrow pointing down
upArrow	an arrow pointing up
leftArrow	an arrow pointing left
rightArrow	an arrow pointing right
crossHair	a cross
hand	a hand
scroll	four arrows pointing out from the center

To temporarily see a cursor shape on the screen, use the same expression followed by a **Terminal read**. For example:

CursorManager hand change. Terminal read

displays the hand cursor, and then returns to normal when you type anything.

Creating a cursor of your own is equally simple. First create a cursor form using the **BitEditor** included with **Smalltalk/V**. (The cursor form must have an extent of 16 @ 16.) Then evaluate the following expression:

```
NewCursor :=  
  CursorManager new  
    initialize: cursorForm hotSpot: aPoint
```

The argument **cursorForm** is the Form containing the cursor image. The hot spot **aPoint** is the point of the image that will be aligned with the actual cursor position. To make the new cursor the active cursor, evaluate:

```
NewCursor change
```

16 SMALLTALK/V 286 STANDARD WINDOWS

This section describes the special windows which **Smalltalk/V** provides for maintenance and debugging purposes.

The *Disk Browser* displays the files on a given directory, and their contents. It also lets you edit those contents.

The *Class Hierarchy Browser* shows you the interrelationship of the classes within **Smalltalk/V**, and lets you edit the code for each class.

Inspectors let you examine and edit objects. They serve as a low-level debugging aid.

Walkback and *Debugger* give views of the state of your program at the point of an error. They are high-level debugging aids.

Method Browsers let you browse and edit a list of methods.

Browsers use indexes to access information. Often the information is organized in a hierarchical manner. A browser is a window consisting of at least two panes, a *list* pane and a *text* pane. Selecting from the list displays related information in the text pane. This text may be modified, and eventually saved.

Disk Browser

The Disk Browser lets you browse the files on a disk device. To open a Disk Browser, select **browse disk** on the system menu. **Smalltalk/V** responds with a menu of available disk devices. Select the drive whose directory you want to see. **Smalltalk/V** then opens the new Disk Browser. You can have as many Disk Browsers open at a time as you want.

The Disk Browser is divided into four panes, as shown in Figure 16.1.

The *directory hierarchy* list appears in the upper-left pane. In this pane, the names of all the directories on the disk are listed. You will see a backslash symbol (\) in the top corner of the pane. This symbol stands for the root or parent dictionary of the entire disk. The names of the directories in the parent directory are indented and appear in hierarchical order. Initially only the first level of subdirectories are displayed and an ellipsis (...) is used to indicate that there may be subdirectories. By selecting **hide/show** from the pane menu or double-clicking on a directory name, the full subdirectory hierarchy will be shown.

The *file* list is in the pane to the right of the directory hierarchy list. When a directory is selected in the directory hierarchy list pane, its files will appear in this pane.

The *contents* pane shows the text of a selected file or further directory information if no file has been selected.

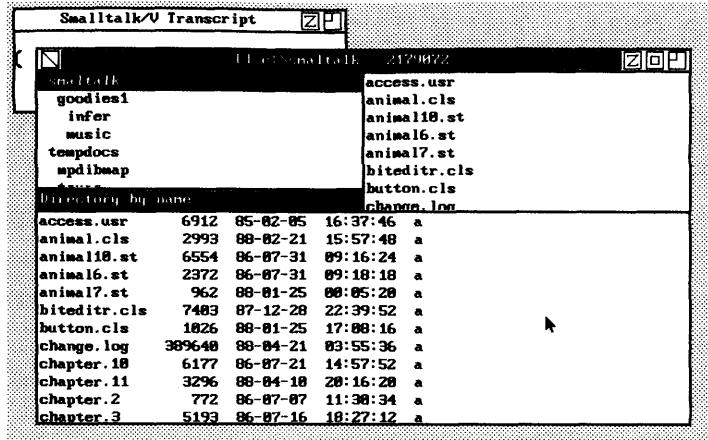


Figure 16.1
Disk Browser

The *directory order* pane contains a statement telling you how the information in a directory is ordered when it is displayed in the contents pane. The pane menu associated with the directory order pane gives you the choice of ordering the directory by date, name, or size.

The label bar of the Disk Browser displays other useful information. If the disk being browsed is labeled, the label is displayed between the brackets, [and]. The full path name of the currently selected directory in the browser is displayed. If no directory is selected, then only the device drive character appears. The numbers at the right side of the label represent the amount of free space on the disk. This number is automatically updated as you use the browser.

Browsing Directories

To *select a directory* to browse, move the cursor into the directory list pane and press the *select* key when the cursor is on top of the directory you wish to browse. **Smalltalk/V** displays a list of file names in the file list pane (the right-hand pane) and a complete listing of the directory information in the contents pane (the bottom pane). You can move the cursor or scroll the directory list pane and select a different directory any time you wish.

The directory hierarchy list menu has four selections:

Remove eliminates a subdirectory from the disk. You will get an error if there are files in the subdirectory

Update allows you to browse another disk in the same drive. Selecting this function tells the window to update itself by rereading the directory structure on the disk. You should select this function only after you have inserted the second disk.

Hide/Show expands the subdirectory hierarchy if the selected directory has an ellipsis (...) at the end, or hides the subdirectories if they are displayed. Note that the same effect can be achieved by merely clicking on a selected directory name.

Create makes a new directory as a subdirectory of the selected directory. When you select this function, a prompter appears, asking for the path (directory) name.

Browsing Files

To *select a file* to browse, move the cursor on top of the file name in the *file list* pane and press the *select* key. **Smalltalk/V** responds by displaying the file contents in the *contents* pane. You can select another file to browse by selecting it in the *file list pane*.

The menu for the *file list* pane displays a variety of functions for files. The *file list* pane is the top-right pane of the Disk Browser.

Remove eliminates the selected file and its contents.

Print causes the selected file to be printed.

Mode permits you to change the system file mode bits (see your PC or MS-DOS manual) of the selected file.

Rename lets you change a selected file's name.

Copy duplicates a selected file in another location.

Create generates an empty new file.

Editing Contents

Figure 16.2 shows the three menus for the *file contents* pane, the pane at the bottom of the window. Which menu appears is dependent upon what you select in the other panes.

If you are looking at a directory, you can pop up the menu shown at center. The menu at left can be brought up only if part of a large file has been read into the pane. You can pop up the menu on the right if you have selected a smaller file. All of these menus are popped up by pressing the *pane menu* key (**Smalltalk/V** knows which one to use) or the right mouse button.

As you can see, the three menus share some common functions. The *copy*, *cut*, and *paste* functions are the same as those used in normal text editing (described in Chapter 15). In addition, the *do it* and *show it* functions are identical to those previously described for Smalltalk expression evaluation. *Save* and *restore* have been explained under text editing as well.

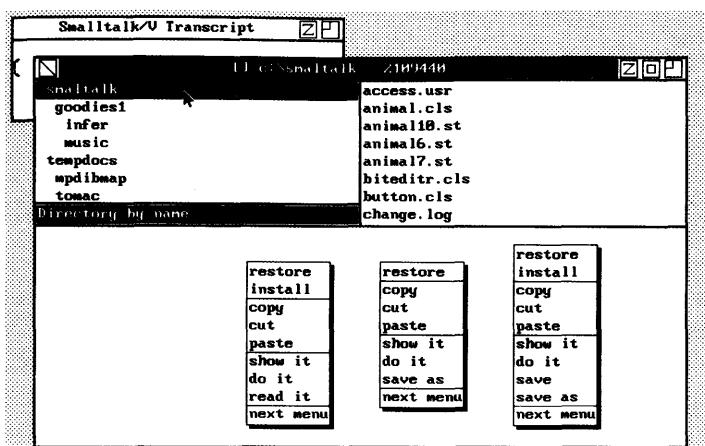


Figure 16.2
File Contents
Pane Menus

The *save as* function allows the directory information to be written on a file. You can also edit the information in the contents pane (for example, adding file comments) before writing it on the file.

The other functions of the file contents pane menus are the following:

Read it is used to read the entire contents of a large file. Normally, when the text of a file exceeds 10,000 bytes, the pane will only display the first 2,000 bytes, followed by the final 8,000 bytes. Any editing changes performed before *read it* is selected are lost. After you select this function, the entire file is read into the pane. The pane menu will change to the one for small files (one on the right). Now you can perform the *save* functions.

Save as is used to save edited material in a file different from the selected one (*save* rewrites the selected file with the text in the pane). Selecting this function causes a Prompter to appear which asks for the file name. You must respond with the name of a new or existing file.

Install compiles the selected text into the system. The selected text must be in the same format as the text in the *change log* or text in a *class definition* file. It is used to evaluate Smalltalk expressions and to compile and install Smalltalk source code from files. Typically, these would be from the *change log* or from a file containing a class definition. This use of the *change log* is described in more detail in Chapter 15. Remember that the code you want to install must be in *change log* format.

Viewing Directory Contents

The directory order pane is located immediately below the directory list pane. It allows you to see the full information associated with each file in a directory, as well as to control the order in which this information is displayed.

To display the directory in the contents pane, move the cursor into the directory order pane and press the select key. Smalltalk/V responds by displaying in the contents pane information about the files in the currently selected directory (the one selected in the directory list pane).

The directory order pane menu contains selections to determine how to sort the directory listing. To display the files in the directory in a different order, pop up the pane menu of the directory order pane by pressing the pane menu key. You now have a choice of ordering by date, name, or size. After you choose, Smalltalk/V will reorder the files.

Class Hierarchy Browser

The Class Hierarchy Browser lets you examine the interrelationships of the classes within Smalltalk/V, and to edit their contents.

Opening a Class Hierarchy Browser

Select the *browse classes* function of the *system* menu to open a Class Hierarchy Browser.

The Class Hierarchy Browser is divided into five panes as shown in Figure 16.3.

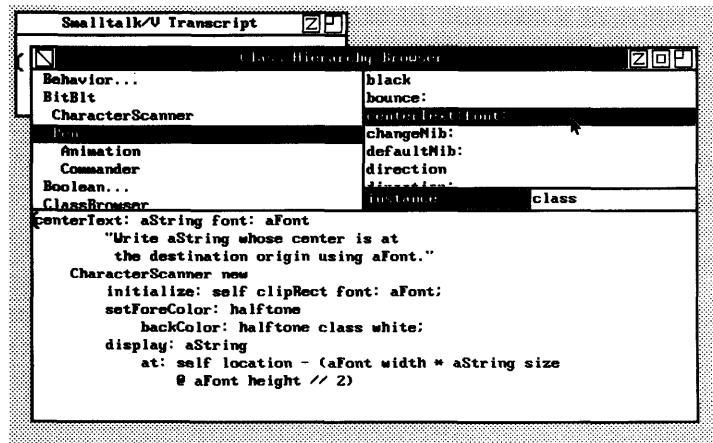


Figure 16.3
Class Hierarchy
Browser

The *class hierarchy list* appears in the upper-left pane. In this pane, the names of all of the classes in the system are presented in a hierarchical order. Notice that *Object* appears first in the list. *Class Object* is uppermost in the hierarchy. All other classes are subclasses to *Object* and therefore appear indented. Subclasses of these classes are indented further.

The *method list* pane is the pane to the right of the class hierarchy list. In this pane, either a list of the *instance* methods or a list of the *class* methods is presented. The two small panes underneath the method list pane are used to select the desired list. You can choose one by moving the cursor over either the class pane or the instance pane and then pressing the *select* key or the left mouse button. There is no pane menu associated with the class or instance pane. If you press the *pane menu* key, Smalltalk/V will pop up the same menu that the *window menu* key pops up.

The Smalltalk code itself is seen in the *contents* pane which occupies the bottom half of the window. When you select a class in the hierarchy pane, the message which defines the class selected is displayed in this pane. Select a method and its code appears. The contents pane is a text pane so it can be edited.

Browsing the Hierarchy

To *select a class to browse*, move the cursor into the class hierarchy list pane, the top-left pane of the window, and select a class by pressing the *select* key. Smalltalk/V displays the class definition in the contents pane and a list of methods implemented by the class in the methods list pane. You can scroll the class hierarchy list pane, like all list panes, with the vertical scrolling keys, Pg Up and Pg Dn, or the mouse.

The class hierarchy list pane menu contains functions which permit you to write the definition of a class and all of its methods on a file, browse a particular class, add a subclass and remove a class.

File out writes the class definition along with all of the instance and class messages of the selected class to a file. The file is in a special *change log* format (see *Automatic Logging of Changes* in Chapter 15 for more information). Smalltalk/V derives the file name from the class name with the extension .cls and places it in the directory from which Smalltalk/V was invoked. Subclasses are not automatically filed out. The *file out* function does not affect the class in the system.

Update tells the Class Hierarchy Browser to recompute the class hierarchy list. You must do this if you have removed a class or added a class using another browser.

Hide|show lets you hide or show the subclasses of the selected class. It is useful for shortening the list of classes by hiding the ones not currently of interest. If the subclasses of a class are hidden, an ellipsis (...) appears after the class name. Note that the same effect can be achieved by simply double-clicking on the desired class.

Add subclass permits you to add a subclass to the selected class.

Remove class lets you remove the selected class.

Adding Classes

To add a subclass of a class, first select in the hierarchy list pane the class that will be the superclass of the new class. Now pop up the pane menu and select the add subclass function. A prompter with the selected class name in the label then asks for a subclass name.

When you reply to the prompter, a menu then appears asking you what type of subclass you want. The subclass type depends upon whether the objects belonging to the class are to contain named instance variables, indexed instance variables, or byte arrays. After you make a choice, the new subclass is built and the class hierarchy list is automatically updated. This subclass is already selected in order to allow you to define its instance and class variables, add methods to it, or define subclasses of it.

Defining Classes

To define the instance variables, class variables, and pool dictionaries of a class, you must first select the class in the class hierarchy list pane. **Smalltalk/V** responds by displaying the current class definition in the contents pane. The definition includes the class's superclass, instance variables, class variables, and pool dictionaries. You change the class definition by editing the text in the contents pane and then selecting the save function from the contents pane menu.

When you change a class definition, **Smalltalk/V** automatically recompiles all of the methods in the class and all of its subclasses. In addition, a message is written on the system change log giving the new class definition.

Changes to a class definition take effect immediately so that all future instances of the class will have the new structure. You should therefore be very careful when modifying classes that are used by the **Smalltalk/V** environment because you are changing the environment. Until you are confident with these procedures, you should define subclasses as opposed to modifying the structure of existing classes.

Removing Classes

To remove a class, first select in the hierarchy list pane the class to be removed. Now pop up the pane menu and select remove class. The system will prompt you for confirmation.

All of the methods in the class are automatically removed when the class is removed. **Smalltalk/V** will not let you remove a class if it has subclasses or if there are instances of the class anywhere in the environment. If either of these exists, you will get a walkback window explaining the problem. Remember that **Smalltalk** automatically removes

(collects as garbage) any object that is not referred to by some other object in the system. To remove all instances of a class, you must change all references to instances of the class to refer to something else.

You can send the message **allReferences** to an object to find out all objects that refer to the receiver object.

Browsing the Methods

The method list pane is the top-right pane of the window. Remember that there are two kinds of methods in a class, **instance** methods and **class** methods. The list that appears is determined by the two panes directly beneath the method list pane, the class and the instance panes. You select the list by moving the cursor over the class or instance pane and then pressing the **select** key or the right mouse button.

To *select a method*, move the cursor into the methods list pane and press the **select** key when the cursor is on top of the desired message selector you wish to select. **Smalltalk/V** responds by placing the source code for the method that implements the message in the contents pane below. You can move the cursor or scroll the methods list pane and then select a different method by pressing the **select** key or the right mouse button.

The method list pane menu contains four functions:

To *remove a method* from the selected class, select the method, pop up the method list menu, and select the **remove** function. The method list is immediately updated.

To *add a new method*, you can use the **new method** function. A template that you can edit appears. You can also edit the text of any existing method in the class. The name of the new method is taken from the first line of the text. In either case, you use the **save** function to invoke the compiler and install the new method. Remember that there are two kinds of methods—class and instance methods. The kind of method you get is determined by whether you have currently selected the class or instance pane.

If a compilation error is detected, you will see a message in your source in the pane at the point where the error was detected. The error message is selected (reversed). Edit the text and use the **save** function again to recompile.

When the method is successfully compiled with no errors, **Smalltalk/V** installs it into the class automatically and all future invocations of this method use this new version. The source code of the new or modified method is written onto the **change log** file.

Senders causes **Smalltalk/V** to search all of the methods in the environment for senders (callers) of the selected message selector. A **senders** window, which is a kind of Method Browser, pops up to display each method (and its class) that sends a message with the selected message selector.

Implementors causes Smalltalk/V to search all of the classes in the environment for implementors (definers) of the selected message selector. An *implementors* window, which is a kind of Method Browser, pops up to display the names of all the classes that implement methods for the selected message selector.

Modifying the Source Code for a Method

To see the source code of a different method, move the cursor into the methods list pane and select another message selector. The contents pane is a text pane used for modifying and adding methods.

All of the functions on the contents pane menu behave exactly as described under **Text Editor** in Chapter 15.

To *modify* an existing method, you must first select it in the method list pane. The source currently being used by the system is displayed in the contents pane. Edit the source code in this pane. After editing the source text, you then use the **save** function on the contents pane menu to invoke the compiler and install the new method in the environment.

Class Browser

A Class Browser can be opened in either of two ways. You can send the message **edit** to any class or you can select the **browse** function from the hierarchy pane menu of the Class Hierarchy Browser described previously in this chapter.

Move the cursor into any text pane. Type in the name of the class you want to browse followed by **edit** and select it. Select the **do it** function from the pane menu. Smalltalk/V then opens a Class Browser for the class.

Figure 16.4 shows a Class Browser for the class **DemoClass**. Class Browsers have three panes: the *dictionary list* pane, the *method list* pane, and the *contents* pane.

The label of a Class Browser identifies the class you are browsing. In this window, only the methods for the selected class can be browsed, added, or modified.

The dictionary list pane is the top-left pane that contains the two choices: **class** and **instance**. If you select **instance**, then the list of instance methods is displayed in the method list pane immediately below this pane. If you select **class**, then the list of class methods is displayed in the method list pane. There is no pane menu associated with the dictionary list pane. If you press the **pane menu** key, Smalltalk/V will pop up the same menu that the **window menu** key pops up.

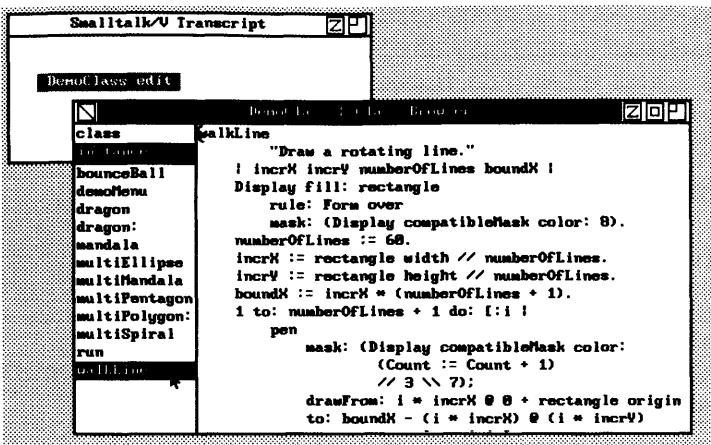


Figure 16.4
Class Browser

The method list pane is immediately below the dictionary list pane. It displays the list of **class** or **instance** methods implemented by the class depending on what you choose (class or instance) in the dictionary list pane. This pane lets you choose a particular method implementation for viewing.

The contents pane is the large pane to the right of the two list panes. It displays the currently selected method in the method list pane. You can edit and recompile methods in this text pane.

Browsing Method Lists

The method list pane displays a list of the message selectors for either the **class** methods or the **instance** methods defined in the class. The choice you make in the dictionary list pane (immediately above this pane) determines which list you see.

The functions on the methods list pane let you remove methods and display the implementors (definers) and the senders (callers) of messages. Note that this is very similar to the method list pane described for the Class Hierarchy Browser. The items appearing in both menus function in the same manner. See **Browsing the Methods** for the Class Hierarchy Browser in this chapter for a description of how to select a method and for explanations of the menu functions.

Modifying and Adding Methods

The contents pane is a text pane used to modify and add methods to the class being browsed. All of these functions behave exactly as described under **Text Editor**.

The process of adding and modifying methods is the same as described for the Class Hierarchy Browser. The Class Browser, however, uses a dictionary list pane while the Class Hierarchy Browser has a class pane and an instance pane.

The Inspector

Figure 16.5 shows an *inspector* window. Inspectors are used to examine and change objects in the system. They are a low-level debugging aid.

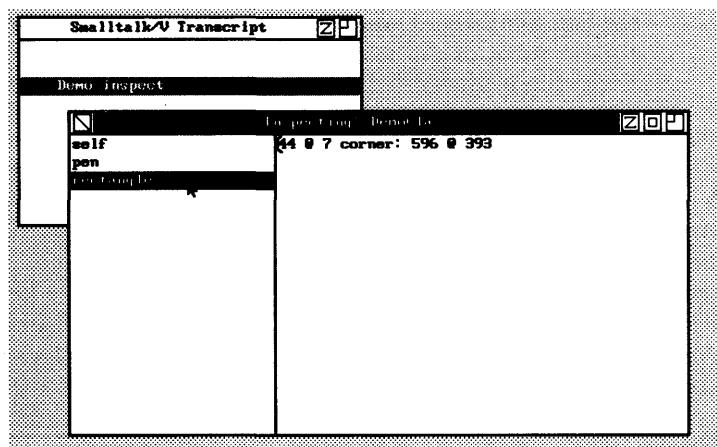


Figure 16.5
Inspector Window

To open an Inspector, send the message *inspect* to any object. For example, to open an inspector on the global variable *Demo*, evaluate the following text:

```
Demo inspect.
```

Inspectors have two panes. The list pane on the left is the *instance variable list* pane. The pane on the right is the *instance variable contents* pane.

The instance variable list pane shows all of the instance variables of the object being inspected. The first item in the list is always the name *self*, which is the object being inspected. The named instance variables, if any, are listed next. If the object being inspected has indexed instance variables, then they are listed last with numerical indices.

When you select one of the variables from the list, its current value is displayed in the instance variable contents pane. If you select *self*, the current value of the inspected object is displayed.

The pane menu of the list pane has only one function, *inspect*. When you select this function, a new Inspector is opened on the currently selected instance variable. Also, if you click on a selected item, an Inspector window is opened on that item.

The instance variable contents pane is a text pane. The pane menu has the normal text pane functions on it. You can use this pane to evaluate any type of expression that you wish. There are two very important features of this pane:

1. Any expression that you evaluate is compiled in the scope of the object being inspected. This means that you can use the names of all of the instance variables in your expressions.
2. If you select the **save** function from the pane menu, the entire contents of the text pane will be compiled and evaluated. The result of this expression will replace the current value of the selected instance variable in the list pane to the left. Similarly, if you select the **restore** function, Smalltalk/V places the current value of the selected instance variable into this text pane.

Inspecting Dictionaries

A *Dictionary* is an object which contains associations between keys and values. The Inspector created for objects which are dictionaries are special. The Dictionary Inspector has two panes: an *instance variable list* pane and an *instance variable contents* pane, just like regular Inspectors. Dictionary Inspectors, however, list the keys in the dictionary in the instance variable list pane, rather than instance variable names and indices. When you choose a key in the pane, the associated value is displayed in the instance variable contents pane. Notice that **self** is not on the list pane of a dictionary inspector.

The dictionary inspector pane menu has three selections. Notice that there are two functions not present on a normal inspector's pane menu.

Remove allows you to remove the association for a selected key from the Dictionary.

Inspect opens an Inspector on the value associated with the currently selected key.

Add allows you to add a new element to the dictionary. The system pops up a Prompter asking for the new key. The associated value is nil until you select the new key, change the nil value in the contents pane, and save it.

Debugger Windows

There are two debugger windows: a *walkback* window and a *debug* window. A walkback window pops up automatically when errors are detected. When you need more information than provided in the walkback window, you explicitly request a debug window using the walkback window menu.

Walkback Window

You request a walkback window by sending the **error:** message to any object with a string describing the error as argument. For example,

```
self error: 'Index is outside of collection bounds'
```

Smalltalk/V also places a walkback window on the screen when any of the following occur.

- The message **halt** is sent to any object, for example:

```
self halt
```

- The break key is typed at the same time that the control key is pressed, causing a **control-break** interrupt.
- The message-send nesting gets too deep, resulting in a stack overflow condition.

Figure 16.6 shows a walkback window produced by evaluating the expression in the System Transcript. This expression attempts to access the eighth character in the string 'hello', an obvious error.

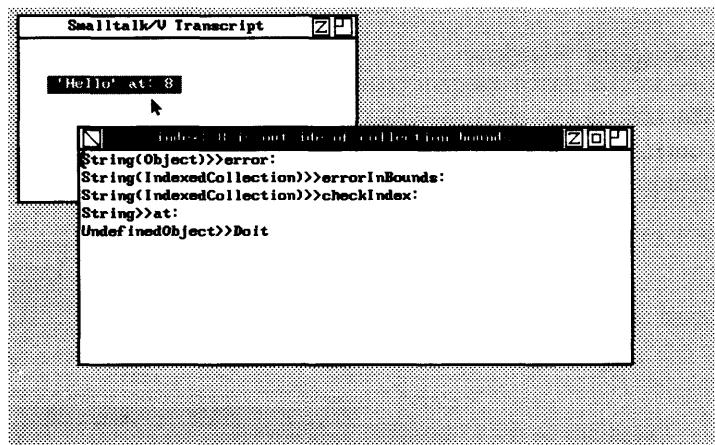


Figure 16.6
Walkback Window

The label of the window describes the error. The text pane of the window contains a *method walkback* showing the incomplete message sends that led to the error. Each line in the text pane represents a single message send, with the most recent send listed first. On each line, the class of the receiver is given first. If the method used is defined in a superclass of the receiver, the class in which the method is defined appears next in parentheses. The string on the right following ">>" is the message selector.

Sometimes you will see lines of the form:

[] in ClassName >> methodName

This indicates that an error occurred during the evaluation of a block in the method `methodName` of the class `ClassName`.

Whenever you get a walkback window, you generally do one of three things:

1. You can determine what the problem is from the information contained in the walkback window. In this case, you normally close the walkback window and then go fix the problem.
2. You can resume execution at the point of interruption, provided that the walkback window occurred either as a result of a **control-break** interrupt, or because a **halt** message was sent. In these cases, there is nothing wrong with the program, so you can pop up the pane menu for the walkback window and select **resume**. The walkback window closes and execution continues.
3. You can decide that you need more information, and would like to use the **debugger** to obtain it. In this case, you pop up the pane menu for the walkback window and select **debug**. The walkback window closes and you are prompted for the corners of the debugger window:

Debugger Window

The debugger window gives you an expanded view of the walkback and allows controlled execution of a process. The window has six panes. Figure 16.7 shows a debugger window.

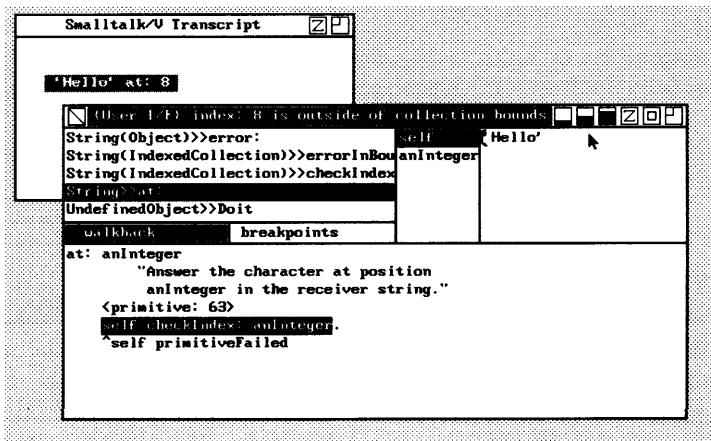


Figure 16.7
Debugger Window

The top left list pane serves two purposes: presenting a walkback and listing breakpoints. If the pane labeled **walkback** is selected, the pane above has the same walkback list that appears in a walkback window. When you select a walkback line, the other panes contain related information.

If the pane labeled **breakpoints** is selected, the pane above lists the class name and method selector for all methods which have breakpoints set. When you select a breakpoint line, the pane below displays the source code for the selected method. Setting a breakpoint in a method causes execution to stop when the method is entered, provided that execution is under control of the debugger (see the description of the **hop**, **skip** and **jump** buttons below).

The bottom pane displays the source code for the selected method and, if **walkback** is selected, the source code for the currently executing statement is highlighted. This pane also serves as a text editor so you can change the code as you do with the Class Hierarchy Browser. You update a method by selecting the **save** entry on the pane menu. If a selected walkback method is updated in this way, all walkback entries above the bottom-most occurrence of the method in the walkback list are discarded, because a method they would return to has changed.

The two panes on the top right serve as an inspector for the receiver, arguments and temporary variables of the selected method. The variable name pane on the left of the two contains **self**, representing the receiver, and the names of all arguments and temporary variables. The variable pane on the right displays the value of the selected variable.

The menu of the variable name pane contains a single entry: **inspect** which allows you to spawn another inspector on the selected object. For example, you can inspect the instance variables of the receiver by double-clicking on **self**.

The menu of the variable value pane is a standard text editing menu. When you select **save**, the value of the selected variable is changed to contain the results of evaluating the expression in the value pane.

The label bar of the debugger window contains three buttons related to debugging: **hop**, **skip** and **jump**. These work as follows:

- **hop** - executes the next expression in the debugged process.
- **skip** - executes the next expression in the selected method or up to the next breakpoint, whichever comes first. Note that **skip** may execute several expressions in lower-level methods.
- **jump** - executes up to the next breakpoint.

The menu of the walkback list pane contains the entries **resume**, **restart**, **senders**, **implementors**, **add breakpoint** and **remove breakpoint**, defined as follows:

- **resume** - as in the walkback window, allows you to continue execution after a **halt** message or a **control-break** interrupt. The debugger window disappears and execution continues from the point of interruption. You will not be allowed to resume, however, if you have changed a method in the walkback list.
- **restart** - if a method is selected, the debugger window disappears and execution is restarted by re-sending the message in the selected walkback entry.
- **senders** - as in the Class Hierarchy Browser, a Method Browser window is popped up listing all methods in Smalltalk/V which send the message corresponding to the selected method.
- **implementors** - as in the Class Hierarchy Browser, a Method Browser window is popped up listing all methods in Smalltalk/V which implement a method with the same name as the selected method.
- **add breakpoint** - two prompter windows appear, requesting the class and selector of the method to add to the breakpoint list.
- **remove breakpoint** - the selected breakpoint entry is removed from the breakpoint list.

Method Browser

The Method Browser lets you browse and edit a list of methods. There are two ways to open a Method Browser. Selecting **senders** in any of the list pane menus (e.g., the method list pane menu of the Class Hierarchy Browser) will open a Method Browser on the list of all methods in the system that send the selected message selector. Selecting **implementors** in any of the list pane menus will open a Method Browser on the list of all methods that implement the selected message selector. These two windows are often referred to as the *senders* window and the *implementors* window respectively. Figure 16.8 shows a Method Browser on all senders of the message **open**.

Method Browsers have two panes. The list pane on the top is the *method list* pane. the pane on the bottom is the *text* pane.

The method list pane shows a list of methods identified by the class and message selector. When you select a method in the list, the source code for the method is displayed in the text pane. The pane menu of the list pane has two functions:

senders causes Smalltalk/V to search all of the methods in the environment for methods that send (call) the selected message selector. A new Method Browser is opened on the methods found.

implementors causes Smalltalk/V to search all of the methods in the environment for methods that implement (define) the selected message selector. A new Method Browser is opened on the methods found.

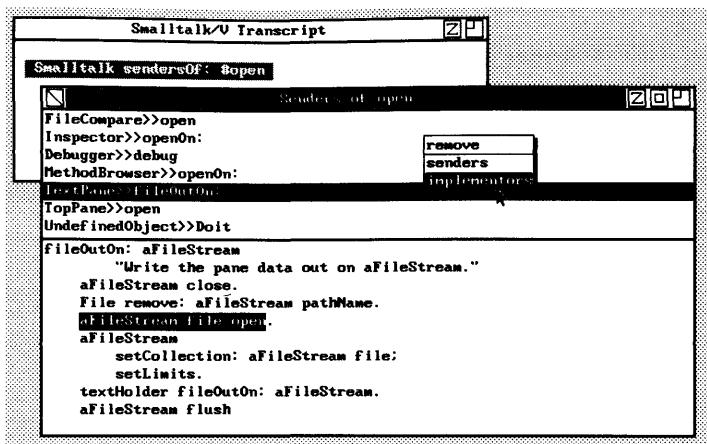


Figure 16.8
Method Browser

The text pane lets you view and edit the source code for a selected method. This pane has the normal text pane editing menu. When you save the edited text, the method is recompiled.

Part 4

Encyclopedia of Classes

Animation

An Animation contains a collection of pens representing objects being animated. Each pen is moved by erasing its image from the old location and then displaying the image in a new location. The message interface is similar to class Pen except that a name must be given to identify the object to move. Each pen in the collection behaves differently than a non-animation pen, for example, the instance variable sourceForm of each pen contains an Array of Forms representing pictures in successive stages of a motion. These pictures are displayed in a cyclical fashion for each BitBlt copy. The overlapped objects are displayed in the order as they are entered into the Animation object. Thus an object entered first will appear to always move behind the object entered later.

Inherits From: Pen BitBlt Object

Inherited By: (None)

Named Instance Variables:

backForm

Contains a Form which is a copy of the background of the animation. A region of this form is pasted onto the underForm to carry out the erasing operation.

clipHeight

(From class BitBlt)

clipWidth

(From class BitBlt)

clipX

(From class BitBlt)

clipY

(From class BitBlt)

curPen

Contains a Pen which is one of the animated objects that is currently moving.

destForm

(From class BitBlt)

destX

(From class BitBlt)

destY

(From class BitBlt)

direction

(From class Pen)

downState

(From class Pen)

fractionX

(From class Pen)

fractionY

(From class Pen)

halftone

(From class BitBlt)

height

(From class BitBlt)

hideBlt

Contains a BitBlt which is used to erase the old image.

pens

Contains an OrderedCollection of pens with information about each animated object.

rule

(From class BitBlt)

shiftRate

Contains an Integer specifying the number of times the same picture will be used before shifting to the next picture of an animated object. For example, a spinning ball will appear to be spinning more slowly when its shiftRate is larger.

sourceForm

(From class BitBlt)

sourceX

(From class BitBlt)

sourceY

(From class BitBlt)

speed

Contains an Integer specifying the number of pixels to skip between successive moves. Skipping a larger number of pixels gives the illusion of moving at a higher speed.

underForm

Contains a Form used as an intermediate place for merging the erasing rectangle and the new displaying rectangle so that the blinking effect is reduced.

width

(From class BitBlt)

Class Variables:**DoubleCenter**

(From class Pen)

Pool Dictionaries: (None)**Class Methods:** (None)**Instance Methods:****add: anArray name: aName color: aColor**

Add the forms in anArray to the receiver with the name aName and color aColor (integer from 0 to 15, or a halftone symbol).

display

Display the currently moving object and all the objects it overlaps with.

initialize: aRectangle

Initialize the receiver to do animation within aRectangle.

setBackground

Set the background form to the contents of the display screen.

shiftRate: anInteger

Specify how many times the current picture will be copied before shifting to the next one.

speed: anInteger

Change the distance between consecutive copies to anInteger.

tell: aName bounce: anInteger

Tell the pen with the name aName to bounce by a distance of anInteger.

tell: aName direction: anInteger

Tell the pen with the name aName to change its direction to anInteger number of degrees.

tell: aName go: anInteger

Tell the pen with the name aName to go for a distance of anInteger.

tell: name goto: aPoint

Tell the object with name to go to aPoint.

tell: aName place: aPoint

Tell the pen with the name aName to be placed at aPoint.

tell: aName turn: anInteger

Tell the pen with the name aName to turn by anInteger number of degrees.

Array

An Array is a collection of any objects accessed through a fixed range of integer indices (representing the positions of the elements within the Array). Most of the protocol to handle arrays is inherited from its superclasses. The only methods contained in this class are the methods to print and store an Array on a stream.

Inherits From: FixedSizeCollection IndexedCollection Collection Object

Inherited By: CompiledMethod

This class contains indexed instance variables.

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:**printOn:** aStream

Append the ASCII representation of the receiver to aStream.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reinstated.

Association

Class Association provides the means of associating two objects known as the key/value pair, and defines the protocol to manipulate them. Association objects are often used as the elements of class Dictionary.

Inherits From: Magnitude Object

Inherited By: (None)

Named Instance Variables:**key**

Contains the first object of the key/value pair. It is primarily used as a key to retrieve the second object (the value) of the association when dealing with instances of class Dictionary.

value

Contains the second object of the key/value pair.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:**key:** anObject

Answer an instance of class Association whose key is initialized to anObject.

key: aKey **value:** anObject

Answer an instance of class Association whose key is initialized to aKey and whose value is initialized to anObject.

Instance Methods:

< anAssociation

Answer true if the receiver key is less than anAssociation key, else answer false.

<= anAssociation

Answer true if the receiver key is less than or equal to anAssociation key, else answer false.

= anAssociation

Answer true if the receiver key is equal to anAssociation key, else answer false.

> anAssociation

Answer true if the receiver key is greater than anAssociation key, else answer false.

>= anAssociation

Answer true if the receiver key is greater than or equal to anAssociation key, else answer false.

hash

Answer the integer hash value for the key of the receiver.

key

Answer the key of the receiver.

key: anObject

Set the key of the receiver to be anObject. Answer the receiver.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reinstated.

value

Answer the value of the receiver.

value: anObject

Set the value of the receiver to be anObject. Answer the receiver.

Bag

A Bag is a collection of unordered elements in which duplicates are allowed. It cannot be accessed through external keys. Bags are useful for containing arbitrary objects and for counting occurrences of equal objects. Bags are hashed for rapid searching.

Inherits From: Collection Object

Inherited By: (None)

Named Instance Variables:

elements

Contains a Dictionary. For each key/value pair, the key contains an element of the bag and the associated value represents the number of occurrences of the element in the bag.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

new

Answer an empty Bag.

Instance Methods:

add: anObject

Answer anObject. Add anObject to the elements of the receiver.

add: anObject withOccurrences: anInteger

Answer anObject. Add anObject to the elements of the receiver anInteger number of times.

at: anInteger

Answer the element of the receiver at index position anInteger. Report an error since bags are not indexable.

at: anInteger put: anObject

Replace the element of the receiver at index position anInteger with anObject. Report an error, since bags are not indexable.

do: aBlock

For each element in the receiver, evaluate aBlock with that element as the argument.

includes: anObject

Answer true if the receiver contains an element equal to anObject, else answer false.

occurrencesOf: anObject

Answer the number of elements of the receiver equal to anObject.

remove: anObject ifAbsent: aBlock

Answer anObject. Remove one occurrence of anObject from the receiver collection. If anObject is not an element of the receiver, evaluate aBlock (with no arguments).

size

Answer the number of elements in the receiver collection.

Behavior

Class Behavior is the abstract class that defines and implements the common protocol for all the classes and metaclasses in Smalltalk. Behavior provides methods that support source code access, compilation, object creation, and class hierarchy access.

Inherits From: Object

Inherited By: Class MetaClass

Named Instance Variables:

comment

This is reserved for future use.

dictionaryArray

Contains an Array of method dictionaries, in message lookup order.

instances

Contains an Array of instance variable names defined by this class. The names are stored as strings.

name

Contains the Symbol that is the name of this class.

structure

Contains a SmallInteger that describes the physical structure of instances of this class. See the class variables of Behavior for the definition of the encoding.

subclasses

Contains an Array of all the subclasses of this class.

superClass

Contains the superclass of this class.

Class Variables:

InstIndexedBit

Contains a SmallInteger which, when logically ANDed with the instance variable structure, determines whether or not an instance of the class can have indexed instance variables. The value contained is 8192.

InstNumberMask

Contains a SmallInteger which, when logically ANDed with the instance variable structure, determines the number of named instance variables for each instance of the class. The value contained is 127.

InstPointerBit

Contains a SmallInteger which, when logically ANDed with the instance variable structure, determines whether or not an instance of the class contains object pointers in their instance variables. The value contained is 16384.

Pool Dictionaries: (None)

Class Methods: (All private)

Instance Methods:

addSelector: aSymbol withMethod: aCompiledMethod

Add aCompiledMethod to the receiver messageDictionary using aSymbol as the key. If aSymbol is not a Symbol report an error.

addSubclass: aClass

Add aClass to the subclasses of the receiver. Make the the receiver the superclass of aClass.

allClasses

Answer a Set of all of the classes contained in Smalltalk.

allClassVarNames

Answer a Set of strings of all of the class variable names defined in the receiver and its superclasses.

allInstances

Answer an Array of all of the instances of the receiver.

allInstVarNames

Answer an Array of strings of all of the instance variable names defined in the receiver and its superclasses.

allSubclasses

Answer an OrderedCollection of all the subclasses of the receiver in hierarchical order. Classes at the same hierarchical level are sorted alphabetically.

allSuperclasses

Answer an OrderedCollection of all the superclasses of the receiver. The superclasses are in inverse hierarchical order, i.e class Object is last.

canUnderstand: aSymbol

Answer true if the receiver or any of the receiver superclasses implement the method named aSymbol, else answer false.

classVariableString

Answer a String of all the class variable names defined by the receiver. The names are separated with blanks.

compile: codeString

Compile the Smalltalk method contained in codeString. The class to use for resolving variables is the receiver. If there are no errors, add the method to the receiver messageDictionary and answer the Association with the message selector as the key and the compiled method as the value. If there is an error, answer nil.

compile: codeString notifying: requestor

Compile the Smalltalk method contained in codeString. The class to use for resolving variables is the receiver. If there are no errors, add the method to the receiver messageDictionary and answer the Association with the message selector as the key and the compiled method as the value. If there is an error the requestor is sent a message by the compiler identifying the error and this method answers nil.

compiledMethodAt: aSymbol

Answer the compiled code of the method named aSymbol defined in the receiver.

compileLogic: codeString

Compile the Prolog method contained in codeString. The class to use for resolving variables is the receiver. If there are no errors, add the method to the receiver messageDictionary and answer the Association with the message selector as the key and the compiled method as the value. If there is an error, answer nil.

compileLogic: codeString notifying: requestor

Compile the Prolog method contained in codeString.

deepCopy

Answer a copy of the receiver with shallow copies of each instance variable. Because classes are unique (cannot be copied), answer the receiver.

implementorsOf: aSymbol

Answer a collection of methods of myself and my subclasses that implement aSymbol.

includesSelector: aSymbol

Answer true if the message dictionary of the receiver includes a method of name aSymbol, else answer false.

inheritsFrom: aClass

Answer true if receiver can inherit methods from aClass, else answer false.

instanceVariableString

Answer a String containing all the instance variable names defined by the receiver. The names are separated with spaces.

instSize

Answer the number of named instance variables contained in instances of the receiver.

instVarNames

Answer the array of instance variable names defined by the receiver.

isBits

Answer true if instances of the receiver contain 8 bit values instead of object pointers, else answer false.

isBytes

Answer true if instances of the receiver contain 8 bit byte values, else answer false.

isFixed

Answer true if instances of the receiver do not contain indexed instance variables, else answer false.

isPointers

Answer true if instances of the receiver contain object pointers instead of 8 bit values, else answer false.

isVariable

Answer true if instances of the receiver contain indexed instance variables, else answer false.

methodDictionary

Answer the dictionary of methods defined in the receiver.

methods

Answer an instance of ClassReader initialized for the receiver.

new

Answer an instance of the receiver. If the receiver is indexable, then allocate zero indexed instance variables. This method is frequently reimplemented as a class message in classes that need special initialization of their instances.

new: anInteger

Answer an instance of the receiver. Allocate anInteger number of indexed instance variables. If the receiver does not have indexed instance variables an error is reported. This method is frequently reimplemented as a class message in classes that need special initialization of their instances.

printOn: aStream

Print the name of the receiver on aStream.

removeSelector: aSymbol

Remove the method named aSymbol from the methods defined in the receiver.

selectors

Answer a Set of symbols of the names of the methods defined by the receiver.

sendersOf: aSymbol

Answer a collection of methods of myself and my subclasses that send aSymbol.

shallowCopy

Answer a copy of the receiver which shares the receiver instance variables. Because classes are unique (cannot be copied), answer the receiver.

sourceCodeAt: aSymbol

Answer a String of the source code for the method named aSymbol in the receiver.

structure

Answer the integer that describes the structure of instances of the receiver. Refer to the class variables of Behavior for a definition of this integer.

subclasses

Answer an Array of subclasses of the receiver.

superclass

Answer the superclass of the receiver.

withAllSubclasses

Answer an OrderedCollection of the receiver and all of its subclasses in hierarchical order.

BiColorForm

A BiColorForm is like a Form except that it contains two additional instance variables, foreColor and backColor. This enables the BiColorForm to take on two arbitrary colors rather than just black and white. The foreColor is associated with 1 bits and the backColor with 0 bits in the bitmap.

Inherits From:

Form DisplayMedium DisplayObject Object

Inherited By:

(None)

Named Instance Variables:**backColor**

Contains an integer representing the background color (for 0 bits in bitmap).

bits

(From class Form)

byteWidth

(From class Form)

deviceType

(From class Form)

foreColor

Contains an integer representing the foreground color (for 1 bits in bitmap).

height

(From class Form)

offset

(From class Form)

width

(From class Form)

Class Variables:**BlackMask**

(From class Form)

DarkGrayMask

(From class Form)

GrayMask

(From class Form)

LightGrayMask

(From class Form)

PrinterMode

(From class Form)

WhiteMask

(From class Form)

Pool Dictionaries: (None)**Class Methods:****black**

Answer a black mask form.

color: aColor

Answer a mask form with aColor as the foreground color and black as the background color.

darkGray

Answer a dark gray mask form.

foreColor: aColor backColor: bColor

Answer a mask form with foreground aColor and background bColor.

gray

Answer a gray mask form.

lightGray

Answer a light gray mask form.

new

Answer a new BiColorForm.

white

Answer a white mask form.

Instance Methods:

backColor

Answer the background color.

backColor: aColor

Set receiver's background color to aColor.

foreColor

Answer the foreground color.

foreColor: aColor

Set receiver's foreground color to aColor.

foreColor: aColor backColor: bColor

Set receiver's foreground color to aColor and background color to bColor.

fromDisplay: aRectangle

Copy aRectangle area of the display screen to receiver. Screen bits that are the receiver foreground color become 1, the rest become 0.

BitBlit

This class defines all the basic graphics operations. Its main function is to transfer bits from one area to another. This involves three forms: the source form, destination form, and mask form. The bits in the source form are first ANDed with the bits in the mask form (tiled if size is smaller than the source rectangle), and then merged into the destination form with a combination rule. The combination rule specifies a logical operation (e.g. AND, OR, XOR, etc.) as to how to combine a masked source bit with its corresponding destination bit. The areas involved in this bit transfer are specified by a source rectangle on the source form, a destination rectangle on the destination form, and a clipping rectangle also on the destination form. After aligning the source rectangle origin with the destination rectangle origin, the final affected area is the intersection of all three rectangles. The prebuilt mask forms and supported combination rules can be obtained by sending class messages to Form or BiColorForm (if colors other than black and white are desired).

Inherits From:

Object

Inherited By:

Animation CharacterScanner Commander Pen

Named Instance Variables:**clipHeight**

Contains the height of the clipping rectangle.

clipWidth

Contains the width of the clipping rectangle.

clipX

Contains the x coordinate of the clipping rectangle origin.

clipY

Contains the y coordinate of the clipping rectangle origin.

destForm

Contains the destination form for bit transfers.

destX

Contains the x coordinate of the destination rectangle origin.

destY

Contains the y coordinate of the destination rectangle origin.

halftone

Contains the mask form which provides the graytone effect.

height

Contains the height of the source (and destination) rectangle.

rule

Contains an integer denoting the combination rule.

sourceForm

Contains the source form for bit transfers.

sourceX

Contains the x coordinate of the source rectangle origin.

sourceY

Contains the y coordinate of the source rectangle origin.

width

Contains the width of the source (and destination rectangle).

Class Variables:

(None)

Pool Dictionaries:

(None)

Class Methods:**destForm: dForm sourceForm: sForm**

Answer a BitBlt with dForm and sForm as its destination and source Forms.

Instance Methods:**clipRect**

Answer the clipping rectangle of the receiver.

clipRect: aRectangle

Set the clipping rectangle of the receiver to aRectangle.

clipX

Answer the x coordinate of the clip rectangle.

clipY

Answer the y coordinate of the clip rectangle.

combinationRule: anInteger

Set the rule for combining the bits of the source and destination forms to anInteger.

copyBits

Copy the bits from the source to the destination form. Hide the cursor if it is within the area of the transfer.

destForm

Answer the destination form of the receiver.

destForm: aForm

Set the destination form of the receiver to aForm.

destForm: dForm sourceForm: sForm

Answer the receiver with dForm and sForm as its destination and source Forms.

destForm: destination

sourceForm: source halftone: mask combinationRule: combinationRule

destOrigin: destOrigin sourceOrigin: sourceOrigin extent: extent

clipRect: clipRect

Initialize all the instance variables of the receiver.

destOrigin: aPoint

Set the origin of the destination rectangle to aPoint.

destRect: aRectangle

Set the destination rectangle to aRectangle.

destX

Answer the x-coordinate of the destination origin.

destX: anInteger

Set the x-coordinate of the destination origin to anInteger.

destY

Answer the y-coordinate of the destination origin.

destY: anInteger

Set the y-coordinate of the destination origin to anInteger.

drawFrom: startPoint to: stopPoint

Draw a line from startPoint to stopPoint.

extent

Answer a point whose coordinates are the width and height of the transfer area.

extent: aPoint

Set the width and height of the transfer area to the coordinates of aPoint.

height

Answer the height of the transfer area.

height: anInteger

Set the height of the transfer area to anInteger.

mask

Answer the mask form which provides the halftone effect.

mask: mask

Set the mask (halftone) form to mask.

sourceForm

Answer the source form of the receiver.

sourceForm: aForm

Set the source form of the receiver to aForm.

sourceOrigin: aPoint

Set the origin of the source rectangle to aPoint.

sourceRect: aRectangle

Set the source rectangle to aRectangle.

sourceX

Answer the x-coordinate of the source origin.

sourceX: anInteger

Set the x-coordinate of the source origin to anInteger.

sourceY

Answer the y-coordinate of the source origin.

sourceY: anInteger

Set the y-coordinate of the source origin to anInteger.

width

Answer the width of the transfer area.

width: anInteger

Set the width of the transfer area to anInteger.

Bitmap

A Bitmap is a fixed size indexable sequence of integers in the range 0 through 255. The elements of a Bitmap are efficiently packed into memory, one per byte. They represent bit maps of forms.

Inherits From: FixedSizeCollection IndexedCollection Collection Object

Inherited By: (None)

This class contains indexed byte values.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

atAllPut: aByte

Replace the bytes of the receiver with aByte. Answer aByte.

replaceFrom: start

to: stop with: aString startingAt: repStart

Replace the bytes of the receiver at index positions start through stop with consecutive bytes of aString beginning at index position repStart. Answer the receiver.

replaceFrom: start to: stop withObject: aByte

Replace the bytes of the receiver at index positions start through stop with aByte. Answer aByte.

Boolean

Class Boolean is an abstract class which defines the common protocol for logical values. The logical values are represented by its two subclasses, True and False.

Inherits From: Object

Inherited By: False True

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (All private)

Instance Methods:

deepCopy

Answer a copy of the receiver with shallow copies of each instance variable. Because there is only one true and one false, answer the receiver.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

shallowCopy

Answer a copy of the receiver which shares the receiver instance variables. Because there is only one true and one false, answer the receiver.

storeOn: aStream

Answer the receiver. Append the character sequence of the receiver to aStream from which the receiver can be reconstructed.

ByteArray

A ByteArray is a fixed size indexable sequence of integers in the range 0 through 255. The elements of a ByteArray are efficiently packed into memory, one per byte.

Inherits From: FixedSizeCollection IndexedCollection Collection Object

Inherited By: FileHandle

This class contains indexed byte values.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

replaceFrom: start**to: stop with: aCollection startingAt: repStart**

Replace the elements of the receiver at index positions start through stop with consecutive elements of aCollection beginning at index position repStart. Answer the receiver.

Character

Class Character defines the protocol for all the characters in the system (ASCII codes from 0 to 255). Instances of this class are immutable, meaning that they cannot be removed and new ones cannot be created. There is one and only one instance of each character in Smalltalk.

Inherits From: Magnitude Object

Inherited By: (None)

Named Instance Variables:

asciiInteger

Contains the ASCII encoding of the character.

Class Variables:

(None)

Pool Dictionaries:**CharacterConstants**

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:**digitValue: anInteger**

Answer the character representation of the digit anInteger.

new

Disallow the instantiation of characters because characters are immutable.

value: anInteger

Answer the character whose ASCII encoding matches anInteger.

Instance Methods:**< aChararacter**

Answer true if the receiver ASCII value is less than the ASCII value of aChararacter, else answer false.

<= aChararacter

Answer true if the receiver ASCII value is less than or equal to the ASCII value of aChararacter, else answer false.

= aChararacter

Answer true if the receiver ASCII value is equal to the ASCII value of aChararacter, else answer false.

> aChararacter

Answer true if the receiver ASCII value is greater than the ASCII value of aChararacter, else answer false.

>= aChararacter

Answer true if the receiver ASCII value is greater than or equal to the ASCII value of aChararacter, else answer false.

asciiValue

Answer the number corresponding to the ASCII encoding of the receiver.

asLowerCase

Answer the lower case value of the receiver if it is a letter, else answer the receiver.

asUpperCase

Answer the upper case value of the receiver if it is a letter, else answer the receiver.

deepCopy

Answer a copy of the receiver with shallow copies of each instance variable. Because characters are immutable (cannot instantiate a copy), answer the receiver.

digitValue

Answer a number corresponding to the digit value of the receiver.

hash

Answer the integer hash.

isAlphaNumeric

Answer true if the receiver is in the range of characters from 0 to 9 or in the range from a to z or in the range from A to Z, else answer false.

isDigit

Answer true if the receiver is in the range of characters from 0 to 9, else answer false.

isLetter

Answer true if the receiver is in the range of characters from a and z or in the range from A and Z, else answer false.

isLowerCase

Answer true if the receiver is in the range of characters from a to z, else answer false.

isSeparator

Answer true if the receiver character is either a space, tab, carriage-return, line-feed or form-feed character, else answer false.

isUpperCase

Answer true if the receiver is in the range of character from A to Z, else answer false.

isVowel

Answer true if the receiver is any one of the characters a,A,e,E,i,I,o,O,u,U, else answer false.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

shallowCopy

Answer a copy of the receiver which shares the receiver instance variables. Because characters are immutable (cannot instantiate a copy), answer the receiver.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reconstructed.

CharacterScanner

The function of this class is to convert characters represented by ASCII values into displayable bit patterns. It contains a font describing the bit patterns of all characters and carries out the conversion operation by transferring bits from the font (as the source form) to the destination form.

Inherits From:

BitBlt Object

Inherited By: (None)

Named Instance Variables:

backColor

Contains a mask form whose contents are used as the color of the background of characters.

blankBitBlt

Contains a BitBlt used to blank a designated area.

clipHeight

(From class BitBlt)

clipWidth

(From class BitBlt)

clipX

(From class BitBlt)

clipY

(From class BitBlt)

destForm

(From class BitBlt)

destX

(From class BitBlt)

destY

(From class BitBlt)

font

Contains the font used for displaying characters.

foreColor

Contains a mask form whose contents are used as the color of the characters.

frame

Contains a Rectangle limiting the area on the destination form to receive the converted bit patterns. Normally the clipping rectangle is the same as this frame but sometimes can be made smaller for special purposes and then restored to be the same as the frame.

halftone

(From class BitBlt)

height

(From class BitBlt)

rule

(From class BitBlt)

sourceForm

(From class BitBlt)

sourceX

(From class BitBlt)

sourceY

(From class BitBlt)

textEnd

Contains an integer specifying the ending character in the *textString* to be displayed.

textPos

Contains an integer specifying the starting character in the `textString` to be displayed.

textString

Contains a String of characters to be displayed on the destination form.

width

(From class BitBlt)

Class Variables:

(None)

Pool Dictionaries:

(None)

Class Methods:

(None)

Instance Methods:

blank: aPoint width: anInteger

Paint to the background color, the rectangle whose origin is `aPoint`, `width` is `anInteger`, and height is the font height.

blankRestFrom: anInteger

Blank the bottom portion of the frame starting from `anInteger` row.

clipRect: aRectangle

Set the clipping rectangle of the receiver.

display: aString at: aPoint

Display the bit pattern of `aString` at `aPoint` in the frame of the receiver.

display: aString from: anInteger at: aPoint

Display the bit pattern of `aString` starting at index position `anInteger` up to the last character of the string at `aPoint` in the frame. The remaining line after the last character will be blanked.

display: aString**from: start to: stop at: aPoint**

Display the bit pattern of `aString` from index position `start` to `stop` at `aPoint` in the frame of the receiver.

displayAll: aCollection**from: firstLine to: lastLine at: columnIndex**

Display the part of `aCollection` between `firstLine` and `lastLine`.

displayForm: aForm**at: aPoint rule: aRule**

Display `aForm` at `aPoint` in the frame using `aRule`.

font

Answer the current font used by the receiver.

frame

Answer the framing rectangle of the receiver. Usually it is the same as the clipping rectangle while the latter is sometimes changed to a smaller rectangle.

gray: aRectangle

Color aRectangle in the frame with gray tone.

initialize: aRectangle font: aFont

Initialize the instance variables of the receiver such that its clipping rectangle is aRectangle and the font is aFont. The destination form is assumed to be the display screen.

initialize: aRectangle**font: aFont dest: aForm**

Initialize the instance variables of the receiver such that its clipping rectangle is aRectangle, the font is aFont, and the destination form is aForm.

recover: aRectangle

Reverse the color of aRectangle in the frame.

reframe: aRectangle

Change the frame of the receiver.

reverse: aRectangle

Reverse the color of aRectangle in the frame.

setFont: aFont

Change the current font to aFont.

setForeColor: fColor backColor: bColor

Set the foreground color to fColor and background color to bColor. They can be either an integer color or a mask form.

show: aString**from: start at: aPoint**

Display the bit pattern of aString from index position start at aPoint in the frame of the receiver.

Class

Class Class is the superclass of all class classes (i.e. metaclasses) in Smalltalk. It provides the common protocol for defining and accessing class variables and pool dictionaries. The subclass creation messages are implemented here as well. Every class is an instance of a metaclass of the same name. The class contains the instance methods while the metaclass contains the class methods.

Inherits From: Behavior Object

Inherited By: (None)

Named Instance Variables:**classPool**

Contains a Dictionary of all the class variables defined by this class. The keys are strings containing the class variable names and the values are the current values of the class variables.

comment

(From class Behavior)

dictionaryArray

(From class Behavior)

instances

(From class Behavior)

name

(From class Behavior)

sharedPools

Contains an Array of symbols for the pool dictionary names referred to by this class.

structure

(From class Behavior)

subclasses

(From class Behavior)

superClass

(From class Behavior)

Class Variables:**InstIndexedBit**

(From class Behavior)

InstNumberMask

(From class Behavior)

InstPointerBit

(From class Behavior)

Pool Dictionaries: (None)

Class Methods:**sortBlock**

Answer a sort block for sorting classes alphabetically.

Instance Methods:**addClassVarName: aString**

Add a new class variable named aString to the receiver.

addSharedPool: aSymbol

Add the shared pool named aSymbol to the receiver shared pool references.

classPool

Answer the dictionary containing the class variables defined in the receiver.

classVarNames

Answer a Set of class variable names defined in the receiver.

edit

Open a ClassBrowser window on the receiver.

fileOutOn: aStream

Append the class definition message for the receiver to aStream.

initialize

Initialize the class variables defined in the receiver. Subclasses usually override this message. The default is to set all class variables to nil.

name

Answer a String containing the receiver name.

removeFromSystem

Remove the receiver from Smalltalk. Report an error if there are any subclasses or instances of the receiver.

rename: aString

Rename the receiver to aString.

sharedPools

Answer an Array of symbols of pool dictionary names referred to by the receiver.

subclass: classSymbol

instanceVariableNames: instanceVariables

classVariableNames: classVariables **poolDictionaries:** poolDictNames

Create or modify the class classSymbol to be a subclass of the receiver with the specified instance variables, class variables, and pool dictionaries.

variableByteSubclass: classSymbol

classVariableNames: classVariables **poolDictionaries:** poolDictNames

Create or modify the class classSymbol to be a variable byte subclass of the receiver with the specified class variables and pool dictionaries.

variableSubclass: classSymbol

instanceVariableNames: instanceVariables

classVariableNames: classVariables **poolDictionaries:** poolDictNames

Create or modify the class classSymbol to be a variable subclass of the receiver with the specified instance variables, class variables, and pool dictionaries.

ClassBrowser

Class ClassBrowser implements a window on all the methods for a single class. Methods can be browsed, edited and cross-referenced. A ClassBrowser window consists of three panes. The first contains the type of methods being browsed (instance or class). The second contains the list of method selectors for the selected type. And the third contains the source code for the selected method. The window's label shows the class name being browsed.

Inherits From:

Object

Inherited By: (None)

Named Instance Variables:

browsedClass

Contains the class on which the browser was opened.

selectedDictionary

Contains browsedClass or browsedClass class, depending upon whether instance or class methods are selected.

selectedMethod

Contains the selector of the selected method, or nil if no method is selected.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

openOn: aClass

Create a class browser window on aClass. Define the type, behavior and relative size of each pane and schedule the window.

ClassHierarchyBrowser

Class ClassHierarchyBrowser implements a window on all the classes in Smalltalk/V. It allows for class definitions and methods to be browsed and edited; the source code for a class to be written to a file; and the senders and implementors of messages to be displayed. A ClassHierarchyBrowser window consists of five panes. The first contains the class hierarchy. The second contains the method selectors for the selected class. The third contains the source code for the selected method or the class definition method for the selected class. The fourth and fifth panes are mutually exclusive, containing the type of the method dictionary selected (instance methods or class methods).

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

browsedClasses

Contains an OrderedCollection of strings of class names with the subclasses indented to show the hierarchy.

hiddenClasses

Contains a Set of classes whose subclasses should not be displayed. These classes have an ellipsis (...) appended to their name in the hierarchy.

instanceSelectedLast

Contains true if the instance pane was selected last, false if the class pane selected last.

methodSelectedLast

Contains true if a method selector was selected last, false if a class name was selected last.

originalClasses

Contains the collection of classes passed as the argument to the openOn: message.

selectedClass

Contains the most recently selected class, or nil if no class is selected.

selectedMethod

Contains the most recently selected method selector, or nil if no method is selected.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:**openOn: aCollection**

Create a class hierarchy browser window giving access to the classes in aCollection and their subclasses.

ClassReader

A ClassReader supports Smalltalk source code reading and installation (compilation) from a stream, and writing to a stream. The source code is in 'chunk' format (See Chapter 13, Maintaining Smalltalk/V, for a definition of chunk). A ClassReader is used for writing the entire source code of a class to a file, for reading a file to define a class, and for reading portions of a file to selectively recover methods (for example, using the DiskBrowser to read the change log).

Inherits From: Object

Inherited By: (None)

Named Instance Variables:**class**

Contains the class to be worked on by the ClassReader.

Class Variables:

(None)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:

forClass: aClass

Answer an instance of the receiver for aClass.

Instance Methods:

fileInFrom: aStream

Read chunks from aStream until an empty chunk (a single '!') is found. Compile each chunk as a method for the class described by the receiver. Log the source code of the method to the change log.

fileOutOn: aStream

File out all the methods for the class described by the receiver to aStream, in chunk format.

Collection

Class Collection is the superclass of all the collection classes. It is an abstract class defining the common protocol for all of its subclasses. Collections are the basic data structures used to store objects in groups in either a linear or nonlinear fashion (e.g. hashed). This class provides the protocol to directly access (or store) a particular element in a collection, to access all the elements of a collection in a particular order, or to perform some block of code for each element accessed.

Inherits From:

Object

Inherited By:

Array Bag Bitmap ByteArray CompiledMethod Dictionary
 FileHandleFixedSizeCollection IdentityDictionary
 IndexedCollection Interval MethodDictionary
 OrderedCollection Process Set SortedCollection String
 Symbol SymbolSet SystemDictionary

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries:

(None)

Class Methods:**with: anObject**

Answer a collection with only one element, anObject.

with: firstObject with: secondObject

Answer a collection of two elements, firstObject and secondObject.

with: firstObject with: secondObject with: thirdObject

Answer a collection of three elements, firstObject, secondObject, and thirdObject.

with: firstObject**with: secondObject with: thirdObject with: fourthObject**

Answer a collection of four elements, firstObject, secondObject, thirdObject, and fourthObject.

Instance Methods:**add: anObject**

Answer anObject. Add anObject to the receiver collection.

addAll: aCollection

Answer aCollection. Add each element of aCollection to the elements of the receiver.

asArray

Answer an Array containing all the elements of the receiver.

asBag

Answer a Bag containing the elements of the receiver.

asOrderedCollection

Answer an OrderedCollection containing the elements of the receiver.

asSet

Answer a Set containing the elements of the receiver.

asSortedCollection

Answer a SortedCollection containing the elements of the receiver sorted in ascending order.

asSortedCollection: aBlock

Answer a SortedCollection containing the elements of the receiver sorted according to aBlock.

collect: aBlock

For each element in the receiver, evaluate aBlock with that element as the argument. Answer a new collection containing the results as its elements from the aBlock evaluations.

deepCopy

Answer a copy of the receiver with shallow copies of each element.

detect: aBlock

Answer the first element of the receiver that causes aBlock to evaluate to true (with that element as the argument). If no such element is found, report an error.

detect: aBlock ifNone: exceptionBlock

Answer the first element of the receiver that causes aBlock to evaluate to true (with that element as the argument). If no such element is found, evaluate exceptionBlock (with no arguments).

do: aBlock

For each element in the receiver, evaluate aBlock with that element as the argument. This method should be implemented in the class of the receiver.

includes: anObject

Answer true if the receiver contains an element equal to anObject, else answer false.

inject: initialValue into: aBinaryBlock

For each element in the receiver collection, evaluate aBinaryBlock with that element as the argument. Starting with initialValue, the block is also provided with its own value from the previous evaluation. Answer this value at the end of the block evaluations.

isEmpty

Answer true if the receiver collection contains no elements, else answer false.

notEmpty

Answer true if the receiver collection contains one or more elements, else answer false.

occurrencesOf: anObject

Answer the number of elements contained in the receiver collection that are equal to anObject.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

reject: aBlock

For each element in the receiver, evaluate aBlock with that element as the argument. Answer a new collection containing those elements of the receiver for which aBlock evaluates to false.

remove: anObject

Answer anObject. Remove the element equal to anObject from the receiver collection. If such an element is not found, report an error.

remove: anObject ifAbsent: aBlock

Answer anObject. Remove an element equal to anObject from the receiver collection. If such an element is not found, evaluate aBlock (with no arguments).

removeAll: aCollection

Answer aCollection. Remove all the elements contained in aCollection from the receiver collection.

select: aBlock

For each element in the receiver, evaluate aBlock with that element as the argument. Answer a new collection containing those elements of the receiver for which aBlock evaluates to true.

shallowCopy

Answer a copy of the receiver which shares the receiver elements.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reinstated.

ColorForm

A ColorForm contains an array of bitmaps. All bits in the same location of each bitmap collectively represent the color of the pixel at that location.

Inherits From: **Form DisplayMedium DisplayObject Object**

Inherited By: **(None)**

Named Instance Variables:

bits

(From class Form)

byteWidth

(From class Form)

deviceType

(From class Form)

height

(From class Form)

offset

(From class Form)

width

(From class Form)

Class Variables:

BlackMask

(From class Form)

DarkGrayMask

(From class Form)

GrayMask

(From class Form)

LightGrayMask

(From class Form)

PrinterMode

(From class Form)

WhiteMask

(From class Form)

Pool Dictionaries: **(None)**

Class Methods:**color: aColor**

Answer a mask form filled with aColor.

new

Answer a new ColorForm.

Instance Methods:**at: aPoint**

Answer the color for pixel at aPoint.

byteValueAtX: xInteger Y: yInteger

Answer the byte at the position specified by the point (xInteger @ yInteger) in the first plane.

compatibleForm

Answer the class of internal form most similar to the receiver.

compatibleMask

Answer the class of mask form most suitable for use with the receiver.

width: wInteger height: hInteger initialByte: aByte

Change the receiver width to wInteger and height to hInteger, and initialize every byte in the bitmap to aByte.

width: wInteger height: hInteger initialColor: aColor

Change the receiver width to wInteger and height to hInteger, and fill with aColor.

ColorScreen

A ColorScreen is like a DisplayScreen except that it has multiple planes (like ColorForm) and thus support multiple colors.

Inherits From: DisplayScreen Form DisplayMedium DisplayObject Object

Inherited By: (None)

Named Instance Variables:**bits**

(From class Form)

byteWidth

(From class Form)

deviceType

(From class Form)

height

(From class Form)

offset

(From class Form)

width

(From class Form)

Class Variables:

BackColor

(From class DisplayScreen)

BlackMask

(From class Form)

DarkGrayMask

(From class Form)

GrayMask

(From class Form)

HighResIBeam

(From class DisplayScreen)

LightGrayMask

(From class Form)

LowResIBeam

(From class DisplayScreen)

Mode

(From class DisplayScreen)

PrinterMode

(From class Form)

WhiteMask

(From class Form)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

compatibleForm

Answer the class of internal form most similar to the receiver.

compatibleMask

Answer the class of mask form most suitable for use with the receiver.

refreshColor

Load the default color palette.

Commander

A Commander commands an Array of pens. When it receives a pen related message, it passes the operation to every pen under its command. When a Commander is in action, it gives the illusion that all of its pens are operating simultaneously.

Inherits From: Pen BitBlt Object

Inherited By: (None)

Named Instance Variables:

clipHeight

(From class BitBlt)

clipWidth

(From class BitBlt)

clipX

(From class BitBlt)

clipY

(From class BitBlt)

destForm

(From class BitBlt)

destX

(From class BitBlt)

destY

(From class BitBlt)

direction

(From class Pen)

downState

(From class Pen)

fractionX

(From class Pen)

fractionY

(From class Pen)

halftone

(From class BitBlt)

height

(From class BitBlt)

pens

Contains an Array of pens being commanded.

rule

(From class BitBlt)

sourceForm

(From class BitBlt)

sourceX

(From class BitBlt)

sourceY

(From class BitBlt)

width

(From class BitBlt)

Class Variables:

DoubleCenter

(From class Pen)

Pool Dictionaries: (None)

Class Methods:**new: anInteger**

Answer aCommander initialized to anInteger number of pens.

Instance Methods:**clipRectAll: aRectangle**

Set the clipping rectangle of every pen to aRectangle.

destX

Answer the x coordinate of the first pen.

destY

Answer the y coordinate of the first pen.

direction: anInteger

Set the direction of every pen to anInteger number of degrees.

down

Set all the pens down.

ellipse: anInteger aspect: aFraction

Make each pen draw an ellipse with aspect ratio aFraction.

fanOut

Change the direction of each pen by an increment of 360 / number of pens.

go: anInteger

Move all pens a distance of anInteger in their current direction.

goto: aPoint

Move the first pen to aPoint and then move the remaining pens by the same distance and direction as the first move.

lineUpFrom: startPoint to: endPoint

Place all the pens on equi-distant points on the line defined by startPoint and endPoint.

location

Answer a Point indicating the position of the first pen.

place: aPoint

Set the position of the first pen to aPoint and modify the position of the remaining pens by the amount of change in the first pen. No drawing takes place.

turn: anInteger

Change the direction of all the pens by anInteger number of degrees.

up

Lift all the pens.

CompiledMethod

A CompiledMethod is produced by the Smalltalk/V compiler and interpretively executed by the Smalltalk/V virtual machine.

Inherits From: Array FixedSizeCollection IndexedCollection Collection Object

Inherited By: (None)

This class contains indexed instance variables.

Named Instance Variables:

byteCodeArray

Contains a ByteArray with codes to be executed.

class

Contains the class whose method dictionary contains the method.

primitive

Contains the user primitive number, or zero if none.

selector

Contains a Symbol representing the message selector.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (All private)

Instance Methods: (All private)

Compiler

Class Compiler is used for converting Smalltalk source code to compiled methods and for evaluating Smalltalk expressions. There are no instances of this class because its behavior is entirely defined with class messages.

Inherits From: Object

Inherited By: LCompiler

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

compile: aString in: aClass

Compile the method aString in aClass. If the method compiles correctly, answer an Association whose key is the method selector and whose value is the compiled method. If not, answer nil and report the error on the Transcript.

compile: aString

in: aClass notifying: requestor ifFail: exceptionBlock

Compile the method aString in aClass. If the method compiles correctly, answer an Association whose key is the method selector and whose value is the compiled method. If not, send the messages: requestor compilerError: errorString at: position in: codeString for: aClass. exceptionBlock value.

evaluate: aString

Compile and evaluate the method ('Doit ', aString) in the context of UndefinedObject. If the method compiles correctly, answer the result of the evaluation. If not, report the error on the Transcript and answer nil.

evaluate: aString

in: aClass to: doitReceiver notifying: requestor ifFail: exceptionBlock

Compile and install the method ('Doit ', aString) in aClass. If the method compiles correctly, answer: doitReceiver Doit. If not, send the messages: requestor compilerError: errorString at: position in: aString. exceptionBlock value. In any case remove the selector #Doit from aClass' method dictionary.

positionsOf: aString in: aClass

Answer highlighting information for source aString in class aClass.

positionsOf: aString

in: aClass notifying: requestor ifFail: exceptionBlock

Answer highlighting information for source aString in class aClass. If compile error, notify requestor and evaluate exceptionBlock.

Instance Methods: (None)

Context

A Context is used to describe the execution state of blocks of code (enclosed in square brackets). They are the objects to which value, value:, value:value: messages are sent to start block evaluation.

Inherits From: Object

Inherited By: HomeContext

Named Instance Variables:**blockArgumentCount**

Contains an integer representing the number of block arguments.

homeContext

Contains the first context for the method, which includes the method arguments and temporaries as indexed instance variables.

startPC

Contains an integer which is the initial program counter for the block.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:**fork**

Create and schedule a new process for the expressions in the receiver block, at the current priority.

forkAt: aNumber

Create and schedule a new process for the expressions in the receiver block, at priority aNumber.

homeContext

Answer the home context.

value

Answer the result of evaluating the no argument block described by the receiver.

value: anObject

Answer the result of evaluating the one argument block described by the receiver.

value: arg1 value: arg2

Answer the result of evaluating the two argument block described by the receiver.

whileFalse: aBlock

Repetitively evaluate the receiver block and aBlock, until the result of receiver block evaluation is true. Answer nil.

whileTrue: aBlock

Repetitively evaluate the receiver block and aBlock, until the result of receiver block evaluation is false. Answer nil.

CursorManager

An instance of **CursorManager** contains the bit pattern to display a cursor shape. In addition, it contains the methods for managing the moving, hiding, and displaying of the cursor. This class serves as an interface between the Smalltalk code and the mouse driver when a mouse driver is loaded in the memory.

Inherits From: **Object**

Inherited By: **NoMouseCursor**

Named Instance Variables:

hotSpot

Contains a Point relative to the top left corner of the cursor shape which aligns the cursor image to the cursor position on the display screen.

image

Contains a String of the cursor image in the format required by the Microsoft mouse when the mouse driver is loaded. Contains a Form of the cursor image when the mouse driver is not loaded.

Class Variables:

NoMouse

Contains a Boolean indicating whether the mouse driver is loaded (false) or not (true).

Position

Contains a Point representing the cursor position on the display screen.

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Cursors

Defines variables for the various cursor shapes.

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods:

corner

Answer the corner cursor.

crossHair

Answer the cross hair cursor.

downArrow

Answer the down arrow cursor.

execute

Answer the hour glass cursor.

hand

Answer the hand cursor.

leftArrow

Answer the left arrow cursor.

normal

Answer the arrow cursor.

origin

Answer the origin cursor.

rightArrow

Answer the right arrow cursor.

scroll

Answer the scroll cursor.

upArrow

Answer the up arrow cursor.

Instance Methods:**change**

Change Cursor to be the receiver.

display

Display the receiver on the screen.

hide

Hide the cursor from the screen.

hotSpot

Answer the hot spot of the receiver cursor.

initForm: aForm hotSpot: aPoint

Initialize the contents of the receiver cursor from aForm (mask on top of cursor shape) using aPoint as its hot spot.

initialize: aForm hotSpot: aPoint

Initialize the contents of the receiver cursor from aForm using aPoint as its hot spot.

isThereInput

Answer true if there is input from keyboard or mouse, else answer false.

offset

Answer a copy of the cursor position.

offset: aPoint

Set the cursor position to aPoint. Answer the new position.

Date

A Date represents a particular day since the start of the Julian calendar. Class Date defines the protocol for creating, comparing, and computing dates.

Inherits From: Magnitude Object

Inherited By: (None)

Named Instance Variables:

day

Contains the number of days from January 1, 1901 up to the date represented by this instance.

Class Variables:

MonthNames

Contains a Dictionary. The keys of the dictionary are instances of class Symbol representing the month names in both the full and abbreviated form. For each of the keys, the corresponding value is an integer from 1 to 12 indicating the index of the month in the year.

MonthStrings

Contains a Dictionary. The keys of the dictionary are instances of class String representing the month names in both the full and abbreviated form. For each of the keys, the corresponding value is an instance of class Symbol representing the name of the month in the abbreviated 3 character form.

Pool Dictionaries: (None)

Class Methods:

calendarForMonth: aSymbol year: anInteger

Answer a String containing the formatted calendar for the month name aSymbol in the year anInteger.

dateAndTimeNow

Answer an Array of two elements. The first element is a Date representing the current date and the second element is a Time representing the current time.

dayOfWeek: aSymbol

Answer a number from 1 to 7 indicating the weekday number for aSymbol (1 meaning Monday, to 7 meaning Sunday).

daysInMonth: aSymbol forYear: anInteger

Answer the total number of days for the month aSymbol in the year yInteger.

daysInYear: anInteger

Answer the total number of days for the year anInteger.

fromDays: anInteger

Answer a Date that is anInteger number of days before or after January 1, 1901 depending on the sign of anInteger.

fromString: aString

Answer a Date specified by aString. aString contains first the day number then the month name and then the year separated with blanks.

indexOfMonth: aSymbol

Answer a number from 1 to 12 indicating the month index for the aSymbol.

leapYear: anInteger

Answer true if the year anInteger is a leap year, else answer false.

leapYearsTo: anInteger

Answer the number of leap years from 1901 to the year number before anInteger.

monthNameFromString: aString

Answer a Symbol for a month name corresponding to the month name in aString.

nameOfDay: anInteger

Answer the weekday name as a Symbol corresponding to the weekday index anInteger (Monday for index 1, to Sunday for index 7).

nameOfMonth: anInteger

Answer the month name as a Symbol corresponding to the month index anInteger (January for index 1, to December for index 12).

newDay: dInteger **month:** aSymbol **year:** yInteger

Answer a Date of the day dInteger in the month aSymbol for the year yInteger.

newDay: dInteger **year:** yInteger

Answer a Date of the day dInteger in the year yInteger.

today

Answer the current date.

Instance Methods:

< aDate

Answer true if the receiver is before aDate.

<= aDate

Answer true if the receiver is before or the same as aDate.

= aDate

Answer true if the receiver is the same as aDate.

> aDate

Answer true if the receiver is after aDate.

>= aDate

Answer true if the receiver is the same or after aDate.

addDays: anInteger

Answer a Date that is anInteger number of days after the receiver.

asSeconds

Answer the number of seconds that have elapsed from January 1, 1901 to the receiver.

day

Answer the number of days from the receiver to January 1, 1901.

dayIndex

Answer a number from 1 to 7 indicating the weekday number of the receiver (1 meaning Monday, to 7 meaning Sunday).

dayName

Answer the name of the weekday of the receiver.

dayOfMonth

Answer a number from 1 to 31 indicating the day number within the month of the receiver.

dayOfYear

Answer a number from 1 to 366 indicating the day within the year of the receiver.

daysInMonth

Answer the total number of days for the receiver month.

daysInYear

Answer the total number of days for the receiver year.

daysLeftInMonth

Answer number of days remaining in the receiver month.

daysLeftInYear

Answer number of days remaining in the receiver year.

elapsedDaysSince: aDate

Answer the number of elapsed days between the receiver and aDate.

elapsedMonthsSince: aDate

Answer the number of elapsed months between the receiver and aDate.

elapsedSecondsSince: aDate

Answer the number of elapsed seconds between the receiver and aDate.

firstDayInMonth

Answer the number of the first day in the receiver month relative to the beginning of the receiver year.

firstDayOfMonth

Answer a Date representing the first day in the receiver month.

formPrint

Answer a string representing the receiver Date in the form: mm/dd/yy.

hash

Answer the integer hash value for the receiver.

monthIndex

Answer a number from 1 to 12 indicating the month of the receiver.

monthName

Answer a Symbol representing the month name of the receiver.

previousWeekday: aSymbol

Answer a Date reflecting the most recent day name represented by aSymbol preceding the receiver.

printOn: aStream

Append the ASCII representation of the receiver to aStream in the form: mmmm dd, yyyy. (The form yyyy is satisfied only for positive year numbers of 4 digits).

subtractDate: aDate

Answer the number of days between the receiver and aDate.

subtractDays: anInteger

Answer the date that is anInteger number of days before the receiver Date.

year

Answer the year number of receiver Date.

Debugger

A Debugger is a window application which allows debugging a Process in two different windows: a single pane walkback window and a six pane debugger window. The Debugger initially creates a walkback window. If requested via the ~debug~ menu choice, the walkback window is replaced with the debug window. The debug window allows the debugged process to be resumed from the point of interruption or restarted by resending a selected message. Traced execution can be controlled by hop, skip, and jump buttons in the window label.

Inherits From: Inspector Object

Inherited By: (None)

Named Instance Variables:

breakpointArray

Contains an Array of pairs of classes and selectors for methods which have breakpoints set.

breakpoints

Contains a SortedCollection of methods which have breakpoints set.

browseWalkback

Contains true if browsing the walkback and false if browsing breakpoints.

instIndex

(From class Inspector)

instList

(From class Inspector)

instPane

(From class Inspector)

label

Contains a String used to give a debugger window the same label as a walkback window

method

Contains the selected CompiledMethod or nil if none.

methodPane

Contains a TextPane which contains the method source.

object

(From class Inspector)

positions

Contains information for highlighting the method source.

process

Contains the Process object being debugged.

resumable

Contains true if the process is resumable, otherwise false.

source

Contains a String representing the source code of the selected method.

stream

Contains (1) a ReadStream for scanning the source of the selected method or (2) a WriteStream used to generate walkback lines for the debug window.

temps

Contains an OrderedCollection of lines containing the names of temporary variables for the selected method.

walkback

Contains an OrderedCollection of walkback lines for the debug window.

walkbackIndex

Contains the index of the selected walkback line or nil if none.

Class Variables:

(None)

Pool Dictionaries:

(None)

Class Methods:

(None)

Instance Methods:

walkbackFor: aProcess label: aString

Pop-up a walkback window with label equal to aString). Display the stacked message sends for the receiver in the window.

walkbackLabel: aString

Pop-up a walkback window with the label equal to aString. Display the stacked message sends for the current process in the window.

DeletedClass

An instance of deleted class is used to replace classes removed from the system.

Inherits From: Object

Inherited By: (None)

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods: (None)

DemoClass

This class demonstrates graphics and animation.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

pen

Contains a Pen for drawing the demonstrations.

rectangle

Contains a Rectangle which limits the drawing area.

Class Variables:

Count

Contains an integer used as the current drawing color.

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

bounceBall

Display a bouncing ball where speed of the ball depends on the position of the cursor.

demoMenu

Answer the menu for the receiver.

dragon

Draw a dragon pattern.

dragon: anInteger

Draw a dragon pattern where anInteger is the recursion factor.

mandala

Draw a mandala.

multiEllipse

Draw 5 ellipses.

multiMandala

Draw 8 mandalas.

multiPentagon

Draw multiple pentagons.

multiPolygon: anInteger

Draw mutiple polygons where each polygon has anInteger number of sides.

multiSpiral

Draw 4 spirals.

run

Initialize and start the animation demonstration.

walkLine

Draw a rotating line.

Dictionary

A Dictionary is a collection of key/value pairs of objects. The keys in a dictionary are unique, whereas values may be duplicated. A Dictionary may be searched either by key or by value. Key searches use hashing for efficiency. Elements may be entered into and extracted from a dictionary either as a pair of objects (e.g., at:put:) or as an Association (e.g., add:). Internally, a Dictionary stores the key/value pairs as a set of associations whereas an IdentityDictionary stores the key/value pairs in successive array elements (see IdentityDictionary).

Inherits From: Set Collection Object

Inherited By: IdentityDictionary MethodDictionary SystemDictionary

Named Instance Variables:

contents

(From class Set)

elementCount

(From class Set)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

add: anAssociation

Answer anAssociation. Add anAssociation to the receiver.

associationAt: aKey

Answer the Association whose key equals aKey from the receiver. If not found, report an error.

associationAt: aKey ifAbsent: aBlock

Answer the Association whose key equals aKey from the receiver. If not found, evaluate aBlock (with no arguments).

associationsDo: aBlock

Answer the receiver. For each key/value pair in the receiver, evaluate aBlock with that pair as the argument.

at: aKey

Answer the value of the key/value pair whose key equals aKey from the receiver. If not found, report an error.

at: aKey ifAbsent: aBlock

Answer the value of the key/value pair whose key equals aKey from the receiver. If not found, evaluate aBlock (with no arguments).

at: aKey put: anObject

Answer anObject. If the receiver contains the key/value pair whose key equals aKey, replace the value of the pair with anObject. Else add the aKey/anObject pair.

deepCopy

Answer a copy of the receiver with shallow copies of each element.

do: aBlock

Answer the receiver. For each value in the receiver, evaluate aBlock with that value as the argument.

includes: anObject

Answer true if the receiver contains the key/value pair whose value equals anObject, else answer false.

includesKey: aKey

Answer true if the receiver contains aKey, else answer false.

inspect

Open a dictionary inspector window on the receiver.

keyAtValue: anObject

Answer the key in the receiver whose paired value equals anObject. If not found, answer nil.

keyAtValue: anObject **ifAbsent:** aBlock

Answer the key in the receiver whose paired value equals anObject. If not found, evaluate aBlock (with no arguments).

keys

Answer a Set containing all the keys in the receiver.

keysDo: aBlock

Answer the receiver. For each key in the receiver, evaluate aBlock with the key as the argument.

occurrencesOf: anObject

Answer the number of key/value pairs in the receiver, whose values are equal to anObject.

remove: anObject **ifAbsent:** aBlock

Remove the key/value pair whose value is anObject from the receiver dictionary. This method reports an error since the values are not unique in a dictionary, the keys are.

removeAssociation: anAssociation

Answer the receiver after anAssociation has been removed from it. If anAssociation is not in the receiver, report an error.

removeKey: aKey

Answer the receiver with the key/value pair whose key equals aKey removed. If such a pair is not found, report an error.

removeKey: aKey **ifAbsent:** aBlock

Answer aKey. Remove the key/value pair whose key equals aKey from the receiver. If such a pair is not found, evaluate aBlock (with no arguments).

select: aBlock

For each key/value pair in the receiver, evaluate aBlock with the value part of the pair as the argument. Answer a new object containing those key/value pairs for which aBlock evaluates to true.

shallowCopy

Answer a copy of the receiver which shares the receiver elements.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reinstated.

values

Answer a Bag containing all the values of the key/value pairs in the receiver.

DictionaryInspector

Class DictionaryInspector implements a window on a dictionary which allows the entries of a dictionary to be viewed and changed. The left list pane displays the ASCII representation of all the dictionary keys. The right text pane displays an ASCII representation of the value associated with the selected key.

Inherits From: Inspector Object

Inherited By: (None)

Named Instance Variables:

instIndex
 (From class Inspector)

instList
 (From class Inspector)

instPane
 (From class Inspector)

object
 (From class Inspector)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods: (All private)

Directory

A Directory represents a disk directory with a device letter and a path name string. Files are generally described in terms of a directory and a file name. A FileStream may be created by sending the message file: or newFile: to a Directory.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

drive
 Contains a Character representing the disk drive letter.

pathName
 Contains a String representing the path name (from the root directory) of the directory, not including the drive letter.

volumeLabel

Contains a String representing the label of the disk containing the directory.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

create: newPathName

Create a DOS directory on disk with complete path name newPathName.

current

Answer a Directory representing the current DOS directory.

currentDisk

Answer the current default drive (0 = A, 1 = B, etc.).

extractDateTimeFrom: aString

Answer a String in form 'yy-mm-dd hh:mm:ss' describing date and time from DOS directory entry aString.

extractFileNameFrom: aString

Answer a string representing the file name from a DOS directory entry aString.

extractFlagsFrom: aString

Answer a String containing attribute flags from a DOS directory entry aString.

Attributes are: 'r' read only, 'h' hidden, 's' system, and 'a' archive.

extractSizeFrom: aString

Answer the file size extracted from a DOS directory entry aString.

pathName: aString

Answer a Directory described by the complete path name in aString.

remove: aString

Remove the DOS directory with the path name of aString.

Instance Methods:

= aDirectory

Answer true if aDirectory represents the same directory as the receiver, else answer false.

create

Create a DOS directory on disk for the receiver directory.

drive

Answer the disk drive letter of the receiver.

drive: aCharacter

Initialize the drive for the receiver to aCharacter.

file: aString

Answer a FileStream for the file named aString in the current directory. If the file does not exist, it will be created.

formatted

Answer a collection of arrays of file information for the receiver directory. Each array has four entries: file name, size, date/time and attributes.

freeDiskSpace

Answer the free space in bytes on the disk containing the current directory.

hasSubdirectory

Answer true if the receiver has a subdirectory.

makeCurrent

Make the receiver the current DOS directory.

newFile: aString

Answer a FileStream for the file named aString in the current directory. If the file exists, it will be removed and a new file will be created.

pathName

Answer a String representing the path name of the receiver directory (drive letter not included).

pathName: aString

Set the receiver directory path name to aString.

remove

Remove the directory described by the receiver from the disk.

subdirectories

Answer an OrderedCollection of arrays, where each Array contains the complete path name and the file name of a subdirectory of the receiver.

volumeLabel

Answer the volume label of the disk containing the receiver.

DiskBrowser

Class DiskBrowser implements a window on the complete directory hierarchy on a disk. It replaces most DOS file commands. Directories can be created, deleted and browsed. Files can be created, deleted, copied, renamed, browsed, printed, edited and their attributes may be modified.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

allFileMenu

Contains the contents pane menu when the entire file has been read.

contentsPane

Contains the TextPane used to display file contents.

device

Contains the disk drive character.

directoryIndex

Contains the index of the selected directory.

directoryList

Contains an OrderedCollection of strings describing the directory hierarchy.

hiddenDirectories

Contains a set of directories currently being hidden.

noFileMenu

Contains the contents pane menu when no file has been read.

partFileMenu

Contains the contents pane menu when part of the file has been read.

pathNameArray

Contains an Array that parallels directoryList. Each entry contains the complete path name of a directory.

selectedDirectory

Contains a Directory for the selected DOS directory, or nil if no directory is selected.

selectedFile

Contains a String representing the selected file name, or nil if no file is selected.

sortCriteria

Contains a block which describes how to sort the files in a directory: by name, size or creation date/time.

sortedFileList

Contains an OrderedCollection of arrays describing the files in the selected directory. Each Array entry has four elements: the file name, size, creation date/time and attributes (mode).

sortPane

Contains a ListPane which describes the sort criteria.

volumeLabel

Contains a String representation of the volume label.

wholeFileRequest

Contains true if the entire file is to be displayed in the text pane, false if only the head and tail are to be displayed.

Class Variables:

(None)

Pool Dictionaries:**CharacterConstants**

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods: (None)

Instance Methods:

openOn: driveCharacter

Open a disk browser window on the device identified by driveCharacter. Define the type, behavior and relative size of each pane and schedule the window.

Dispatcher

Class Dispatcher is an abstract class which provides the common protocol for its subclasses. Its main function is to provide default methods for processing input from both the keyboard and mouse. It communicates with its associated pane to keep the pane contents up to date. It also provides the protocol for opening, closing, activating, and deactivating a window.

Inherits From: Object

Inherited By: GraphDispatcher ListSelector PointDispatcher
PromptEditor ScreenDispatcher ScrollDispatcher
TextEditor TopDispatcher

Named Instance Variables:

active

Contains true when the pane associated with this dispatcher is active and false when it is not active.

pane

Contains the pane associated with this dispatcher.

Class Variables:

WindowActivateKey

Contains the first character to process when the window is activated and the character is not nil.

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods:

new

Answer a new initialized Dispatcher.

Instance Methods:**activate**

Make the receiver active. Most subclasses supplement this method.

activateWindow

Make the receiver window active by displaying it and then giving control to the main processing loop for the active window.

active

Answer true if the receiver is the active dispatcher, else answer false.

boxOfSize: aPoint

Answer a Rectangle with extent of aPoint and centered at cursor position.

closeIt

Close the receiver window and resume the Scheduler main processing loop.

closeWindow

Close the receiver window and remove the receiver from the Scheduler dispatchers.

cycle

Deactivate the receiver window and cause the windows to rotate.

cyclePane

Move to the next pane in the receiver window.

deactivate

Make the receiver inactive.

deactivateWindow

Mark the receiver to be inactive and change the pane visual cues to reflect it.

display

Display the receiver window.

displayIn: visibleRegions

Display the portion of the pane of the receiver within visibleRegions.

doesNotHandle

Ring the bell for input not handled by the receiver.

homeCursor

Move the cursor to the receiver home position.

isControlActive

Answer true if the receiver is active and contains the cursor. Some subclasses will override and/or supplement this test.

isControlWanted

Answer true if the pane contains the cursor, else answer false.

modified

Indicate whether or not the contents of the receiver pane have been modified. Answer false as default.

open

Open and activate the receiver window of default size.

openIn: aRectangle

Open the receiver window as the active one in aRectangle.

openWindow

Open the receiver window.

pane

Answer the pane of the receiver.

pane: aPane

Set the receiver pane to be aPane and initialize the receiver.

scheduleWindow

Activate the receiver window.

searchForActivePane

Give control to the pane that contains the cursor. This is the main processing loop for the active window.

select

Ring the bell since this function should be implemented by subclasses of this class.

topDispatcher

Answer the top dispatcher for the receiver window.

DispatchManager

A DispatchManager schedules windows by providing messages for adding and removing windows, displaying all the windows, or making a specific window be the top one and activating it. A global variable, Scheduler, contains an instance of class DispatchManager and this should be the only instance in the system.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

dispatchers

Contains an OrderedCollection of dispatchers used to control windows. Windows can be overlaid and the first one in the collection is always the top one displayed on the screen. A top window may not be active, but an active window is always the top one.

Class Variables:

None

Contains a ScreenDispatcher which has control when no window is active.

TransientWrite

Contains the Dispatcher of the TextPane that a process in the active window uses to do output to an inactive window.

Pool Dictionaries: (None)

Class Methods:

new

Answer a DispatchManager with no dispatchers.

Instance Methods:

add: aDispatcher

Add the window associated with aDispatcher to the receiver's stack.

clearScreen

Paint the display screen black.

cycle

Rotate the order of the windows displayed on the screen.

dispatchers

Answer the OrderedCollection of dispatchers known to the receiver.

display

Display all the windows except the top one.

displayAll

Display all windows.

includes: aDispatcher

Answer true if aDispatcher is included in the receiver, else answer false.

initialize

Close all the windows including the System Transcript and then create a new Transcript.

reinitialize

Close all the windows including the System Transcript and then create and schedule the new Transcript.

remove: aDispatcher

Remove aDispatcher from the collection of dispatchers in the receiver.

resume

Restart the main processing loop of the user interface.

run

Drop all the pending message sends, restart the main processing loop by giving control to the top dispatcher.

systemDispatcher

Answer the screen dispatcher.

topDispatcher

Answer the dispatcher for the top window.

DisplayMedium

A DisplayMedium is an abstract class without any instance variables. It contains methods to color and to draw borders around rectangular areas.

Inherits From: DisplayObject Object

Inherited By: BiColorForm ColorForm ColorScreen DisplayScreen Form

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

black

Paint the receiver black.

black: aRectangle

Paint aRectangle in the receiver black.

border: aRectangle

Frame aRectangle with a width 1 border.

border: aRectangle**clippingBox: clipRectangle rule: anInteger mask: aForm**

Frame aRectangle with a width 1 border on the display screen bounded by clipRectangle. The border is formed by combining aForm with the destination using anInteger as the rule.

border: aRectangle rule: anInteger mask: aForm

Frame aRectangle with a width 1 border. The border is formed by combining aForm with the destination using anInteger as the rule.

fill: aForm

Tile the receiver with aForm.

fill: aRectangle**clippingBox: clipRectangle rule: anInteger mask: aForm**

Tile the receiver with aForm bounded by clipRectangle. The combination rule is anInteger.

fill: aRectangle rule: anInteger mask: aForm

Tile the receiver with aForm bounded by aRectangle. The combination rule is specified by anInteger.

gray

Paint the receiver gray.

gray: aRectangle

Paint aRectangle in the receiver gray.

white

Paint the receiver white.

white: aRectangle

Paint aRectangle in the receiver white.

DisplayObject

Class **DisplayObject** is an abstract class which provides the common protocol for transferring a rectangular block of characters from the receiver display object to a **DisplayMedium**. Note that the source of the transfer can be an instance of class **DisplayObject** or its subclasses while the destination must be an instance of class **DisplayMedium** or its subclasses.

Inherits From: **Object**

Inherited By: **BiColorForm ColorForm ColorScreen DisplayMedium
DisplayScreen Form**

Named Instance Variables: **(None)**

Class Variables:

(None)

Pool Dictionaries: **(None)**

Class Methods: **(None)**

Instance Methods:

boundingBox

Answer a Rectangle which bounds the receiver.

display

Show the receiver on the display screen at the position indicated by offset.

displayAt: aPoint

Show the receiver on the display screen at aPoint.

displayAt: aPoint clippingBox: aRectangle

Display the contents of the receiver at aPoint on the display screen using aRectangle as the clipping box.

displayOn: aDisplayMedium

at: aPoint **clippingBox:** aRectangle **rule:** anInteger **mask:** aForm

 Display the receiver on aDisplaymedium at aPoint with aRectangle as the clipping rectangle, anInteger as the combination rule, and aForm as the halftone.

extent

 Answer the width and height of the receiver as a Point.

height

 Answer the height of the receiver.

offset

 Answer the offset of the receiver.

offset: aPoint

 Change the offset to aPoint.

width

 Answer the width of the receiver.

DisplayScreen

A DisplayScreen is a special kind of form whose bit map address and size is determined by the hardware graphics adapter and whose content will be shown directly on the display screen by the adapter. A global variable, Display, contains an instance of either class DisplayScreen or ColorScreen.

Inherits From: Form DisplayMedium DisplayObject Object

Inherited By: ColorScreen

Named Instance Variables:

bits

 (From class Form)

byteWidth

 (From class Form)

deviceType

 (From class Form)

height

 (From class Form)

offset

 (From class Form)

width

 (From class Form)

Class Variables:

BackColor

 Contains a mask Form representing the display screen background color.

BlackMask

(From class Form)

DarkGrayMask

(From class Form)

GrayMask

(From class Form)

HighResIBeam

Contains a Form of the gap selector image in high resolution graphics.

LightGrayMask

(From class Form)

LowResIBeam

Contains a Form of the gap selector image in low resolution graphics.

Mode

Contains the current graphics mode of the display adapter.

PrinterMode

(From class Form)

WhiteMask

(From class Form)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:

ATTmono

Set graphics mode to AT&T monochrome 640 by 400.

backgroundColor: aForm

Tile the screen background with aForm.

checkMode: aSymbol withAspect: aFraction

Check if the Screen mode has changed. If so, reinitialize the environment.

EGAcolor

Set graphics mode to EGA color 640 by 350.

EGAcolorLowRes

Set graphics mode to EGA color 640 by 200.

EGAlowRes

Set graphics mode to EGA mono 640 by 200.

EGAmono

Set graphics mode to EGA mono 640 by 350.

hercules

Set graphics mode to Hercules monochrome 720 by 348.

IBM3270

Set graphics mode to IBM3270 mono 720 by 350.

initSystem

Initialize the environment.

lowRes

Set graphics mode to monochrome 640 by 200.

new

Answer a new DisplayScreen.

newPage2

Answer a new DisplayScreen using the second page of the graphics adaptor.

toshiba

Set graphics mode to toshiba monochrome 640 by 400.

VGA640x480

Set graphics mode to VGA color 640 by 480.

Wyse640x400

Set graphics mode to Wyse mono 640 by 400.

Instance Methods:**background: aRectangle**

Retile aRectangle area of the screen with the background mask.

change: aRectangle from: oldColor to: newColor

Change oldColor to newColor in aRectangle of receiver.

outputToPrinter

Output the contents of the display screen to the printer in landscape orientation.

outputToPrinterUpright

Output the contents of the display screen to the printer in portrait orientation.

refreshColor

Load the default color palette. For DisplayScreen, do nothing since it has no color.

setWidth: wInteger height: hInteger

Set the width and height of the display screen to wInteger and hInteger respectively.

Dos

Class Dos allows DOS interrupt calls or interface with I/O ports directly from Smalltalk code. It contains an Array of registers whose values are loaded into machine registers prior to the call. Upon return from the call, the values in the registers reflect the machine state at the end of the interrupt or I/O port call.

Inherits From:

Object

Inherited By:

(None)

Named Instance Variables:

registers

Contains an Array of 17 elements. The first 16 are AH, AL, BH, BL, CH, CL, DH, DL, SI(H), SI(L), DI(H), DI(L), DS(H), DS(L), ES(H), ES(L). The 17th element is the flag register.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

checkDosError: registers

Generate a walkback if the carry flag is set in registers. The error code is obtained from AL in registers.

dosError: anInteger

Initiate a walkback for a DOS error described by anInteger.

environmentVariable: aString

Answer a String which is the value of DOS environment variable aString if the variable exists, else answer nil.

new

Answer a new instance of Dos.

Instance Methods:

call: aPoint

Far call to seg @ offset

dosPrimitive: function**registers:** anArray **value:** anInteger

Perform a DOS function with an immediate value anInteger and registers in anArray. Functions are: interrupt = 0 inWord = 1 inByte = 2 outWord = 3 outByte = 4 peek = 5 poke = 6 blockMove = 7 farCall = 8

initialize

Initialize registers.

interrupt: anInteger

Issue software interrupt number anInteger.

outByte: byteValue **toPort:** portAddress

Output byteValue to portAddress.

peekFrom: aPoint

Answer the byte value at address aPoint (x = segment, y = offset).

poke: aByte **to:** aPoint

Store aByte into address aPoint (x = segment, y = offset).

registers

Answer the register array.

setPaletteRegister: anInteger **to:** aColor

Set palette register anInteger to aColor.

setReg: regInteger **to:** anObject

Set register regInteger to anObject. AX = 0, BX = 1, CX = 2, DX = 3, SI = 4, DI = 5, DS = 6, ES = 7

setRegHigh: regInteger **to:** valInteger

Set the high byte of register regInteger to valInteger. AX = 0, BX = 1, CX = 2, DX = 3, SI = 4, DI = 5, DS = 6, ES = 7

setRegLow: regInteger **to:** valInteger

Set the low byte of register regInteger to valInteger. AX = 0, BX = 1, CX = 2, DX = 3, SI = 4, DI = 5, DS = 6, ES = 7

EmptySlot

This class represents a deleted element in certain hashed system data structures. There is only a single instance.

Inherits From: Object

Inherited By: (None)

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods: (None)

False

Class False has a single instance, false, representing logical falsehood. This class defines the protocol for logical operations on false.

Inherits From: Boolean Object

Inherited By: (None)

Named Instance Variables: (None)

Class Variables:**(None)****Pool Dictionaries:** **(None)****Class Methods:** **(None)****Instance Methods:****& aBoolean**

Answer true if both the receiver and aBoolean are true, else answer false.

and: aBlock

If the receiver is true, answer the result of evaluating aBlock (with no arguments), else answer false.

eqv: aBoolean

Answer true if the receiver is equivalent to aBoolean, else answer false.

ifFalse: aBlock

If the receiver is true, answer the result of evaluating aBlock (with no arguments), else answer nil.

ifFalse: falseBlock ifTrue: trueBlock

If the receiver is true, answer the result of evaluating trueBlock, else answer the result of evaluating falseBlock. Both blocks are evaluated with no arguments.

ifTrue: aBlock

If the receiver is true, answer the result of evaluating aBlock (with no arguments), else answer nil.

ifTrue: trueBlock ifFalse: falseBlock

If the receiver is true, answer the result of evaluating trueBlock, else answer the result of evaluating falseBlock. Both block are evaluated with no arguments.

not

Answer true if the receiver is false, else answer false.

or: aBlock

If the receiver is false, answer the result of evaluating aBlock (with no arguments), else answer true.

xor: aBoolean

Answer true if the receiver is not equivalent to aBoolean, else answer false.

| aBoolean

Answer true if either the receiver or aBoolean are true, else answer false.

File

A File provides sequential or random access to a DOS file. Each read operation answers one page (maximum 2K bytes) of the file with the exception of the last page which may have fewer than page size bytes. The number of bytes to write may be from one to the page size. A FileStream provides buffered stream access on a File. A File provides the logical page access using a FileHandle.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

directory

Contains the Directory which includes the file.

fileId

Contains the FileControlBlock used to access the file.

name

Contains a String representing the file name.

Class Variables:

PageSize

Contains an integer representing the number of bytes in a page to read from or write to a file. When this variable is set, all the files opened afterwards will assume the new page size.

Pool Dictionaries: (None)

Class Methods:

changeModeOf: aString to: attrString

Change the attributes of the file named aString to those of attrString. Attributes are: \$r - read only, \$h - hidden, \$s - system, \$a - archive (see DOS manual).

copy: oldFile to: newFile

Copy the file named oldFile to the file named newFile.

fileName: nString extension: eString

Answer a String which is a file name abbreviated from nString and eString. Lower case vowels are dropped from the right of nString until it is less than or equal to 8 characters.

open: aString in: aDirectory

Answer a File opened on a file named aString in aDirectory.

pageSize

Answer the number of bytes in a file page.

pageSize: anInteger

Set the page size to anInteger for the files opened from now on.

pathName: aString

Answer a FileStream with path name aString.

pathName: aString **in:** aDirectory

Answer a FileStream with path name aString with default directory aDirectory.

remove: aString

Erase the file named aString.

rename: oldString **to:** newString

Rename the file named oldString to newString.

Instance Methods:

close

Close the receiver.

directory

Answer the directory which contains the receiver.

fileId

Answer the file handle used to access the receiver.

flush

Force all data written to the receiver to be recorded on disk.

getDate

Answer an Array of time and date of the file (in DOS format).

name

Answer a String containing the receiver file name.

open

Open the file with a new file handle.

readBuffer: aString **atPosition:** anInteger

Read the page of the receiver file containing the position anInteger into aString.

Answer the number of bytes read.

setDate: anArray

Set the time and date of the the receiver file to anArray (in DOS format).

size

Answer the receiver file size in bytes.

writeBuffer: aString **ofSize:** n **atPosition:** anInteger

Write the first n bytes of aString into the receiver file at position anInteger.

FileHandle

A FileHandle is an Array of two bytes. Its sixteen bit value represents a DOS file handle number which is used to access files.

Inherits From:

ByteArray FixedSizeCollection IndexedCollection
Collection Object

Inherited By: (None)

This class contains indexed byte values.

Class Variables:

FileHandles

Pool Dictionaries: (None)

Class Methods:

open: aString in: aDirectory

Answer a file handle for an opened file named aString in aDirectory.

Instance Methods:

close

Close the file identified by the receiver.

endByte

Answer the size in bytes of the file identified by the receiver.

openIn: fileName

Answer an opened FileHandle for the file named fileName.

readInto: aString atPosition: anInteger

Read a page or less (if at end of file) at position anInteger modulo aString size from the receiver file into aString. Answer the number bytes read.

writeFrom: aString toPosition: anInteger for: size

Write size bytes of aString to the receiver file at position anInteger modulo aString size.

FileStream

A FileStream allows streaming over the characters of files for read and write access. It has an internal record of the current position. It has messages to read and write the character(s) at the current position and cause the position to be advanced. Messages are defined for changing the stream position, so that random access is possible. A FileStream accesses its file in pages, and actually streams across the string object containing the current file page. Note that because writes are buffered, a flush or close message must be sent to the FileStream to insure that the written data is physically recorded.

Inherits From: ReadWriteStream WriteStream Stream Object

Inherited By: (None)

Named Instance Variables:**collection**

(From class Stream)

file

Contains a File being streamed over.

lastByte

Contains the high water mark for the file. For file streams, writeLimit contains the high water mark for the current buffer.

lineDelimiter

Contains a character, either carriage-return or line-feed. File lines are delimited by either the carriage-return line-feed pair, or line-feed only.

pageStart

Contains the position of the current buffer relative to the beginning of the file. A FileStream position is pageStart + position.

position

(From class Stream)

readLimit

(From class Stream)

writeLimit

(From class WriteStream)

writtenOn

Contains a Boolean indicating whether or not the current file page buffer has been changed.

Class Variables:

(None)

Pool Dictionaries:**CharacterConstants**

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:

(None)

Instance Methods:**atEnd**

Answer true if the receiver is positioned at the end (beyond the last object), else answer false.

close

Close the file associated with the receiver stream after writing all the data to the file.

copyFrom: first to: last

Answer a String containing the characters of the receiver stream from positions first to last.

copyFrom: first to: last into: aByteObject

Copy the characters of the receiver stream from positions first to last into an object containing bytes.

cr

Write the line terminating character (carriage-return line-feed pair or line-feed) to the receiver stream.

file

Answer the file over which the receiver is streaming.

flush

Guarantee that any writes to the receiver stream are physically recorded on disk.

lineDelimiter

Answer the line delimiter character for the receiver file stream, either carriage-return or line-feed.

lineDelimiter: aCharacter

Change the line delimiter character to aCharacter.

next

Answer the next character accessible by the receiver and advance the stream position. Report an error if the receiver stream is positioned at end.

nextLine

Answer a String consisting of the characters of the receiver up to the next line delimiter.

nextPut: aCharacter

Write aCharacter to the receiver stream.

nextPutAll: aCollection

Write each of the characters in aCollection to the receiver stream.

pathName

Answer the complete pathname of the file over which the receiver is streaming.

position

Answer the current receiver stream position.

position: anInteger

Set the receiver stream position to anInteger.

FixedSizeCollection

Class FixedSizeCollection is an abstract class for all the indexable fixed size collections. A fixed size collection cannot grow or shrink, hence elements cannot be added or removed from it. Only the element values can be changed.

Inherits From:

IndexedCollection Collection Object

Inherited By: **Array Bitmap ByteArray CompiledMethod FileHandle
Interval String Symbol**

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

with: anObject

Answer a collection with only one element, anObject.

with: firstObject with: secondObject

Answer a collection of two elements, firstObject and secondObject.

with: firstObject with: secondObject with: thirdObject

Answer a collection of three elements, firstObject, secondObject, and thirdObject.

with: firstObject

with: secondObject with: thirdObject with: fourthObject

Answer a collection of four elements, firstObject, secondObject, thirdObject, and fourthObject.

Instance Methods:

add: anObject

Add anObject to the receiver. This method reports an error since fixed size collections cannot grow.

collect: aBlock

For each element in the receiver, evaluate aBlock with that element as the argument. Answer a collection containing the results from the aBlock evaluations as its elements.

copyReplaceFrom: start to: stop with: aCollection

Answer a collection containing the elements of the receiver with entries indexed from start through stop being replaced by the elements of aCollection.

remove: anObject ifAbsent: aBlock

Remove anObject from the receiver. This method reports an error since elements cannot be removed from fixed size collections, they can only be changed.

select: aBlock

For each element in the receiver, evaluate aBlock with that element as the argument. Answer a collection containing those elements of the receiver for which aBlock evaluates to true.

size

Answer the number of indexed instance variables of the receiver.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reinstated.

Float

Class Float defines the protocol to perform arithmetic operations on floating point numbers. The use of this class requires the 8087 coprocessor. If float is used and the coprocessor is not present, the system will report an error.

Inherits From: Number Magnitude Object

Inherited By: (None)

This class contains indexed byte values.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

floatError

Query the floating point coprocessor as to the type of exception and report it.

fromInteger: anInteger

Answer a floating point representation of the argument anInteger.

pi

Answer the floating point representation of pi.

status

Answer the status of the floating point coprocessor as a small integer (refer to coprocessor status word definition).

Instance Methods:

*** aNumber**

Answer the result of multiplying the receiver by aNumber.

+ aNumber

Answer sum of the receiver and aNumber.

- aNumber

Answer the difference between the receiver and aNumber.

/ aNumber

Answer the result of dividing the receiver by aNumber.

// aNumber
Answer the integer quotient after dividing the receiver by aNumber with truncation towards negative infinity.

< aNumber
Answer true if the receiver is less than aNumber, else answer false.

<= aNumber
Answer true if the receiver is less than or equal to aNumber, else answer false.

= aNumber
Answer true if the receiver is equal to aNumber, else answer false.

> aNumber
Answer true if the receiver is greater than aNumber, else answer false.

>= aNumber
Answer true if the receiver is greater than or equal to aNumber, else answer false.

\\" aNumber
Answer the integer remainder after dividing the receiver by aNumber with truncation towards negative infinity.

arcTan
Answer the arc-tangent, an angle in radians, of the receiver.

asFloat
Answer the receiver as a floating point number.

cos
Answer the cosine of the receiver. The receiver is an angle measured in radians

deepCopy
Answer the receiver.

degreesToRadians
Answer the number of radians the receiver represents in degrees.

exp
Answer the exponential of the receiver.

exponent
Answer the floating point number whose value is the exponent part of the floating point representation of the receiver.

hash
Answer the integer hash value for the receiver.

ln
Answer the natural log of the receiver.

negated
Answer the receiver subtracted from zero.

printOn: aStream
Answer the receiver. Append the ASCII representation (maximum of 8 digits) of the receiver to aStream.

radiansToDegrees

Answer the number of degrees the receiver represents in radians.

reciprocal

Answer one divided by the receiver.

shallowCopy

Answer the receiver.

significand

Answer the floating point number whose value is the significand part of the floating point representation of the receiver.

sin

Answer the sine of the receiver. The receiver is an angle measured in radians

sqrt

Answer the square root of the receiver.

tan

Answer the tangent of the receiver. The receiver is an angle measured in radians

timesTwoPower: anInteger

Answer 2 to the exponent anInteger multiplied by the receiver.

truncated

Answer the receiver as a kind of Integer truncating the fraction part.

Font

A Font defines the bitmap patterns and attributes of all characters to be displayed.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

basePoint

Contains a Point whose x is currently undefined and y is the base line relative to the top of the font. Thus the ascent of the font is y and the descent of the font is (font height - y).

charSize

Contains a Point whose x and y coordinates are the width and height of each character in the font.

endChar

Contains an Integer representing the ASCII value of the last character in the font.

fixedWidth

Contains a Boolean which is true if each character has fixed width; is false otherwise.

glyphs

Contains a Form whose contents are the bit patterns of all characters in the font. The image of each character is appended horizontally. Thus the height of the Form is the same as that of each character and the width is the aggregate of all characters.

startChar

Contains an Integer representing the ASCII value of the first character in the font.

xTable

Contains an Array of Integers specifying the x coordinate of each character within the glyphs.

Class Variables:

EightLine

Contains the eight line high font.

FourteenLine

Contains the fourteen line high font.

SixteenLine

Contains the sixteen line high font.

Pool Dictionaries: (None)

Class Methods:

eightLine

Answer the 8 pixel height font.

fourteenLine

Answer the 14 pixel height font.

setSysFont: aFont

Set the global system font to aFont.

sixteenLine

Answer the 16 pixel height font.

Instance Methods:

basePoint

Answer a Point where the y-coordinate is the ascent of the font. Note: charSize y - basePoint y is the descent.

charSize

Answer a Point, the pixel extent of the largest character in the font.

charWidth: aCharacter

Answer the width of aCharacter.

fixedWidth

Answer true if the font is of fixed width, else answer false.

getIndex: aCharacter

Answer the index of aCharacter into xTable.

glyphs

Answer the form containing the image of each character.

height

Answer the height of the font.

installFixedSize: glyphForm

charSize: aPoint **startChar:** sInteger **endChar:** eInteger **basePoint:** bPoint
 Install a font with fixed size characters. The bit pattern of all the characters is in **glyphForm**. The width and height of each character is specified by **aPoint**. The **sInteger** and **eInteger** are the ASCII values of the first and last characters in the font. The base point is specified by **bPoint**.

stringWidth: aString

Return the pixel width of **aString** written in the receiver font.

width

Answer the width of the widest character in the font.

Form

A Form contains a bit map and other instance variables to describe the bit map as a two dimensional array of bits. Class Form provides the protocol to initialize a Form or change the size of a Form. The contents of a Form can be changed by using the messages defined in class BitBlt and its subclasses.

Inherits From: DisplayMedium DisplayObject Object

Inherited By: BiColorForm ColorForm ColorScreen DisplayScreen

Named Instance Variables:

bits

Contains the bit map.

byteWidth

Contains the width of the bit map as an integral number of bytes (e.g., a bit width of 9 has a byte width of 2).

deviceType

Contains an integer indicating the type of hardware that the form describes. Currently defined forms are: 0 for main memory and 1 for the display screen memory.

height

Contains the height of the bit map in bits.

offset

Contains a Point which is the origin of the form relative to the origin of the display screen.

width

Contains the width of the bit map in bits.

Class Variables:**BlackMask**

Contains a mask form whose contents are all black.

DarkGrayMask

Contains a mask form whose contents are all dark gray.

GrayMask

Contains a mask form whose contents are all gray.

LightGrayMask

Contains a mask form whose contents are all light gray.

PrinterMode

Contains a String representing the printer graphics mode (e.g.: Esc K).

WhiteMask

Contains a mask form whose contents are all white.

Pool Dictionaries:**CharacterConstants**

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:**andRule**

Answer the logical AND combination rule.

biColorForm

Answer the device type for bi-color forms.

black

Answer a black mask form.

changeColor

Answer the combination rule for changing colors.

color: aColor

Answer a black mask for colors 0-7, or a white mask for colors 8-15.

colorForm

Answer the device type for ColorForm.

darkGray

Answer a dark gray mask form.

displayPage2

Answer the device type for the second page of display screen.

displayScreen

Answer the device type for the display screen.

erase

Answer the erase combination rule.

exchangeColor

Answer the combination rule which exchanges the foreground color with background.

foreColor: aColor backColor: bColor

Answer a mask form with foreground aColor and background bColor.

fromDisplay: aRectangle

Answer a Form which is a copy of the area aRectangle of the display screen.

gray

Answer a gray mask form.

internalForm

Answer the device type for Form.

lightGray

Answer a light gray mask form.

new

Answer a new Form.

orRule

Answer the logical OR combination rule.

orThru

Answer the combination rule which erases the destination bits corresponding to source one bits and then applies the under rule.

over

Answer the over combination rule.

printerMode: aString

Set the graphic printer mode to aString (e.g. <ESC> K).

reverse

Answer the logical XOR combination rule.

under

Answer the under combination rule.

white

Answer a white mask form.

width: wInteger height: hInteger

Answer a white Form whose width is wInteger and height is hInteger.

Instance Methods:**at: aPoint**

Answer the bit at location aPoint.

at: aPoint put: aBit

At location aPoint, put aBit.

backColor

Answer the background color, black.

backColor: aColor

Set the background color to aColor. For Forms do nothing.

bitmap

Answer the bitmap of the receiver.

byteValueAt: aPoint **put:** aByte

Replace the byte at the position aPoint by aByte.

byteValueAtX: xInteger **Y:** yInteger

Answer the byte at the position specified by the point (xInteger @ yInteger).

compatibleForm

Answer the class of internal form most similar to the receiver.

compatibleMask

Answer the class of mask form most suitable for use with the receiver.

copy: aRectangle **from:** aForm **to:** aPoint **rule:** anInteger

Copy from aRectangle in aForm to aPoint on the receiver by the rule anInteger.

deviceType: anInteger

Set the device type of the receiver.

extent

Answer a Point whose coordinates are the width and height of the receiver.

extent: aPoint

Change the receiver width and height to the coordinates of aPoint.

foreColor

Answer the foreground color, white.

foreColor: aColor

Change the foreground color to aColor. For Form, do nothing.

fromDisplay

Copy the receiver contents from the display screen.

fromDisplay: aRectangle

Copy the receiver contents from aRectangle area of the display screen.

height

Answer the height of the receiver.

magnify: aRectangle **by:** scale

Answer a form containing the image of aRectangle in the receiver magnified by scale whose x is the horizontal magnifying factor and y the vertical factor.

offset

Answer the offset of the receiver.

offset: aPoint

Change the offset of the receiver to aPoint.

outputToPrinter

Output the contents of the receiver to the printer sideways (landscape).

outputToPrinterUpright

Output the contents of the receiver to the printer (with 8 pins) upright (portrait).

reverse

Reverse the bit map of the receiver.

width

Answer the width of the receiver.

width: wInteger height: hInteger

Change the receiver width to wInteger and height to hInteger, and allocate its bitmap with the appropriate size.

width: wInteger height: hInteger initialByte: aByte

Change the receiver width to wInteger and height to hInteger, and initialize every byte in the bitmap to aByte.

Fraction

Class Fraction defines the protocol to perform arithmetic operations on rational numbers. A Fraction consists of a numerator denominator pair each of which is an integer so that no precision is lost during computations.

Inherits From: Number Magnitude Object

Inherited By: (None)

Named Instance Variables:

denominator

Contains an integer representing the denominator.

numerator

Contains an integer representing the numerator.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

numerator: n denominator: d

Answer an instance of class Fraction and initialize both numerator and denominator instance variables to n and d respectively.

Instance Methods:

*** aNumber**

Answer the result of multiplying the receiver by aNumber.

+ aNumber

Answer sum of the receiver and aNumber.

- aNumber

Answer the difference between the receiver and aNumber.

/ aNumber

Answer the result of dividing the receiver by aNumber.

// aNumber

Answer the integer quotient after dividing the receiver by aNumber with truncation towards negative infinity.

< aNumber

Answer true if the receiver is less than aNumber, else answer false.

<= aNumber

Answer true if the receiver is less than or equal to aNumber, else answer false.

= aNumber

Answer true if the receiver is equal to aNumber, else answer false.

> aNumber

Answer true if the receiver is greater than aNumber, else answer false.

>= aNumber

Answer true if the receiver is greater than or equal to aNumber, else answer false.

\\" aNumber

Answer the integer remainder after dividing the receiver by aNumber with truncation towards negative infinity.

asFloat

Answer the receiver as a floating point number.

denominator

Answer the denominator of the receiver.

hash

Answer the integer hash value for the receiver.

negated

Answer an instance of class Fraction which is the negative of the receiver.

numerator

Answer the numerator of the receiver.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

reciprocal

Answer the reciprocal of the receiver by dividing the denominator by the numerator.

truncated

Answer the receiver as a kind of Integer truncating the fraction part.

GraphDispatcher

A GraphDispatcher handles the user input directed to a GraphPane. The input can be either from the keyboard or from the mouse.

Inherits From: Dispatcher Object

Inherited By: (None)

Named Instance Variables:

active
 (From class Dispatcher)
pane
 (From class Dispatcher)

Class Variables:

WindowActivateKey
 (From class Dispatcher)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods: (None)

Instance Methods: (All private)

GraphPane

A GraphPane allows generalized graphic drawing in the pane. Each GraphPane is associated with a Form which contains a copy of the bitmap image shown in the pane so, that the pane can recover its contents after being obscured by other windows.

Inherits From: SubPane Pane Object

Inherited By: (None)

Named Instance Variables:

changeSelector
 (From class SubPane)

curFont

(From class Pane)

dispatcher

(From class Pane)

formHolder

Contains a Form with the image in the pane.

frame

(From class Pane)

framingBlock

(From class Pane)

margin

(From class SubPane)

model

(From class Pane)

name

(From class SubPane)

paneMenuSelector

(From class Pane)

paneScanner

(From class Pane)

scrollBar

(From class SubPane)

selection

Contains a Point which is the position on the screen where the last selection is made.

subpanes

(From class Pane)

superpane

(From class Pane)

topCorner

(From class SubPane)

Class Variables:

WindowClip

(From class Pane)

ZoomedPane

(From class Pane)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:

notifier: labelString content: aString at: aPoint

Pop up a window at aPoint with labelString as its label and aString magnified as its content.

notifier: labelString

content: aString **at:** aPoint **menu:** aMenu

Pop up a window at aPoint with labelString as its label and magnified aString as its content. Answer the selected symbol on aMenu which is automatically popped up for the notifier window.

Instance Methods:

activatePane

Mark the dispatcher of the receiver pane as active and inform its model.

charsInColumn

Answer the receiver frame height in characters.

close

Close the pane.

deactivatePane

Mark the receiver pane dispatcher as inactive and inform the model.

defaultDispatcherClass

Answer the default dispatcher of a GraphPane.

form

Answer the backup form for the receiver.

form: aForm

Change the backup form to aForm.

reframe: aRectangle

Change the frame of the receiver pane to aRectangle.

saveGraph

Save the screen image to the backup form.

selectAtCursor

Change the selection to the current cursor position and inform the model.

selection: aPoint

Change the selection to aPoint.

showWindow

Draw the borders of the pane and copy the backup form to the pane.

topCorner

Answer a Point which represents the current position of the pane on the backup form.

totalLength

Answer the height of the form.

update

Refresh the screen.

HomeContext

A HomeContext is used to contain method temporaries and arguments and to describe blocks of code (enclosed in square brackets). They are the objects to which value, value:, value:value: messages are sent to start the block evaluations.

Inherits From: **Context Object**

Inherited By: **(None)**

This class contains indexed instance variables.

Named Instance Variables:

blockArgumentCount
 (From class Context)

frameOffset
 Contains an integer offset of the associated stack frame.

homeContext
 (From class Context)

method
 Contains the compiled method in which the block appears.

receiver
 Contains the receiver for the method containing the block.

reserved
 Reserved for future use.

startPC
 (From class Context)

Class Variables:

(None)

Pool Dictionaries: **(None)**

Class Methods: **(None)**

Instance Methods: **(All private)**

Icon

An Icon is a graphical shape displayed on the screen. Its purpose is to provide a graphical representation of an object and to respond to mouse clicks for the object.

Inherits From: **Object**

Inherited By: **(None)**

Named Instance Variables:**form**

Contains a form representing the icon image.

hideFlag

Contains true if icon should not be shown, else false.

name

Contains a Symbol which denotes the message to be performed when the icon is selected.

origin

Contains a Point defining the location of the icon on the screen.

Class Variables:

(None)

Pool Dictionaries: (None)**Class Methods:****new**

Answer a new Icon.

Instance Methods:**containsPoint: aPoint**

Answer true if the icon contains aPoint, else answer false.

display

Display the icon on screen.

form

Answer the form of the icon.

form: aForm

Set the form of the icon to aForm.

frame

Answer the Rectangle containing the icon.

hide

Mark the icon as hidden.

isHidden

Answer true if the icon is hidden, else false.

name

Answer the name of the icon.

name: anObject

Set the name of the icon to anObject.

origin: aPoint

Set the origin of the icon to aPoint.

show

Mark the icon as visible.

width

Answer the width of the icon.

IdentityDictionary

An IdentityDictionary is a collection of key/value pairs of objects. The keys in an IdentityDictionary are unique, whereas the values may be duplicated. It can be searched either by key or by value. Key searches use hashing for efficiency. Elements may be entered into and extracted from an IdentityDictionary either as a pair of objects (e.g., at:put:) or as an Association (e.g., add:). Internally, an IdentityDictionary stores the key/value pairs in successive elements of the contents array whereas a Dictionary stores the key/value pairs as a set of associations. For this class, two keys are equal when they are actually the same object.

Inherits From: **Dictionary Set Collection Object**

Inherited By: **MethodDictionary**

Named Instance Variables:

contents

(From class Set)

elementCount

(From class Set)

Class Variables:

(None)

Pool Dictionaries: **(None)**

Class Methods:

new

Answer a new IdentityDictionary.

new: anInteger

Create a new instance with an initial capacity of anInteger elements. This method reports an error since the size of an idendity dictionary must be a power of 2.

Instance Methods:

add: anAssociation

Answer anAssociation. Add anAssociation to the receiver.

associationAt: aKey ifAbsent: aBlock

Answer an Association, with aKey and its corresponding value if aKey exists in the receiver, else evaluate aBlock (with no arguments).

associationsDo: aBlock

Answer the receiver. For each key/value pair in the receiver, evaluate aBlock with that pair as the argument.

at: aKey

Answer the value of the key/value pair whose key equals aKey from the receiver. If not found, report an error.

at: aKey ifAbsent: aBlock

Answer the value of the key/value pair whose key equals aKey from the receiver. If not found, evaluate aBlock (with no arguments).

at: aKey put: anObject

Answer anObject. If aKey exists in the receiver, replace the corresponding value with anObject, else add the aKey/anObject pair to the receiver.

do: aBlock

Answer the receiver. For each value in the receiver, evaluate aBlock with that value as the argument.

includesKey: aKey

Answer true if the receiver contains aKey, else answer false.

keyAtValue: anObject ifAbsent: aBlock

Answer the key in the receiver whose paired value equals anObject. If not found, evaluate aBlock (with no arguments).

keys

Answer a Set containing all the keys in the receiver.

removeKey: aKey ifAbsent: aBlock

Answer aKey. Remove the key/value pair whose key is aKey from the receiver. If aKey is not in the receiver, evaluate aBlock (with no arguments).

values

Answer a Bag containing all the values of the key/value pairs in the receiver dictionary.

IndexedCollection

Class IndexedCollection is an abstract class providing the common protocol for all the indexable collection subclasses. It includes methods to concatenate elements between collections, to replace elements of one collection with another, to iterate over the collection and perform some block of code on each element. Indexable collections can be accessed using integer indices.

Inherits From:

Collection Object

Inherited By:	Array Bitmap ByteArray CompiledMethod FileHandle FixedSizeCollection Interval OrderedCollection Process SortedCollection String Symbol
Named Instance Variables: (None)	
Class Variables:	
(None)	
Pool Dictionaries:	
(None)	
Class Methods:	
(None)	
Instance Methods:	
, aCollection	Answer a new collection containing the elements of the receiver followed by the elements of aCollection.
= aCollection	Answer true if the elements contained by the receiver are equal to the elements contained by the argument aCollection.
atAll: aCollection put: anObject	Answer the receiver after replacing those elements, indexed by the indices contained in aCollection, with anObject.
atAllPut: anObject	Answer the receiver after each element has been replaced with anObject.
copyFrom: start to: stop	Answer a new collection containing the elements of the receiver indexed from start through stop.
copyReplaceFrom: start to: stop with: aCollection	Answer a new collection containing a copy of the receiver with the elements at index positions from start through stop replaced with the elements of aCollection.
copyWith: anObject	Answer a copy of the receiver with anObject added to it as an element.
copyWithout: anObject	Answer a copy of the receiver excluding the first element that equals anObject, if any.
do: aBlock	Answer the receiver. For each element in the receiver, evaluate aBlock with that element as the argument.
findFirst: aBlock	Answer the index of the first element of the receiver that causes aBlock to evaluate to true (with that element as the argument). If no such element is found, report an error.

findLast: aBlock

Answer the index of the last element of the receiver that causes aBlock to evaluate to true (with that element as the argument). If no such element is found, report an error.

first

Answer the first element of the receiver. Report an error if the receiver has no elements.

grow

Answer the receiver expanded in size to accomodate more elements.

includes: anObject

Answer true if the receiver contains an element equal to anObject, else answer false.

indexOf: anObject

Answer the index position of the element equal to anObject in the receiver. If no such element is found, answer zero.

indexOf: anObject ifAbsent: aBlock

Answer the index position of the element equal to anObject in the receiver. If no such element is found, evaluate aBlock (without any arguments).

last

Answer the last element of the receiver. Report an error if the receiver has no elements.

replaceFrom: start to: stop with: aCollection

Answer the receiver. Replace the elements of the receiver at index positions start through stop, with the elements of aCollection. The number of elements being replaced must be the same as the number of elements in aCollection, else report an error.

replaceFrom: start**to: stop with: aCollection startingAt: repStart**

Replace the elements of the receiver at index positions start through stop with consecutive elements of aCollection beginning at index position repStart. Answer the receiver.

replaceFrom: start to: stop withObject: anObject

Replace each of the elements of the receiver at index positions start through stop with anObject. Answer anObject.

reversed

Answer a new object containing the elements of the receiver in reverse order.

reverseDo: aBlock

For each element in the receiver, starting with the last element, evaluate aBlock with that element as the argument.

shallowCopy

Answer a copy of the receiver which shares the receiver elements.

size

Answer the number of elements of the receiver.

with: aCollection do: aBlock

For each pair of elements (the first from the receiver and the second from aCollection), evaluate aBlock with those elements as the arguments. The receiver and aCollection must contain the same number of elements, else report an error.

InputEvent

An InputEvent reads all keyboard and mouse events. The global variable CurrentEvent contains the instance of InputEvent used by the environment. Events are requested by using the message `getNextEvent` which waits on the `KeyboardSemaphore`. The `KeyboardSemaphore` is signaled by keyboard and mouse interrupts.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

button

Contains a SmallInteger corresponding to the mouse button currently depressed.
0 = no button, 1 = left button, 2 = right button, 3 = middle button. Shift key state at time of button press is indicated by adding 4 if shift key was depressed.

type

Contains a Symbol describing the event type last read by the primitive. This corresponds to the symbol returned by the `getNextEvent` message.

typeArray

Contains an Array of 5 symbols used by the get next event primitive to set the type of the event. The 5 primitive events and the corresponding event types are:
1 = characterInput, 2 = functionInput, 3 = mouseMove, 4 = mouseButton,
5 = nullEvent.

value

Contains an object which is the value of the event. For mouse events it is the button involved. For keyboard events it is the character or scan code of the key depressed.

x

Contains the x-coordinate of the mouse at the time of the event as a SmallInteger.

y

Contains the y-coordinate of the mouse at the time of the event as a SmallInteger.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:**new**

Answer a new InputEvent.

Instance Methods:**nextEvent**

Answer the next event from the terminal (keyboard or mouse).

type

Answer the event type of the receiver.

type: aSymbol

Set the event type of the receiver to aSymbol.

value

Answer the event value of the receiver.

Inspector

Class Inspector implements a window on an object which allows the instance variables to be viewed and changed for that object. The window consists of two panes. The left pane contains the names (for the named instance variables) and/or numbers (for the indexed instance variables). The right pane contains the ASCII representation of the value of the selected instance variable. The left pane menu allows the opening of a new inspector on the selected instance variable. The right pane menu has all the text editing functions. The 'save' function replaces the value of the selected instance variable by the evaluated pane contents.

Inherits From:

Object

Inherited By:

Debugger DictionaryInspector

Named Instance Variables:**instIndex**

Contains the index of the selected entry in the list pane. If no entry is selected, instIndex contains 1 representing self (the object being inspected).

instList

Contains an OrderedCollection of strings to be displayed in the list pane which are the names and/or numbers of the inspected object.

instPane

Contains the ListPane which displays the list of inspected object instance variable names and/or numbers.

object

Contains the inspected object.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

openOn: anObject

Open an inspector window on anObject. Define the pane sizes and behavior, and schedule the window.

Integer

Class Integer is an abstract class used for comparing, counting, and measuring instances of its subclasses representing integral numbers. The precision of integral numbers is virtually infinite (the integer bit representation must be less than 64K bytes).

Inherits From: Number Magnitude Object

Inherited By: LargeNegativeInteger LargePositiveInteger SmallInteger

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

*** aNumber**

Answer the result of multiplying the receiver by aNumber.

+ aNumber

Answer the sum of the receiver and aNumber.

- aNumber

Answer the difference between the receiver and aNumber.

/ aNumber

Answer the result of dividing the receiver by aNumber.

// aNumber

Answer the quotient of dividing the receiver by aNumber with truncation towards negative infinity.

< aNumber

Answer true if the receiver is less than aNumber, else answer false.

<= aNumber

Answer true if the receiver is less than or equal to aNumber, else answer false.

= aNumber

Answer true if the receiver is equal to aNumber, else answer false.

> aNumber

Answer true if the receiver is greater than aNumber, else answer false.

>= aNumber

Answer true if the receiver is greater than or equal to aNumber, else answer false.

\\" aNumber

Answer the integer remainder after dividing the receiver by aNumber with truncation towards negative infinity.

asCharacter

Answer the character whose ASCII encoding matches the value of the receiver.

asFloat

Answer the floating point representation of the receiver.

basicHash

Answer the positive integer hash value for the receiver.

bitAnd: anInteger

Answer an Integer representing the receiver bits ANDed with the argument anInteger.

bitAt: anInteger

Answer 0 if the bit at index position anInteger in the receiver is 0, else answer 1.

bitInvert

Answer an integer whose bit values are the inverse of the bit values of the receiver.

bitOr: anInteger

Answer an Integer representing the receiver bits ORed with the argument anInteger.

bitShift: anInteger

Answer an integer which is the receiver shifted left anInteger number of bit positions if anInteger is positive, or shifted right for anInteger negated number of bit positions if anInteger is negative.

bitXor: anInteger

Answer the receiver bit XORed with the argument anInteger.

deepCopy

Answer a copy of the receiver with shallow copies of each instance variable. Because integers cannot be changed, answer the receiver.

factorial

Answer the factorial of the receiver.

gcd: anInteger

Answer the greatest common divisor between the receiver and anInteger.

hash

Answer the positive integer hash value for the receiver.

lcm: anInteger

Answer the least common multiple between the receiver and anInteger.

negated

Answer the negative value of the receiver.

printOn: aStream

Append the ASCII representation (radix 10) of the receiver to aStream.

printOn: aStream **base:** anInteger

Append the ASCII representation of the receiver with radix b to aStream.

printPaddedTo: anInteger

Answer the string containing the ASCII representation of the receiver padded on the left with blanks to be at least anInteger characters.

quo: aNumber

Answer the integer quotient of the receiver divided by aNumber with truncation toward zero.

radix: anInteger

Answer a string which is the ASCII representation of the receiver with radix anInteger.

reciprocal

Answer one divided by the receiver.

rem: aNumber

Answer the integer remainder after dividing the receiver by aNumber with truncation towards zero.

rounded

Answer the receiver.

shallowCopy

Answer a copy of the receiver which shares the receiver instance variables. Because integers cannot change, answer the receiver.

timesRepeat: aBlock

Evaluate aBlock n number of times, where n is the receiver.

truncated

Answer the receiver.

 $\sim =$ aNumber

Answer true if the receiver is not equal to aNumber, else answer false.

Interval

An Interval is a collection used to represent mathematical progressions. It is characterized as having a first number, a limit for the last computed number, and an increment amount for computing the next number in the progression.

Inherits From: FixedSizeCollection IndexedCollection Collection Object

Inherited By: (None)

Named Instance Variables:

beginning

Contains the beginning number of the interval.

end

Contains the limit for the last computed number of the interval.

increment

Contains the increment amount to compute the next number from the previous number.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

from: beginningInteger to: endInteger

Answer an Interval from beginningInteger to endInteger incrementing by one.

from: beginningInteger

to: endInteger by: incrementInteger

Answer an Interval from beginningInteger to endInteger incrementing by incrementInteger.

Instance Methods:

at: anInteger

Answer the number at index position anInteger in the receiver interval.

at: anInteger put: aNumber

Replace the number in the receiver indexed by anInteger with the argument aNumber. This message is not valid for intervals since interval collections are implicitly defined (the elements are computed).

increment

Answer the increment of the receiver Interval.

size

Answer the number of elements of the receiver.

species

Answer class Array as the species of Interval.

LargeNegativeInteger

Class LargeNegativeInteger is used to define the data structure for instances of integral numbers less than -32767. The precision of these instances is virtually infinite (the integer bit representation must be less than 64K bytes).

Inherits From: Integer Number Magnitude Object

Inherited By: (None)

This class contains indexed byte values.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods: (None)

LargePositiveInteger

Class LargePositiveInteger is used to define the data structure for instances of integral numbers greater than 32767. The precision of these instances is virtually infinite (the integer bit representation must be less than 64K bytes).

Inherits From: Integer Number Magnitude Object

Inherited By: (None)

This class contains indexed byte values.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods: (None)

LCompiler

Class LCompiler is used for converting Prolog source code to compiled methods. There are no instances of this class because its behavior is entirely defined with class messages.

Inherits From: Compiler Object

Inherited By: (None)

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

compile: aString

in: aClass notifying: requestor ifFail: exceptionBlock

Compile the Prolog method aString in aClass. If the method compiles correctly, answer an Association whose key is the method selector and whose value is the compiled method. If not, send the messages: requestor compilerError: errorString at: position in: codeString for: aClass. exceptionBlock value.

evaluate: aString

in: aClass to: doitReceiver notifying: requestor ifFail: exceptionBlock

Compile the Prolog method: ('Doit ', aString) in aClass. If the method compiles correctly, answer: doitReceiver Doit. If not, send the messages: requestor compilerError: errorString at: position in: aString. exceptionBlock value. In any case remove the selector #Doit from aClass' method dictionary.

Instance Methods: (None)

ListPane

Class ListPane provides functions to display and scroll a portion of the data held by the pane. The data is represented as an indexed collection of strings. When one of the strings in the collection is selected, either the selected string or its index in the list is passed to the application model for further processing.

Inherits From: SubPane Pane Object

Inherited By: (None)

Named Instance Variables:

changeSelector

(From class SubPane)

curFont

(From class Pane)

currentLine

Contains the line index where the cursor is positioned.

dispatcher

(From class Pane)

frame

(From class Pane)

framingBlock

(From class Pane)

list

Contains the pane data which can be any IndexedCollection of strings.

margin

(From class SubPane)

model

(From class Pane)

name

(From class SubPane)

paneMenuSelector

(From class Pane)

paneScanner

(From class Pane)

returnIndex

Contains a Boolean. When true, it indicates that the index of the selected string should be passed to the application model when a selection is made. When false, it indicates that the string itself should be passed back.

scrollBar

(From class SubPane)

selection

Contains the line index of the selected string.

subpanes

(From class Pane)

superpane

(From class Pane)

topCorner

(From class SubPane)

Class Variables:

WindowClip

(From class Pane)

ZoomedPane

(From class Pane)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:

(None)

Instance Methods:**close**

Close the pane.

deactivatePane

Change visual cues to reflect an inactive pane and make the pane dispatcher inactive.

defaultDispatcherClass

Answer the default dispatcher.

restore

Refresh the list from the model and maintain the position in the list without selecting it.

restoreSelected

Refresh the list from the model and keep the old selection.

restoreSelected: anObject

Display the list with the line indicated by anObject selected. anObject is either the index into the list or a string with which the list is to be searched with.

restoreWithRefresh: aString

Refresh the list from the model and keep the line equal to aString showing and selected.

returnIndex: aBoolean

Set the returnIndex to aBoolean.

selection

Answer an Integer representing the index of the currently selected item.

selection: anInteger

Set selection to anInteger.

showSelection

Highlight the selected line.

showWindow

Display the receiver pane and the selection.

topCorner

Answer the topCorner.

topCorner: aPoint

Change topCorner to aPoint.

update

Refresh the list from the model and display it.

ListSelector

A ListSelector processes input for its associated ListPane. Valid input can be a cursor movement, scrolling command, menu request, or line selection.

Inherits From: ScrollDispatcher Dispatcher Object

Inherited By: (None)

Named Instance Variables:

active
(From class Dispatcher)
pane
(From class Dispatcher)

Class Variables:

PageScroll
(From class ScrollDispatcher)
WindowActivateKey
(From class Dispatcher)

Pool Dictionaries:

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods: (None)

Instance Methods: (All private)

Magnitude

Class Magnitude is an abstract class used for comparing, counting, and measuring instances of its subclasses.

Inherits From: Object

Inherited By: Association Character Date Float Fraction Integer
LargeNegativeInteger LargePositiveInteger Number
SmallInteger Time

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:**< aMagnitude**

Answer true if the receiver is less than aMagnitude, else answer false.

<= aMagnitude

Answer true if the receiver is less than or equal to aMagnitude, else answer false.

= aMagnitude

Answer true if the receiver is equal to aMagnitude, else answer false.

> aMagnitude

Answer true if the receiver is greater than aMagnitude, else answer false.

>= aMagnitude

Answer true if the receiver is greater than or equal to aMagnitude, else answer false.

between: min and: max

Answer true if the receiver is greater than or equal to min and less than or equal to max, else answer false.

hash

Answer the positive integer hash value for the receiver.

max: aMagnitude

Answer the receiver if it is greater than aMagnitude, else answer aMagnitude.

min: aMagnitude

Answer the receiver if it is less than aMagnitude, else answer aMagnitude.

Menu

Class Menu defines the protocol for an application to present a menu of items to the user, allow the selection of an item, and then take some action based on the selection. Therefore, to define a menu, two ingredients must be supplied -- a String of menu items (separated by line-feeds) to be shown to the user and an Array of action selectors to be invoked when its corresponding item gets selected. A Menu is usually created by the class message labels:lines:selectors:. It can be activated by either the popUpAt: or popUpAt:for: instance message.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:**currentLine**

Contains an integer representing the index of the line in the menu where the cursor is positioned.

frame

Contains the rectangle that the menu occupies on the screen.

hiddenArea

Contains a Form containing a copy of the display screen image underneath the popped up menu.

offset

Contains a Point describing the position of the top left corner of the menu.

popUpForm

Contains a Form with the image of the menu.

priorCursor

Contains a Point describing the position of the cursor prior to the pop-up of the menu.

selectors

Contains an Array of symbols representing the action selectors corresponding to the items in the menu.

Class Variables:

(None)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods:

colors: colorArray selectors: selectorArray

Answer a menu with colorArray for the items, selectorArray for actions.

labelArray: labelArray lines: lineArray selectors: selectorArray

Answer a menu with labelArray for the items, selectorArray for actions, and lines drawn under the item numbers contained in lineArray.

labels: aString lines: lineArray selectors: selectorArray

Answer a menu with aString for the items, selectorArray for actions, and lines drawn under the item numbers contained in lineArray.

message: aString

Display aString as a one line menu.

Instance Methods:

popUpAt: aPoint

Pop up menu at aPoint, give it control, and answer the user response or nil if no response.

popUpAt: aPoint for: anObject

Pop up menu at aPoint, give it control, and send response to anObject.

Message

Class Message defines a data structure with an Array of message arguments and a message selector to describe a Smalltalk message. When an undefined message is encountered during execution, the virtual machine passes an instance of class Message describing the undefined message to the method 'doesNotUnderstand:' in class Object which in turn displays an appropriate error message in the walkback window.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

arguments

Contains an Array which contains the message arguments.

selector

Contains the message selector.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

arguments

Answer the arguments array for the message.

arguments: anArray

Set the arguments array for the message.

selector

Answer the message selector.

selector: aSymbol

Set the message selector.

MetaClass

Class MetaClass is the class of all metaclasses (e.g., of Array). It contains the common protocol for creating classes. Every metaclass has exactly one instance which is the class of the same name (e.g., Point is the only instance of Point class). The metaclass contains the class methods while class contains the instance methods. Metaclasses are referred to by sending the message 'class' to the class.

Inherits From: Behavior Object

Inherited By: (None)

Named Instance Variables:

comment
 (From class Behavior)
dictionaryArray
 (From class Behavior)
instances
 (From class Behavior)
name
 (From class Behavior)
structure
 (From class Behavior)
subclasses
 (From class Behavior)
superClass
 (From class Behavior)

Class Variables:

InstIndexedBit
 (From class Behavior)
InstNumberMask
 (From class Behavior)
InstPointerBit
 (From class Behavior)

Pool Dictionaries: (None)

Class Methods: (All private)

Instance Methods:

classPool
 Answer the pool dictionary of the only instance (a class) of the receiver (a metaclass).
classVarNames
 Answer a Set of the class variable names defined in the receiver.
name
 Answer a String containing the receiver name.
sharedPools
 Answer an Array of symbols of pool dictionary names referred to by the receiver.

MethodBrowser

A MethodBrowser is a window used for browsing a collection of related methods, such as senders or implementors of a message. It can also be used to edit the browsed methods.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

highlightLiteral

Contains an object (usually a Symbol) whose first reference in the source should be highlighted.

label

Contains the window label String.

methodPane

Contains the TextPane which contains the method source.

methods

Contains an OrderedCollection of methods being browsed.

positions

Contains information for highlighting the method source.

selectedMethod

Contains the method selected in the list pane.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

label: aString

Set the window label to aString.

literal: anObject

Request highlighting of source code that refers to anObject.

openOn: aMethodCollection

Create a method browser window on aMethodCollection

MethodDictionary

A MethodDictionary is a special kind of identity dictionary used to describe the compiled methods for each class. For each dictionary entry, the key is a message selector symbol and the associated value is a compiled method. Any updates to an instance of this class cause the method cache to be flushed, so that a subsequent method lookup will use the updated method dictionaries.

Inherits From: IdentityDictionary Dictionary Set Collection Object

Inherited By: (None)

Named Instance Variables:

contents

(From class Set)

elementCount

(From class Set)

reserved

Reserved for future use.

Class Variables:

Removing

Contains a Boolean to indicate whether or not a selector is being removed from the method dictionary.

Pool Dictionaries: (None)

Class Methods: (All private)

Instance Methods:

add: anAssociation

Answer anAssociation. Add anAssociation to the receiver. Flush the method cache in case an old method has changed.

at: aSymbol put: aMethod

Answer aMethod. Enter aSymbol and aMethod as a key/value pair in the receiver. Flush the method cache in case an old method has changed.

removeKey: aSymbol ifAbsent: aBlock

Answer aSymbol. Remove entry with key aSymbol from the receiver. If aSymbol is not a key of the receiver, evaluate aBlock (with no arguments). Flush the method cache.

NoMouseCursor

An instance of NoMouseCursor contains the bit pattern needed to display a cursor shape. In addition, it contains the methods for managing the moving, hiding, and displaying of the cursor. This class is used when there is no mouse driver loaded in memory. Thus the displaying and hiding of the cursor is done with Smalltalk code.

Inherits From: CursorManager Object

Inherited By: (None)

Named Instance Variables:

aBitBlt

Contains a BitBlt which displays the cursor by transferring from the image Form and hides the cursor by transferring from the under Form.

hotSpot

(From class CursorManager)

image

(From class CursorManager)

level

Contains an Integer. When it is less than zero, the cursor is hidden; otherwise, the cursor is shown. It is incremented by one everytime the cursor is displayed, and decremented by one everytime it is hidden.

oldPosition

Contains a Point which is a copy of the cursor's old position.

under

Contains a Form which is a copy of the screen image under the cursor. It is used to hide the cursor.

Class Variables:

ConditionShown

Contains a Boolean: if false it means that the cursor has been hidden during a previous conditional hide operation; if true it means that the cursor remained shown during the conditional hide operation.

NoMouse

(From class CursorManager)

Position

(From class CursorManager)

Pool Dictionaries: (None)

Class Methods:

new

Answer a NoMouseCursor, used when no mouse driver is present.

Instance Methods:**change**

Change Cursor to be the receiver.

display

Display the receiver on the screen.

hide

Hide the cursor from the screen.

hideX: x y: y width: width height: height

Hide the cursor if it moves within the rectangle of **x @ y extent: width @ height**.

offset: aPoint

Set the cursor position to aPoint. Answer the new position.

Number

Class Number is an abstract class used for comparing, counting, and measuring instances of its numerical subclasses.

Inherits From: Magnitude Object

Inherited By: Float Fraction Integer LargeNegativeInteger
LargePositiveInteger SmallInteger

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:**new**

Answer an instance of the receiver. This method reports an error.

new: argumentIgnored

Answer an instance of the receiver. This method reports an error.

Instance Methods:*** aNumber**

Answer the result of multiplying the receiver by aNumber.

+ aNumber

Answer the sum of the receiver and aNumber.

- aNumber

Answer the difference between the receiver and aNumber.

/ aNumber

Answer the result of dividing the receiver by aNumber.

// aNumber

Answer the integer result of dividing the receiver by aNumber with truncation towards negative infinity.

@ aNumber

Answer a point with the receiver as the x-coordinate and aNumber as the y-coordinate.

\ \ aNumber

Answer the integer remainder after dividing the receiver by aNumber with truncation towards negative infinity.

abs

Answer the absolute value of the receiver.

arcCos

Answer the arc-cosine, an angle in radians, of the receiver.

arcSin

Answer the arc-sine, an angle in radians, of the receiver.

arcTan

Answer the arc-tangent, an angle in radians, of the receiver.

ceiling

Answer the integer nearest the receiver towards positive infinity.

cos

Answer a Float which is the cosine of the receiver. The receiver is an angle measured in radians.

degreesToRadians

Answer the receiver converted from degrees to radians.

denominator

Answer the denominator of the receiver. Default is one which can be overridden by the subclasses.

even

Answer true if the integer part of the receiver is even, else answer false.

exp

Answer a Float which is the exponential of the receiver.

floor

Answer the integer nearest the receiver truncating towards negative infinity.

integerCos

Answer the integer cosine of the receiver angle, measured in degrees, scaled by 100.

integerSin

Answer the integer sine of the receiver angle, measured in degrees, scaled by 100.

ln

Answer a Float which is the natural log of the receiver.

log: aNumber

Answer a Float which is the log base aNumber of the receiver.

negated

Answer the negation of the receiver.

negative

Answer true if the receiver is less than zero, else answer false.

numerator

Answer the numerator of the receiver. Default is the receiver which can be overridden by the subclasses.

odd

Answer true if the integer part of the receiver is odd, else answer false.

positive

Answer true if the receiver is greater than or equal to zero, else answer false.

printFraction: numberFractionDigits

Answer a string, the ASCII representation of the receiver truncated to numberFractionDigits decimal places.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

printRounded: numberFractionDigits

Answer a string, the ASCII representation of the receiver rounded to numberFractionDigits decimal places.

quo: aNumber

Answer the integer quotient with truncation toward zero.

radiansToDegrees

Answer the receiver converted from radians to degrees.

raisedTo: aNumber

Answer a Float which is the receiver raised to the power of aNumber.

raisedToInteger: anInteger

Answer the receiver raised to the power of anInteger.

reciprocal

Answer one divided by the receiver.

rem: aNumber

Answer the integer remainder after dividing the receiver by aNumber with truncation towards zero.

rounded

Answer the nearest integer to the receiver.

roundTo: aNumber

Answer the receiver rounded to the nearest multiple of aNumber.

sign

Answer 1 if the receiver is greater than zero, answer -1 if the receiver is less than zero, else answer zero.

sin

Answer a Float which is the sine of the receiver. The receiver is an angle measured in radians.

sqrt

Answer a Float which is the square root of the receiver.

squared

Answer the receiver multiplied by the receiver.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reconstructed.

strictlyPositive

Answer true if the receiver is greater than zero, else answer false.

tan

Answer a Float which is the tangent of the receiver. The receiver is an angle measured in radians.

timesTwoPower: anInteger

Answer the result of multiplying the receiver by 2 to the exponent anInteger.

to: aNumber

Answer an Interval for the numbers between the receiver and the argument aNumber where each number is the previous number plus 1.

to: sNumber by: iNumber

Answer an Interval for the numbers between the receiver and the argument sNumber where each number is the previous number plus the argument iNumber.

to: sNumber by: iNumber do: aBlock

Evaluate the one argument block aBlock for the numbers between the receiver and the argument sNumber where each number is the previous number plus the argument iNumber.

to: aNumber do: aBlock

Evaluate the one argument block aBlock for the numbers between the receiver and the argument aNumber where each number is the previous number plus 1.

truncateTo: aNumber

Answer the receiver truncated (towards zero) to the nearest multiple of aNumber.

Object

Class Object is the superclass of all other classes and defines the protocol common to all objects. It defines the default behavior for displaying, comparing, copying, hashing, inspecting objects, evaluating blocks, accessing indexed instance variables and error handling. It includes capabilities to maintain dependency relationships between objects and to broadcast messages from an object to its dependents. It also provides the entry point for interrupt handling.

Inherits From: (None)

Inherited By: (All classes)

Named Instance Variables: (None)

Class Variables:

Dependents

Contains an IdentityDictionary representing object dependents. For each entry, the key is an object and the associated value is the set of all other objects which are dependent on the key object. It is initialized to an empty dictionary.

RecursionInError

Contains a Boolean indicating whether an error has occurred while reporting an error in a walkback window. If so, the destination of the error output switches from the walkback window to the display screen. It is initialized to false.

RecursiveSet

Contains a Set of objects whose display has started but not finished. It is used for gracefully detecting the display of self referencing data structures (e.g., show it from TextPane menu).

Pool Dictionaries: (None)

Class Methods:

initDependents

Initialize the Dependents dictionary to empty.

initialize

Initialize the class variables for detecting recursive data structures.

Instance Methods:

= anObject

This is the default equality test. Answer true if the receiver and anObject are the same object, else answer false.

== anObject

Answer true if the receiver and anObject are the same object, else answer false.

addDependent: anObject

Add anObject to the class variable Dependents of class Object.

allDependents

Answer a Set containing all the dependents of the receiver.

allReferences

Answer an Array of all of the references to the receiver.

at: anInteger

Answer the object in the receiver at index position anInteger. If the receiver does not have indexed instance variables, or if anInteger is greater than the number of indexed instance variables, report an error.

at: anInteger put: anObject

Answer anObject. Replace the object in the receiver at index position anInteger with anObject. If the receiver does not have indexed instance variables, or if anInteger is greater than the number of indexed instance variables, report an error.

basicAt: anInteger

Answer the object in the receiver at index position anInteger. If the receiver does not have indexed instance variables, or if anInteger is greater than the number of indexed instance variables, report an error.

basicAt: anInteger put: anObject

Answer anObject. Replace the object in the receiver at index position anInteger with anObject. If the receiver does not have indexed instance variables, or if anInteger is greater than the number of indexed instance variables, report an error.

basicHash

Answer the integer hash based on its hash field.

basicSize

Answer the number of indexed instance variables in the receiver.

become: anObject

The receiver takes on the identity of anObject. All the objects that referenced the receiver will now point to anObject.

broadcast: aSymbol

Send the argument aSymbol as a unary message to all of the receiver's dependents.

broadcast: aSymbol with: anObject

Send the argument aSymbol as a keyword message with argument anObject to all of the receiver's dependents.

changed

The receiver changed in some general way. Inform all dependents by sending each dependent an update message.

changed: aParameter

Something has changed relateed to the dependents of the receiver. Send the 'update: aParameter' message to all the dependents.

changed: firstParameter with: secondParameter

Something has changed relateed to the dependents of the receiver. Send the 'update: firstParameter with: secondParameter' message to all the dependents.

changed: firstParameter

with: secondParameter **with:** thirdParameter

 Something has changed related to the dependents of the receiver. Send the 'update: firstParameter with: secondParameter' message to all the dependents.

class

 Answer the class of the receiver.

copy

 Answer a shallow copy of the receiver.

deepCopy

 Answer a copy of the receiver with shallow copies of each instance variable.

dependents

 Answer a collection of all dependents of the receiver.

dependsOn: anObject

 Add the receiver to anObject's collection of dependents.

doesNotUnderstand: aMessage

 Initiate a walkback because a message was sent which is not understood, i.e., there is no matching method.

error: aString

 Create a walkback window describing an error condition with the error message aString in the window label.

halt

 Initiate a walkback with 'halt encountered' message for debugging.

hash

 Answer the integer hash value of the receiver. This is the default implementation which uses the object hash value assigned at the creation time.

hash: anInteger

 Set the hash value of the receiver to anInteger.

implementedBySubclass

 Initiate a walkback because a subclass doesn't implement a message that it should.

inspect

 Open an inspector window on the receiver.

invalidMessage

 Initiate walkback because inappropriate message was sent to the receiver.

isKindOf: aClass

 Answer true if receiver is an instance of aClass or one of its subclasses, else answer false.

isMemberOf: aClass

 Answer true if the receiver is an instance of aClass, else answer false.

isNil

 Answer true if the receiver is the object nil, else answer false.

notNil

Answer true if the receiver is not the object nil, else answer false.

perform: aSymbol

Answer the result of sending a unary message to the receiver with selector aSymbol.
Report an error if the number of arguments expected by the selector is not zero.

perform: aSymbol with: anObject

Answer the result of sending a binary message to the receiver with selector aSymbol
and argument anObject. Report an error if the number of arguments expected by
the selector is not one.

perform: aSymbol with: firstObject with: secondObject

Answer the result of sending a keyword message to the receiver with selector
aSymbol and arguments firstObject and secondObject. Report an error if the
number of arguments expected by the selector is not two.

perform: aSymbol**with: firstObject with: secondObject with: thirdObject**

Answer the result of sending a keyword message to the receiver with selector
aSymbol and arguments firstObject, secondObject and thirdObject. Report an
error if the number of arguments expected by the selector is not three.

perform: aSymbol withArguments: anArray

Answer the result of sending a message to the receiver with selector aSymbol and
arguments the elements of anArray. Report an error if the number of arguments
expected by the selector is not equal to anArray size.

printOn: aStream

Append the ASCII representation of the receiver to aStream. This is the default
implementation which prints 'a' ('an') followed by the receiver class name.

printString

Answer a String that is an ASCII representation of the receiver.

release

Discard all dependents of the receiver, if any.

respondsTo: aSymbol

Answer true if the receiver class or one of its superclasses implements a method
with selector equal to aSymbol.

shallowCopy

Answer a copy of the receiver which shares the receiver instance variables.

size

Answer the number of indexed instance variables in the receiver.

species

Answer a class which is similar to (or the same as) the receiver class which can be
used for containing derived copies of the receiver.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the
receiver can be reinstated.

storeString

Answer the receiver represented as a String from which it can be reconstructed.

update: aParameter

An object on whom the receiver is dependent has changed. The receiver updates its status accordingly (the default behavior is to do nothing). The argument aParameter usually identifies the kind of update.

update: firstParameter with: secondParameter

An object on whom the receiver is dependent has changed. The receiver updates its status accordingly (the default behavior is to do nothing). The argument firstParameter usually identifies the kind of update and the secondParameter is a unary message defined in the receiver protocol.

update: firstParameter**with: secondParameter with: thirdParameter**

An object on whom the receiver is dependent has changed. The receiver updates its status accordingly (the default behavior is to do nothing). The argument firstParameter usually identifies the kind of update and the secondParameter is a unary message defined in the receiver protocol.

yourself

Answer the receiver.

~= anObject

Answer true if the receiver and anObject do not compare equal (using =), else answer false.

~~ anObject

Answer true if the receiver and anObject are not the same object, else answer false.

OrderedCollection

An OrderedCollection can be used like a dynamic array, stack or queue. Unlike fixed size collections, an OrderedCollection can grow to accomodate more elements if the original collection is not big enough.

Inherits From: IndexedCollection Collection Object

Inherited By: Process SortedCollection

Named Instance Variables:

contents

Contains an Array of objects included in the OrderedCollection.

endPosition

Contains the contents index position of the last element of the collection. All successive index positions (to contents size), are assumed to be empty.

startPosition

Contains the contents index position of the first element of the collection. All preceding index positions (to contents position 1), are assumed to be empty.

Class Variables:

(None)

Pool Dictionaries: (None)**Class Methods:****new**

Answer an instance of OrderedCollection capable of holding 12 elements initially.

new: anInteger

Answer an initialized instance of OrderedCollection capable of holding anInteger number of elements.

Instance Methods:**, aCollection**

Answer an OrderedCollection containing all the elements of the receiver followed by all the elements of aCollection.

add: anObject

Answer anObject. Add anObject after the last element of the receiver collection.

add: newObject after: oldObject

Answer newObject. Insert newObject immediately after the element oldObject in the receiver collection. If oldObject is not in the collection, report an error.

add: anObject afterIndex: anInteger

Answer anObject. Insert anObject at index position anInteger + 1 in the receiver collection. If anInteger is out of the collection bounds, report an error.

add: newObject before: oldObject

Answer newObject. Insert newObject immediately before the element oldObject in the receiver collection. If oldObject is not in the collection, report an error.

add: anObject beforeIndex: anInteger

Answer anObject. Insert anObject at index position anInteger - 1 in the receiver collection. If anInteger is out of the collection bounds, report an error.

addAllFirst: aCollection

Answer aCollection. Add all the elements contained in aCollection to the receiver before its first element.

addAllLast: aCollection

Answer aCollection. Add all the elements contained in aCollection to the receiver after its last element.

addFirst: anObject

Answer anObject. Add anObject before the first element of the receiver.

addLast: anObject

Answer anObject. Add anObject after the last element of the receiver.

after: anObject

Answer the element that immediately follows anObject in the receiver collection. If anObject is not an element of the receiver, report an error.

after: anObject ifNone: aBlock

Answer the element that immediately follows anObject in the receiver collection. If anObject is not an element of the receiver, aBlock is evaluated (with no arguments).

at: anInteger

Answer the element of the receiver at index position anInteger. If anInteger is an invalid index for the receiver collection, report an error.

at: anInteger put: anObject

Answer anObject. Replace the element of the receiver at index position anInteger with the anObject. If anInteger is an invalid index for the receiver collection, report an error.

before: anObject

Answer the element that immediately precedes anObject in the receiver collection. If anObject is not an element of the receiver, report an error.

before: anObject ifNone: aBlock

Answer the element that immediately precedes anObject in the receiver collection. If anObject is not an element of the receiver, aBlock is evaluated (with no arguments).

copyFrom: beginning to: end

Answer an OrderedCollection containing the elements of the receiver from index position beginning through index position end.

do: aBlock

Answer the receiver. For each element in the receiver, evaluate aBlock with that element as the argument.

includes: anObject

Answer true if the receiver contains an element equal to anObject, else answer false.

remove: anObject ifAbsent: aBlock

Answer anObject. Remove the element anObject from the receiver collection. If anObject is not an element of the receiver, aBlock is evaluated (with no arguments).

removeFirst

Remove and answer the first element of the receiver. If the collection is empty, report an error.

removeIndex: anInteger

Answer the receiver. Remove the element of the receiver at index position anInteger. If anInteger is an invalid index for the receiver, report an error.

removeLast

Remove and answer the last element of the receiver. If the collection is empty, report an error.

replaceFrom: start **to:** stop **with:** aCollection

Answer a new OrderedCollection containing the receiver whose elements at index position start through stop have been replaced by the elements of aCollection.

size

Answer the number of elements contained by the receiver collection.

Pane

Class Pane is an abstract class which provides the common protocol for all its subclasses. A pane is a subarea of a window. It is responsible for displaying a portion of its contents in a designated area on the display screen. It is one of the three major elements (pane, dispatcher, and application model) of a window application. Each pane is associated with one dispatcher. All the panes of a window are normally tied to one application model.

Inherits From: Object

Inherited By: GraphPane ListPane SubPane TextPane TopPane

Named Instance Variables:

curFont

Contains the current font used in the pane.

dispatcher

Contains the dispatcher associated with the pane object.

frame

Contains the Rectangle on the display screen into which the pane object can display its contents.

framingBlock

Contains a block of code which computes the frame rectangle of the pane based on an argument representing the pane's outer frame.

model

Contains the application model that controls this pane object.

paneMenuSelector

Contains a no-argument message selector which, when invoked, answers a Menu customized for the pane.

paneScanner

Contains a StringBlt which is responsible for all the output to the borders form (refer to class TopPane) and transfer from the borders form to the display screen.

subpanes

Contains an OrderedCollection of subpanes which are under the control of this Pane object.

superpane

Contains the parent pane that controls this pane object. Note that a TopPane has no superpane.

Class Variables:**WindowClip**

Contains the clipping Rectangle of the display screen to update following the move or a close of a window.

ZoomedPane

Contains the TextPane currently being zoomed.

Pool Dictionaries: (None)

Class Methods:**initWindowClip**

Reinitialize the clipping rectangle of redrawing windows to be the whole display screen.

new

Answer a new pane.

windowClip

Answer the clipping rectangle for redrawing windows.

windowClip: aRectangle

Set the clipping rectangle for redrawing windows to aRectangle.

Instance Methods:**activatePane**

Mark the dispatcher of the receiver pane as active.

border

Draw a border around the receiver frame.

border: aRectangle

Draw two lines around aRectangle. The outside one has the foreground color and the inside one the background color.

close

Close the subpanes and release their dependencies from the model.

cyclePane

Move the cursor to the pane next to the receiver. If none, home the cursor in the receiver pane.

deactivatePane

Mark the receiver pane dispatcher as inactive.

deactivateWindow

Mark the dispatcher of the receiver as inactive and change their visual cues to reflect an inactive window.

dispatcher

Answer the dispatcher for the pane.

dispatcher: aDispatcher

Make aDispatcher the dispatcher of the receiver pane and initialize it.

font

Answer a Font, the font currently associated with the receiver pane.

frame

Answer a Rectangle, the frame of the receiver.

hasCursor

Answer true if the pane contains the cursor, else answer false.

hasZoomedPane

Answer true if a textPane has been zoomed.

homeCursor

Move the cursor to the top left corner of the pane.

menu: aSymbol

Set the paneMenuSelector to the message selector contained in aSymbol which is used to generate the pane menu.

model

Answer the model for the receiver.

model: anObject

Set the model of the receiver to anObject and add the receiver as a dependent of the model.

paneScanner

Answer the CharacterScanner associated with the pane.

popUp: aMenu

Display aMenu at the cursor and perform the menu selection.

popUp: aMenu **at:** aPoint

Display aMenu at aPoint. If the user choice is nil, do nothing. If the model can respond to the choice, let it perform the choice. Else, let the dispatcher perform it.

release

Remove model dependency and disconnect the model from the pane.

superpane: aPane

Set the receiver superpane to aPane.

unzoom

Return zoomed pane to normal.

Pattern

An instance of Pattern contains a finite state pattern to be used to match against another object. The pattern itself and the object to be matched against can be any subclass of IndexedCollection. In addition, the Pattern class contains methods needed for performing the match. There are two ways to invoke the pattern matching. One is to pass the entire matching collection to the pattern. Another is to setup a loop and pass one element at a time from the matching collection to the Pattern and a prebuilt block will be executed each time a match occurs. The first way is more efficient when there is only one matching collection. The second way is normally used when there are several matching collections.

Inherits From: Object

Inherited By: WildPattern

Named Instance Variables:

fail

Contains an Array of integers corresponding to elements in the pattern collection. Each integer denotes the next element to go to when the current pattern element fails to match the element in the matching collection.

first

Contains the first element in the pattern collection. This is used to increase matching speed.

input

Contains the collection to be used as the pattern.

matchBlock

Contains a block of code to be executed when a match occurs using the second way of invoking the pattern match.

state

Contains an Integer denoting the current state of the pattern collection which is the index of the current pattern element being matched.

Class Variables:

WildcardChar

Contains a Character which when appears in the pattern collection will match zero or more elements in the matching collection.

Pool Dictionaries: (None)

Class Methods:

new: aString

Answer a new pattern with aString as the pattern to match.

wildcardChar

Answer the wild card character.

Instance Methods:

match: anObject

Compare anObject against the pattern. If anObject completes the matching of the pattern, evaluate the match block.

match: aCollection index: anInteger

Answer a Point representing the start and stop of the subcollection within aCollection that matches the receiver starting at index position anInteger. Answer nil if no match.

matchBlock: aBlock

Set the match block of the receiver to aBlock. This block will be evaluated when the pattern is fully matched.

reset

Reset the receiver to start matching at the beginning of the pattern.

Pen

This class extends the functions of BitBlt to provide a turtle graphics type of drawing interface. The source form serves as the nib of the pen, the mask form the color of the pen, and the destination form the canvas for drawing. It draws by repeatedly copying the masked bits from the source form to the destination form. For each copy made, the destination origin is moved according to the specified drawing pattern, e.g. a line or a circle.

Inherits From:

BitBlt Object

Inherited By:

Animation Commander

Named Instance Variables:

clipHeight

(From class BitBlt)

clipWidth

(From class BitBlt)

clipX

(From class BitBlt)

clipY

(From class BitBlt)

destForm

(From class BitBlt)

destX

(From class BitBlt)

destY

(From class BitBlt)

direction

Contains an Integer denoting the current drawing direction in degrees from 0 to 359.

downState

Contains a Boolean specifying whether the pen is down (true) on the canvas or lifted (false).

fractionX

Contains an Integer which is the hundredth fractional part of the x coordinate of the current pen location. The integral part is contained in variable destX.

fractionY

Contains an Integer which is the hundredth fractional part of the y coordinate of the current pen location. The integral part is contained in variable destY.

halftone

(From class BitBlt)

height

(From class BitBlt)

rule

(From class BitBlt)

sourceForm

(From class BitBlt)

sourceX

(From class BitBlt)

sourceY

(From class BitBlt)

width

(From class BitBlt)

Class Variables:

DoubleCenter

Contains a point representing the center of an ellipse multiplied by two.

Pool Dictionaries: (None)

Class Methods:

new

Answer a Pen with its instance variables initialized using Display as destination form.

new: aForm

Answer a Pen with its instance variables initialized using aForm as destination form.

Instance Methods:

black

Change the pen color to black.

bounce: anInteger

If the pen touches the clipping rectangle after moving for an increment of anInteger, change its direction so that it looks like it is bouncing off the wall.

centerText: aString font: aFont

Write aString whose center is at the destination origin using aFont.

changeNib: aForm

Change the source form (the nib) to aForm.

defaultNib: size

Change the size of the nib to size which can be either an Integer or a Point.

direction

Answer the current direction of the receiver pen in degrees from 0 to 359. East is degree 0, south is 90.

direction: anInteger

Set the direction to anInteger number of degrees.

down

Set the pen down.

dragon: anInteger

Draw a dragon pattern where anInteger is the recursion factor.

drawRectangle

Draw a single pixel rectangle on the inside of the destination rectangle.

ellipse: anInteger aspect: aFraction

Draw an ellipse with the pen position as its center, anInteger as half of the width, aFraction as the ratio of the ellipse height to width. The height will be adjusted by the global variable Aspect.

fillAt: aPoint

Color all pixels that are connected to aPoint and have the same color as that of aPoint with the pattern contained in the mask form.

frame

Answer the clipping rectangle of the pen.

frame: aRectangle

Set the clipping rectangle to aRectangle.

go: anInteger

Move the pen for the distance anInteger number of pixels in the current direction. The y-axis is adjusted by Aspect.

goto: aPoint

Move the pen to aPoint.

gray

Change the pen color to gray.

grid: anInteger

Draw a grid within the clipping rectangle where anInteger is the number of pixels between the lines.

home

Center the pen on the destination form.

location

Answer a Point, the current position of the pen.

mandala: sInteger diameter: dInteger

Draw a mandala with sInteger number of sides and dInteger as the diameter.

north

Set the direction of the pen to 270 degrees.

place: aPoint

Position the pen at aPoint.

polygon: lInteger sides: sInteger

Draw a polygon with sInteger number of sides where each is of length lInteger.

solidEllipse: anInteger aspect: aFraction

Draw an ellipse with anInteger as half the width and aFraction as the aspect ratio with the pen position as the center, and fill its insides with the color of the mask form.

spiral: anInteger angle: dInteger

Draw a spiral with anInteger number of lines where dInteger is the angle between two successive lines.

turn: anInteger

Change the direction of the pen anInteger number of degrees. anInteger can be either positive or negative.

up

Lift the pen up.

white

Change the color of the pen to white.

Point

A Point represents a position in two dimensions (e.g., a character's position in a form). It consists of a pair of numbers, x and y. By convention, x increases to the right and y increases downward.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

x

Contains a number representing the x-coordinate (column) of the point.

y

Contains a number representing the y-coordinate (row) of the point.

Class Variables:

(None)

Pool Dictionaries: (None)**Class Methods:** (None)**Instance Methods:***** scale**

Answer a new Point which is the product of the receiver and scale. Scale can be a number or a Point. If scale is a Point, the x-coordinates are multiplied and the y-coordinates are multiplied.

+ delta

Answer a new Point which is the sum of the receiver and delta. Delta can be a number or a Point. If delta is a Point, the x-coordinates are added and the y-coordinates are added.

- delta

Answer a new Point which is the difference of the receiver and delta. delta can be a number or a Point. If delta is a Point, the x-coordinates are subtracted and the y-coordinates are subtracted.

// scale

Answer a new Point which is the receiver Point divided by scale. Scale can be a number or a Point. If scale is a Point, the x-coordinates are divided and the y-coordinates are divided.

< aPoint

Answer true if the x and y coordinates of the receiver are less than the x and y coordinates of aPoint, respectively, else answer false.

<= aPoint

Answer true if the x and y coordinates of the receiver are less than or equal to the x and y coordinates of aPoint, respectively, else answer false.

= aPoint

Answer true if the x and y coordinates of the receiver are equal to the x and y coordinates of aPoint, respectively, else answer false.

> aPoint

Answer true if the x and y coordinates of the receiver are greater than the x and y coordinates of aPoint, respectively, else answer false.

>= aPoint

Answer true if the x and y coordinates of the receiver are greater than or equal to the x and y coordinates of aPoint, respectively, else answer false.

\ \ scale

Answer a new Point which is the integer remainder of the receiver Point divided by scale. Scale can be a number or a Point. If scale is a Point, the x-coordinates are divided and the y-coordinates are divided.

abs

Answer a Point with coordinates that are the absolute value of the x and y coordinates of the receiver.

between: aPoint and: bPoint

Answer true if the receiver is greater than or equal to aPoint and less than or equal to bPoint, else answer false.

corner: aPoint

Answer a Rectangle with origin equal to the receiver and corner equal to aPoint.

dotProduct: aPoint

Answer a number which is the sum of the product of the x-coordinates and the product of the y-coordinates of the receiver and aPoint.

extent: aPoint

Answer a Rectangle with origin equal to the receiver and extent equal to aPoint.

hash

Answer the integer hash value of the receiver.

isBefore: aPoint

Answer true if receiver is text-wise earlier than aPoint, else answer false.

max: aPoint

Answer a Point with the maximum of the x-coordinates and the maximum of the y-coordinates of the receiver and aPoint.

min: aPoint

Answer a Point with the minimum of the x-coordinates and the minimum of the y-coordinates of the receiver and aPoint.

moveBy: aPoint

Answer the receiver with its x-coordinate incremented by aPoint x and y-coordinate incremented by aPoint y.

negated

Answer a Point with the x and y coordinates of the receiver negated.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

rounded

Answer a Point which has the receiver coordinates rounded to integers.

transpose

Answer a Point with x-coordinate equal to the receiver's y-coordinate and y-coordinate equal to receiver's x-coordinate.

truncated

Answer a Point which has the receiver coordinates truncated to integers.

x

Answer the receiver's x-coordinate.

x: aNumber

Answer the receiver. Set the receiver's x-coordinate to aNumber.

y

Answer the receiver's y-coordinate.

y: aNumber

Answer the receiver. Set the receiver's y-coordinate to aNumber.

PointDispatcher

A PointDispatcher is used to interactively define or modify a rectangle on the screen. The rectangle can be moved about the screen, be expanded or shrunk in size. A PointDispatcher answers a Point if only move is allowed and a Rectangle if resizing is also allowed. It is largely used for moving or framing a window.

Inherits From: Dispatcher Object

Inherited By: (None)

Named Instance Variables:

active

(From class Dispatcher)

aPen

Contains a Pen to draw the outline of the rectangle.

cursorOffset

Contains a Point representing the offset between the display box origin and the cursor.

displayBox

Contains the Rectangle shown on the display screen.

minBoxExtent

Contains a Point describing the minimum width and height of the final rectangle.

moveOrSizeBox

Contains the Symbol #move, #frame, or #size. The Symbol #move means that the displayBox can only be moved but not resized. During a framing operation, the Symbol #frame is used to locate the origin and #size is used to define the corner of the displayBox.

pane

(From class Dispatcher)

returnBlock

Contains a one-argument block of code which, when executed, will exit the PointDispatcher and answer a Point or a Rectangle depending on whether it is a move or a frame operation.

Class Variables:

WindowActivateKey

(From class Dispatcher)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods:

cornerFromUserOfOrigin: aPoint minExtent: extent

Answer a Rectangle which resizes a rectangle with aPoint as the origin and extent as the minimum extent.

new

Answer a new initialized PointDispatcher.

pointFromUserDisplaying: aRectangle offset: aPoint

Display a rectangle of size box which may be moved by the user to the desired position. Answer the top left corner of the rectangle when the user selects a position. 'aPoint' is the offset between the cursor and the origin of aRectangle

Instance Methods:

drawBox: aRect

Draw a border around aRect.

Process

A Process is an object representing a sequence of Smalltalk computations. The computations are performed by objects sending messages to other objects and waiting for the results. The process describes the current execution state, including the stack of unanswered messages.

Inherits From: OrderedCollection IndexedCollection Collection Object

Inherited By: (None)

Named Instance Variables:

contents

(From class OrderedCollection)

debugger

Contains the debugger window associated with the process if it is being debugged, or nil if not debugged.

endPosition

(From class OrderedCollection)

frameBias

Contains an integer which when added to a hardware stack pointer converts it to a process object index.

interruptFrame

Contains an integer used by the 'resume:' primitive.

isUserIF

Contains true if the process is a user interface process, else false.

name

Contains a String representing the process name.

priority

Contains an integer representing the process priority.

runable

Contains true if the process is runable, else false.

sendFrame

Contains an integer frame number used to trigger the step interrupt.

startPosition

(From class OrderedCollection)

topFrame

Contains the hardware stack pointer for the stack frame at the top of the stack.

Class Variables:

(None)

Pool Dictionaries:

(None)

Class Methods:

breakpointInterrupt

Implement breakpoint interrupt.

controlBreakInterrupt

Initiate a control-break walkback.

copyStack

Answer a Process object containing the current stack contents.

dropSenderChain

Discard stacked message sends (sent but not answered) to outermost send, the input request loop.

enableInterrupts: aBoolean

Answer the previous interrupt enable state. Set the interrupt enable state to aBoolean.

interrupt: interruptNumber

Put an interrupt number in the virtual machine queue.

ioErrorInterrupt

Initiate a DOS critical error walkback.

keyboardInterrupt

Implement keyboard interrupt.

new

Answer a new Process.

overrunInterrupt

Initiate an interrupt queue overrun walkback.

queueWalkback: aString makeUserIF: ifBoolean resumable: resumeBoolean

Enter a walkback for current process in pending event queue. Create new user interface process if ifBoolean is true.

timerInterrupt

Implement the timer interrupt.

Instance Methods:

debugger

Answer the debugger associated with the receiver, or nil if none.

debugger: aDebugger

Set receiver's debugger to aDebugger.

isUserIF

Answer true if receiver is a user interface process.

makeUserIF

Make the receiver be the user interface process.

name

Answer the process name.

name: aString

Set the process name to aString.

priority

Answer an integer representing the receiver process priority.

priority: aNumber

Change the priority of the receiver process to aNumber.

resume

Resume the receiver process.

ProcessScheduler

Class ProcessScheduler provides the mechanism for scheduling process execution according to process priorities. There is a single instance of the class maintained in the global variable, Processor.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

readyProcesses

Contains an Array of OrderedCollections of ready to run processes. The array is indexed by the process priority.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:**fork: forkBlock**

Create a new process to execute forkBlock and schedule it at same priority as the active process.

fork: forkBlock at: aPriority

Create a new process to execute forkBlock and schedule it at priority priority.

highUserPriority

Answer the highest priority for a user process.

initialize

Initialize the receiver by discarding all processes and then creating a new user interface process.

lowUserPriority

Answer the lowest priority for a user process.

resume: aProcess

Add aProcess to the process scheduler's queue of ready processes. If aProcess has the highest priority, make it the current process.

schedule

Schedule the highest priority ready process, or if none, create the idle process. Called with interrupts disabled.

suspendActive

Suspend the active process and schedule the highest priority ready process, if any. Called with interrupts disabled and CurrentProcess already entered in proper waiting queue.

topPriority

Answer the highest allowable priority for system processes.

userPriority

Answer the priority of the user interface process.

yield

Give other processes at the priority of the currently running process a chance to run.

PromptEditor

A **PromptEditor** processes input for its associated **TextPane** in a **Prompter**. Its allowed input is more restrictive than a **TextEditor**. It will not give up control until the user either accepts or cancels the **Prompter**. The user response is accepted by either pressing the carriage-return key or selecting the accept item from the menu.

Inherits From: **TextEditor ScrollDispatcher Dispatcher Object**

Inherited By: **(None)**

Named Instance Variables:

active
 (From class **Dispatcher**)
modified
 (From class **TextEditor**)
newSelection
 (From class **TextEditor**)
pane
 (From class **Dispatcher**)
priorSelection
 (From class **TextEditor**)
priorText
 (From class **TextEditor**)

Class Variables:

CopyBuffer
 (From class **TextEditor**)
PageScroll
 (From class **ScrollDispatcher**)
PriorCommand
 (From class **TextEditor**)
StandardEditMenu
 (From class **TextEditor**)
WindowActivateKey
 (From class **Dispatcher**)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class **Character**) input from the keyboard or mouse.

Class Methods: **(None)**

Instance Methods:**isControlActive**

Answer true if the receiver is active.

Prompter

A Prompter is a window with one TextPane which allows an application to pose a question and solicit an answer from the user. The question is shown as the label of the window. The answer is typed in the TextPane by the user with full editing capabilities. Depending on which class method is used to invoke a Prompter, the output from the Prompter can be either the entered string from the user or an object as the result of evaluating the entered string.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:**evaluating**

Contains a Boolean which is set to true when the evaluation of the user response is requested, else it is set to false.

exitBlock

Contains a block of code with no arguments. The block, when evaluated, answers a result and exits the Prompter.

hiddenArea

Contains a Form containing a copy of the display screen image underneath the Prompter window.

reply

Contains the result of the Prompter. It is a String if the answer is not evaluated, else it is the resulting object of evaluating the String entered by the user.

replyPane

Contains the TextPane of the Prompter.

withBlank

Contains a Boolean. If true then accept the user input as is, else trim leading and trailing white space.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:**prompt: questionString default: answerString**

Open a Prompter with questionString as its question and answerString as its default answer. Answer the user response (a String) with leading and trailing spaces trimmed.

prompt: questionString **defaultExpression:** answerString

Open a Prompter with questionString as its question and answerString as its default answer. Answer the resulting object after evaluating the user response.

promptWithBlanks: questionString **default:** answerString

Open a Prompter with questionString as its question and answerString as its default answer. Answer the user response (a String) without trimming the blanks.

Instance Methods: (All private)

ReadStream

A ReadStream allows streaming over an indexed collection of objects for read access, but not write access. A stream has an internal record of its current position. It has access messages to get the object(s) at the current position and cause the position to be advanced. Messages are defined for changing the stream position, so that random access is possible.

Inherits From: Stream Object

Inherited By: (None)

Named Instance Variables:

collection

(From class Stream)

position

(From class Stream)

readLimit

(From class Stream)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

contents

Answer the collection over which the receiver is streaming.

next

Answer the next object accessible by the receiver and advance the stream position. Report an error if the receiver stream is positioned at end.

ReadWriteStream

A ReadWriteStream allows streaming over an indexed collection of objects for read and write access. A stream has an internal record of its current position. It has access messages to get and put the object(s) at the current position and cause the position to be advanced. Messages are defined for changing the stream position, so that random access is possible.

Inherits From: WriteStream Stream Object

Inherited By: FileStream TerminalStream

Named Instance Variables:

collection

(From class Stream)

position

(From class Stream)

readLimit

(From class Stream)

writeLimit

(From class WriteStream)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

contents

Answer the collection over which the receiver is streaming.

next

Answer the next object accessible by the receiver and advance the stream position.

Report an error if the receiver stream is positioned at end.

nextByte

Answer the next byte accessible by the receiver and advance the stream position.

Report an error if the stream is positioned at end.

nextPut: anObject

Write anObject to the receiver stream. Answer anObject.

nextPutAll: aCollection

Write each of the objects in aCollection to the receiver stream. Answer aCollection.

setToEnd

Set the position of the receiver stream to the end.

truncate

Set the size of the receiver stream to its current position.

Rectangle

A Rectangle represents a rectangular area (frequently used to define a subarea of a Form). A Rectangle can be described by an origin (top left corner) and a corner (bottom right corner) point, or by an origin and an extent point, where the extent describes the width and height of the rectangle.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

corner

Contains a Point describing the bottom right corner.

origin

Contains a Point describing the top left corner.

Class Variables:

(None)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:

origin: originPoint corner: cornerPoint

Answer a Rectangle with origin and corner points described by originPoint and cornerPoint.

origin: originPoint extent: extentPoint

Answer a Rectangle whose origin and extent (width and height) are described by originPoint and extentPoint.

Instance Methods:

bottom

Answer the y-coordinate of the bottom of the receiver.

center

Answer a Point, the center of the receiver.

containsPoint: aPoint

Answer true if aPoint is contained within the receiver, else answer false.

corner

Answer a Point, the bottom right corner of the receiver.

corner: aPoint

Change the receiver so that its bottom right corner is aPoint without changing its extent.

expandBy: delta

Answer a Rectangle which is the receiver expanded by delta, where delta is a Rectangle, a Point or a Number.

extent

Answer a Point representing the receiver width and height.

extent: aPoint

Change the extent of receiver to aPoint.

height

Answer a number representing the receiver height.

height: aNumber

Change the receiver height to aNumber.

insetBy: delta

Answer a Rectangle which is the receiver inset by delta, where delta is a Rectangle, a Point or a Number.

intersect: aRectangle

Answer a Rectangle representing the area in which the receiver and aRectangle overlap.

intersects: aRectangle

Answer true if the receiver and aRectangle have any area in common, else answer false.

left

Answer the x-coordinate of the origin.

merge: aRectangle

Answer the smallest Rectangle which contains the receiver and aRectangle.

moveBy: aPoint

Increment the receiver origin and corner by aPoint.

moveTo: aPoint

Move the receiver to aPoint.

nonIntersections: aRectangle

Answer an OrderedCollection of rectangles describing areas of the receiver outside aRectangle.

origin

Answer a Point, the top left corner of the receiver.

origin: originPoint **corner:** cornerPoint

Change the receiver's top left corner to originPoint and its bottom right corner to cornerPoint.

origin: originPoint **extent:** extentPoint

Change the receiver's top left corner to originPoint and its extent to extentPoint.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

right

Answer the the x-coordinate of the receiver's bottom right corner.

rounded

Answer the receiver with the coordinates of its origin and corner rounded to integers.

scaleBy: delta

Answer a Rectangle with the receiver origin and corner multiplied by delta, where delta is either a Number or a Point.

scaleTo: aRectangle

Answer a Rectangle whose size is proportional to the receiver with ratios specified by aRectangle.

top

Answer the y-coordinate of the origin of the receiver.

translateBy: delta

Answer a Rectangle which is the receiver with position incremented by delta, where delta is either a Number or a Point.

truncated

Answer the receiver with the coordinates of its origin and corner truncated to integers.

width

Answer a number representing the receiver width.

width: anInteger

Change the receiver width to anInteger.

ScreenDispatcher

A ScreenDispatcher processes the user input directed to the background (the area outside all windows). The primary user input to the screen background is to request the system menu.

Inherits From:

Dispatcher Object

Inherited By:

(None)

Named Instance Variables:

active
(From class Dispatcher)
pane
(From class Dispatcher)

Class Variables:

ScreenMenu
Contains the system menu.
WindowActivateKey
(From class Dispatcher)

Pool Dictionaries:

CharacterConstants
Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys
Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods:

systemMenu
Answer the system menu.

Instance Methods:

activateWindow
Make the screen background active. Begin the loop to process input.

execute: pathName parameters: aString
Execute a Dos program whose file name is pathName and parameter is aString.

executeCommands: aStringArray
Create the batch file to execute aStringArray as DOS commands.

executeProgram: pathName parameters: aString
Close files in use and temporarily exit to DOS. Reopen files upon return from DOS.

exit
Pop-up the exit menu.

select
Select outside all windows. Do nothing.

ScrollDispatcher

Class **ScrollDispatcher** is an abstract class which processes scrolling related inputs from either the keyboard or mouse. The scrolling commands issued from the keyboard are straightforward. The scrolling caused by the mouse right-button are of two types: a move if the cursor never goes out of the current pane while the button remains down; or a continuous scroll if the cursor goes out of the pane while the button is down. A move slides the text from the point where the mouse button is pressed to the point where the button is released. A continuous scroll moves the text by some amount continuously as long as the mouse button remains pressed down and outside the pane. In this case, the scrolling direction is determined by the cursor position relative to the active pane (e.g., text goes up when the cursor is below the active pane).

Inherits From: **Dispatcher Object**

Inherited By: **ListSelector PromptEditor TextEditor**

Named Instance Variables:

active

(From class **Dispatcher**)

pane

(From class **Dispatcher**)

Class Variables:

PageScroll

Contains true if the scrolling amount is the pane height for vertical scrolling or a half of the pane width for horizontal scrolling. Contains false if the scrolling amount is one line vertically or four characters horizontally.

WindowActivateKey

(From class **Dispatcher**)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class **Character**) input from the keyboard or mouse.

Class Methods: **(None)**

Instance Methods:

processFunctionKey: aCharacter

Process scrolling related input from keyboard or mouse.

Semaphore

A Semaphore is an object used to synchronize multiple processes.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

signalCount

Contains an integer representing the number of signal messages minus the number of wait messages sent to the semaphore during its entire lifetime.

waitingProcesses

Contains an OrderedCollection of processes that have sent the message wait to the semaphore without a corresponding signal message.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

new

Answer a new semaphore with empty waiting queue and zero signal count.

Instance Methods:

hasSignals

Answer true if there have been more signals than waits, else answer false.

signal

Increment the receiver's signal count. If there are processes waiting on the semaphore, resume the longest waiting. Upon exit, interrupts are always enabled

wait

Force the current process to be suspended until the receiver semaphore is signalled. Upon exit, interrupts are always enabled

Set

A Set represents an unordered collection of objects with no external keys. All elements of a Set are unique, i.e., duplicates are not maintained. Sets are hashed for rapid searching.

Inherits From: Collection Object

Inherited By: Dictionary IdentityDictionary MethodDictionary
SymbolSet SystemDictionary

Named Instance Variables:**contents**

Contains an Array of objects included in the Set.

elementCount

Contains the number of elements of the Set which is the number of non-nil entries in the contents array.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:**new**

Answer a new Set.

new: anInteger

Answer a new Set with an initial capacity of anInteger elements.

Instance Methods:**add: anObject**

Answer anObject. Add anObject to the receiver if the receiver does not already contain it.

at: anInteger

Access the element at index position anInteger in the receiver. This method reports an error since sets cannot be indexed.

at: anInteger put: anObject

Replace the element at index position anInteger in the receiver with anObject. This method reports an error since sets are not indexable.

do: aBlock

Answer the receiver. For each element in the receiver, evaluate aBlock with that element as the argument.

includes: anObject

Answer true if the receiver includes anObject as one of its elements, else answer false.

occurrencesOf: anObject

Answer 1 if the receiver includes anObject as one of its elements, else answer zero.

remove: anObject ifAbsent: aBlock

Answer anObject. Remove the element anObject from the receiver collection. If anObject is not an element of the receiver, aBlock is evaluated (with no arguments).

size

Answer the number of elements contained in the receiver.

SmallInteger

Class SmallInteger is used to define additional protocol for numbers in the range of -32767 to + 32767. These numbers are represented directly in their object pointer so they do not use object space.

Inherits From: Integer Number Magnitude Object

Inherited By: (None)

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

asPrinterErrorFlag

Set the receiver as the printer error flags mask.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

SortedCollection

A SortedCollection contains elements sorted according to the two argument block of code known as the sort block (sortBlock). The sortBlock, when evaluated with two elements as the arguments, will dictate which element comes first in the collection. When elements are added or removed from a sorted collection, the collection remains in sorted order.

Inherits From: OrderedCollection IndexedCollection Collection Object

Inherited By: (None)

Named Instance Variables:

contents

(From class OrderedCollection)

endPosition

(From class OrderedCollection)

sortBlock

Contains the block of code such that when it is evaluated for a pair of elements, it dictates which element comes first in the SortedCollection.

startPosition

(From class OrderedCollection)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

new: anInteger

Answer a SortedCollection capable of holding anInteger number of elements which will sort in ascending order.

sortBlock: aBlock

Answer aSortedCollection which will sort in the order defined by aBlock.

Instance Methods:

add: anObject

Answer anObject. Add anObject to the receiver in sorted position.

add: newObject after: oldObject

Add newObject after the element oldObject in the receiver. This method reports an error since the sortBlock determines element order.

add: newObject before: oldObject

Add newObject before the element oldObject in the receiver. This method reports an error since the sortBlock determines element order.

addAll: aCollection

Answer aCollection. Add all the elements in aCollection to the receiver in sorted order.

addAllFirst: aCollection

Add all the elements of aCollection to the receiver before its first element. This method reports an error since the sortBlock determines element order.

addAllLast: aCollection

Add all the elements of aCollection to the receiver after its last element. This method reports an error since the sortBlock determines element order.

addFirst: anObject

Add anObject before the first element of the receiver. This method reports an error since the sortBlock determines element order.

addLast: anObject

Add anObject after the last element of the receiver. This method reports an error since the sortBlock determines element order.

at: anInteger put: anObject

Replace the element at index position `anInteger` in the receiver collection with `anObject`. This method reports an error since the `sortBlock` determines element order.

copyFrom: beginning to: end

Answer a `SortedCollection` containing the elements of the receiver from index position `beginning` through index position `end`.

sortBlock

Answer the block that determines sort ordering for the receiver.

sortBlock: aBlock

Answer the receiver. Set the sort block for the receiver to `aBlock` and resort the receiver.

Stream

Class `Stream` and its subclasses are used for accessing files, devices and internal objects as a sequence of characters or other objects. A stream has an internal record of its current position. It has access messages to get or put the object(s) at the current position and cause the position to be advanced. Messages are defined for changing the stream position, so that random access is possible.

Inherits From: `Object`

Inherited By: `FileStream ReadStream ReadWriteStream`
`TerminalStream WriteStream`

Named Instance Variables:

collection

Contains the indexed collection being streamed over. For `FileStreams`, it contains the file page buffer string.

position

Contains an integer representing the current stream position.

readLimit

Contains an integer representing the current number of elements in the stream.

Class Variables:

(None)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., `Space` for the space character, `Lf` for the line-feed character, etc.).

Class Methods:**on: anIndexedCollection**

Answer a new instance of the receiver on anIndexedCollection.

Instance Methods:**atEnd**

Answer true if the receiver is positioned at the end (beyond the last object), else answer false.

close

Close the stream. Do nothing for non-file streams.

contents

Answer the collection over which the receiver is streaming.

copyFrom: firstIndex to: lastIndex

Answer the subcollection of the collection over which the receiver is streaming, from firstIndex to lastIndex.

countBlanks

Skip over blank and tab characters. Answer the number of character positions skipped, counting 1 for blanks and 4 for tabs.

do: aBlock

Evaluate aBlock once for each element in the receiver, from the current position to the end.

fileIn

Read and execute the Smalltalk source code chunks from the receiver. If a chunk starts with ! send it the message fileInFrom: self

isEmpty

Answer true if the receiver stream contains no elements, else answer false.

lineDelimiter

Answer the default line delimiter, line-feed.

lineDelimiter: aCharacter

Change the line delimiter character to aCharacter. Ignore for non-file streams.

next: anInteger

Answer the next anInteger number of items from the receiver, returned in a collection of the same species as the collection being streamed over.

next: anInteger put: anObject

Answer anObject. Put anObject to the receiver stream anInteger number of times.

nextChunk

Answer a String up to '!', undoubling embedded '!'s. Trailing white space is skipped. The methods in sources.sml and change.log are in chunk format.

nextChunkPut: aString

Output aString terminated with '!', doubling embedded !'s and replacing groups of leading blanks with tabs. Destination is receiver stream. The methods in sources.sml and change.log are in chunk format.

nextLine

Answer a String consisting of the characters of the receiver up to the next line delimiter.

nextMatchFor: anObject

Access the next object in the receiver. Answer true if it equals anObject, else answer false.

nextPiece

File sources.sml consists of compressed sequences of characters called pieces. Answer a String containing the next piece of text to be compressed from the receiver stream.

nextWord

Answer a String containing the next word in the receiver stream. A word starts with a letter, followed by a sequence of letters and digits.

peek

Answer the next object in the receiver stream without advancing the stream position. If the stream is positioned at the end, answer nil.

peekFor: anObject

Answer true if the next object to be accessed in the receiver stream equals anObject, else answer false. Only advance the stream position if the answer is true.

position

Answer the current receiver stream position.

position: anInteger

Set the receiver stream position to anInteger. Report an error if anInteger is outside the bounds of the receiver collection.

reset

Position the receiver stream to the beginning.

reverseContents

Answer a collection of the same species as the receiver collection, with the contents in reverse order.

setToEnd

Set the position of the receiver stream to the end.

show: aCollection

Equivalent to nextPutAll: for streams. For text editor windows, causes immediate display on screen.

size

Answer the size of (number of objects in) the receiver stream.

skip: anInteger

Increment the position of the receiver by anInteger.

skipTo: anObject

Advance the receiver position beyond the next occurrence of anObject, or if none, to the end of stream. Answer true if anObject occurred, else answer false.

upTo: anObject

Answer the collection of objects from the receiver starting with the next accessible object and up to but not including anObject. Set the position beyond anObject. If anObject is not present, answer the remaining elements of the stream.

String

A String is a fixed size indexable sequence of characters (ASCII codes from 0 to 255). This class provides the protocol to compare strings, replace characters within the string, covert characters to upper or lower case, output its instances to the printer, convert its instances to Date objects, etc.

Inherits From: FixedSizeCollection IndexedCollection Collection Object

Inherited By: Symbol

This class contains indexed byte values.

Class Variables:

(None)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods: (None)

Instance Methods:

< aString

Answer true if the receiver is before aString, else answer false. The comparison is not case sensitive.

<= aString

Answer true if the receiver is before or equal to aString, else answer false. The comparison is not case sensitive.

= aString

Answer true if the receiver is equal to aString, else answer false. The comparison is case sensitive.

> aString

Answer true if the receiver is after aString, else answer false. The comparison is not case sensitive.

>= aString

Answer true if the receiver is after or equal to aString, else answer false. The comparison is not case sensitive.

asArrayOfSubstrings

Answer an array of substrings from the receiver. The receiver is divided into substrings at the occurrences of one or more space characters.

asAsciiZ

Answer a new String containing all the characters of the receiver followed by the character of ASCII value zero.

asDate

Answer a Date representing the date described by the receiver. The receiver must contain first the day number then the month name and then the year separated by blanks.

asInteger

Answer the integer conversion of the receiver; the receiver is expected to be a sequence of digits only.

asLowerCase

Answer a String containing the receiver with alphabetic characters in lower case.

asStream

Answer a ReadWriteStream on the receiver.

asString

Answer the string representing the receiver (the receiver itself).

asSymbol

Answer a symbol whose characters are the same as the receiver string.

asUpperCase

Answer a String containing the receiver with alphabetic characters in upper case.

at: anInteger

Answer the character at position anInteger in the receiver string.

at: anInteger put: aCharacter

Answer aCharacter. At index position anInteger in the receiver put the character aCharacter.

basicAt: anInteger

Answer the character at position anInteger in the receiver string.

basicAt: anInteger put: aCharacter

Answer aCharacter. At index position anInteger in the receiver put the character aCharacter.

displayAt: aPoint

Output the receiver directly onto the display screen at aPoint.

displayAt: aPoint font: aFont

Output the receiver onto the display screen in white at aPoint with font aFont.

edit

Open a workspace window with the receiver string as the contents.

equals: aString

Answer true if the receiver is equal to the argument aString, else answer false. Note that the comparison is case sensitive.

fileExtension

Answer a three character String that follows the receiver's first period character (for DOS file names).

fileName

Answer the characters of the receiver string up to the first period character. Report an error if the resulting string is greater than eight or less than one character (for DOS file names).

hash

Answer the integer hash value for the receiver.

magnifyBy: aPoint

Answer a Form containing the receiver using system font magnified by aPoint. The coordinates of aPoint define the horizontal and vertical magnification factors respectively.

outputToPrinter

Answer the receiver. Output the receiver string to the printer. Report an error if unsuccessful.

printOn: aStream

Append the receiver as a quoted string to aStream doubling all internal single quote characters.

replaceFrom: start**to: stop with: aString startingAt: repStart**

Replace the characters of the receiver at index positions start through stop with consecutive characters of aString beginning at index position repStart. Answer the receiver.

replaceFrom: start to: stop withObject: aCharacter

Replace the characters of the receiver at index positions start through stop with aCharacter. Answer aCharacter.

size

Answer the size of the receiver string.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reinstated.

stringHash

Answer the integer hash value for the receiver.

trimBlanks

Answer a String containing the receiver string with leading and trailing blanks removed.

withCrs

Answer the receiver string where each occurrence of the character \ has been replaced with a line-feed character.

StringModel

A StringModel serves as a text holder and assists the TextEditor class by performing editing functions on the text it contains. It holds an OrderedCollection of strings (as lines in a document) and provides functions like cut, paste, and copy to modify the text. Most of these editing functions are applied to a selection of characters in the text. A selection is a Rectangle whose origin and corner represent the positions of the beginning and ending characters included in the selection.

Inherits From: Object

Inherited By: (None)

Named Instance Variables:

charScanner

Contains a CharacterScanner used to display the contents of the receiver.

extent

Contains the Point that represents the position of the last character in the text.

frame

Contains the Rectangle that limits the display area of the receiver.

lastChild

Contains the TextPane associated with the StringModel.

lines

Contains an OrderedCollection of strings which are the text data held by a StringModel. Each String is a line of the text.

topCorner

Contains a Point describing the position of the top left corner of the frame in relation to the beginning of the text.

Class Variables:

(None)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:**for: aString**

Answer a new instance of class StringModel containing an OrderedCollection of strings created from aString by separating it at the line-feed characters.

Instance Methods:**appendChar: aCharacter**

Append aCharacter to the end of the last line and inform the text pane to update.

appendText: aString

Append aString to the end of the last line and inform the text pane to update.

delete: selection

Delete the text contained in selection. Answer the Point position before the deletion.

display: aRectangle

Display the text contained in aRectangle.

displayAll

Display the text contained in the pane.

extent

Answer the last character position in the text as a Point.

fileInFrom: aStream

Replace receiver's contents with the contents of aStream.

fileOutOn: aStream

Write the receiver contents to aStream.

frame: aRectangle

Change the frame to aRectangle.

getSelectionFrom: beginIndex to: endIndex

Answer a TextSelection containing the characters from beginIndex to endIndex, treating the lines of the receiver as one string.

lineAt: anInteger

Answer the String in the line indexed by anInteger.

linesIn: aTextSelection

Answer an OrderedCollection of the lines contained in aTextSelection.

maxLineBetween: x and: y

Answer the max line length between line x and line y.

replace: aTextSelection withChar: aCharacter

Replace the text in aTextSelection with aCharacter. Answer a Point describing the position of the new character. Inform the text pane of the change.

replace: aTextSelection withText: aString

Replace the text in aTextSelection with aString. Answer a Point describing the position of the last replacement character. Inform the text pane of the change.

replaceAtColumns: aPoint **by:** aString **startAt:** aTextSelection

Replace the line contents between the coordinates of aPoint with aString in aTextSelection. Answer a TextSelection of the new string.

scanForWordAt: aPoint

Find the word which surrounds the point.

scanner: aCharacterScanner

Set scanner to aCharacterScanner.

searchBack: aTextSelection **for:** aPattern

Search backward for aPattern starting from the end of aTextSelection. Answer the matched selection if found, else answer nil.

searchFrom: aTextSelection **for:** aPattern

Search for aPattern starting from the end of aTextSelection. Answer the matched selection if found, else answer nil.

string

Answer a String containing the receiver contents.

string: aString

Change the receiver contents to aString (lines are separated by line-feeds).

stringIn: aTextSelection

Answer a String which concatenates all the lines contained in the aTextSelection.

textPane: aTextPane

Associates aTextPane to the receiver by setting the lastChild to it.

topCorner: aPoint

Change topCorner to aPoint.

totalLength

Answer the number of lines held by the receiver.

SubPane

Class SubPane is an abstract class which provides the functions that are common to the ListPane and TextPane classes.

Inherits From: Pane Object

Inherited By: GraphPane ListPane TextPane

Named Instance Variables:

changeSelector

Contains a message selector which is used when a change in the pane has global effects (affects model or other panes).

curFont

(From class Pane)

dispatcher

(From class Pane)

frame

(From class Pane)

framingBlock

(From class Pane)

margin

Currently not used.

model

(From class Pane)

name

Contains a Symbol which is used as both the name of the pane and a message selector to be sent when an update of the pane is needed.

paneMenuSelector

(From class Pane)

paneScanner

(From class Pane)

scrollBar

Contains a BitBlt used in drawing the scroll bar.

subpanes

(From class Pane)

superpane

(From class Pane)

topCorner

Contains a Point describing the position of the top left corner of the frame in relation to the beginning of the text.

Class Variables:

WindowClip

(From class Pane)

ZoomedPane

(From class Pane)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods: (None)

Instance Methods:

activateWindow

Perform the window activation function for the receiver pane. Default is do nothing.

change: aSymbol

Set the changeSelector to aSymbol.

charsInColumn

Answer the receiver frame height in characters.

charsInRow

Answer the receiver frame width in characters.

displayWindow

Display the portion of the receiver pane that intersects with WindowClip.

framingBlock: aBlock

Initialize the framingBlock to the one argument block aBlock which, when executed, yields the pane frame rectangle.

framingRatio: aRectangle

Initialize the framingBlock to a one argument block which, when executed, yields the pane frame rectangle proportional with the ratios specified by aRectangle.

name: aSymbol

Set the pane name to aSymbol.

reframe: aRectangle

Change the frame of the receiver pane to aRectangle. Also initialize the scroll bar and the characterScanner.

scrollBarUpdate

Update the mark in the scroll bar.

showGap

Show the gap selection if the pane has one.

topPane

Answer the top pane of the receiver pane.

update

Update the contents of the receiver pane. Default is do nothing.

update: aSymbol

If aSymbol is equal to the name of the receiver pane then update the contents of the pane.

update: nameSymbol with: aSymbol

If nameSymbol is equal to the name of the receiver pane then perform the selector aSymbol.

update: nameSymbol with: aSymbol with: anObject

If nameSymbol is equal to the name of the receiver pane then perform aSymbol with anObject as its argument.

Symbol

A Symbol is a fixed size sequence of characters guaranteed to be unique throughout the system. Hence, no copies can be made of the instances of this class and the existing instances cannot be modified. Symbols are removed from the system through the cloning process.

Inherits From: String FixedSizeCollection IndexedCollection Collection Object

Inherited By: (None)

This class contains indexed byte values.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

mustBeSymbol: aSymbol

Report an error if aSymbol is not a Symbol.

new: ignoreArgument

Answer an instance of the receiver. This method reports an error.

purgeUnusedSymbols

Purge unused symbols from symbol table.

Instance Methods:

= aSymbol

Answer true if the receiver object is the the argument aSymbol, else answer false.

asString

Answer a String of the characters contained by the receiver.

asSymbol

Answer a Symbol for the receiver. The receiver itself is answered since it is a Symbol.

at: anInteger put: aCharacter

Replace the character in the receiver indexed by anInteger with the argument aCharacter. This message is not valid for symbols, since they are not allowed to change.

deepCopy

Answer a copy of the receiver with shallow copies of each instance variable. Because symbols are unique (cannot be copied), answer the receiver.

hash

Answer the integer hash of the receiver.

printOn: aStream

Append the ASCII representation of the receiver to aStream.

shallowCopy

Answer a shallow copy of the receiver. Because symbols are unique (cannot be copied), answer the receiver.

species

Answer class String as the species of symbols.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reconstructed.

SymbolSet

A SymbolSet is a set used to record all the Symbol instances. There is only one instance of SymbolSet which is answered by the message Symbol symbolTable. The symbol set is special in that entries are hashed and compared as strings rather than as symbols, guaranteeing that all symbols are unique.

Inherits From: Set Collection Object

Inherited By: (None)

Named Instance Variables:

contents

(From class Set)

elementCount

(From class Set)

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods: (None)

Instance Methods:

species

Answer class Set as the species of SymbolSet.

SystemDictionary

A SystemDictionary contains all the global variables. There is only one instance of class SystemDictionary which may be referred to by the name Smalltalk. Each global variable is represented by an Association. The key is a Symbol containing the global variable name (beginning with an upper-case character). The associated value contains the value of the global variable. Class SystemDictionary also defines the protocol for system utility functions.

Inherits From: Dictionary Set Collection Object

Inherited By: (None)

Named Instance Variables:

contents
 (From class Set)
elementCount
 (From class Set)

Class Variables:

(None)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods: (None)

Instance Methods:

add: anAssociation

Answer anAssociation. Add anAssociation to receiver. Ensure that the key is a symbol.

at: aSymbol put: anObject

Answer anObject. Enter anObject at key aSymbol in the receiver. Ensure that aSymbol is a symbol.

compressChanges

Build a new change log file retaining only the latest version of changed methods in the current change log. Save the image to the image file.

compressSources

Build a new source file which contains the latest version of all methods. Build a zero length change log file. Save the image to the image file.

exit: aBoolean

Temporarily exit to DOS. If aBoolean is true, clear the screen upon exit, else leave screen as is.

getSourceClasses

Answer an OrderedCollection of all classes in hierarchical order.

implementorsOf: aSymbol

Pop-up an implementors window for selector aSymbol.

loadPrimitivesFrom: aFilePathName

Load a user primitive module from file aFilePathName.

sendersOf: aSymbol

Pop-up an senders window for selector aSymbol.

startUp

Initiate a Smalltalk/V session by filing in the 'go' file.

unusedMemory

Answer an integer which is the number of bytes of unused memory available for object storage.

TerminalStream

Class TerminalStream defines the streaming protocol to and from the terminal. There is a global variable Terminal which is the instance of TerminalStream used throughout Smalltalk/V. Output to a TerminalStream writes characters on the display screen, input from TerminalStream returns user input from the keyboard and mouse. TerminalStream uses a finite state machine to decode the user input from InputEvent.

Inherits From: **ReadWriteStream WriteStream Stream Object**

Inherited By: **(None)**

Named Instance Variables:

collection

(From class Stream)

mouseOffset

Contains the cursor location each time the right mouse button is pressed. When scrolling is detected, this position is used as the scrolling start position.

mouseTime

Contains the time in milliseconds each time the left or right mouse button is pressed. This variable is used to compute the delay between the press and the release of the mouse button.

position

(From class Stream)

readLimit

(From class Stream)

state

Contains the current state which is the method to be performed next (when the method read is invoked).

writeLimit

(From class WriteStream)

Class Variables:

(None)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods: (None)

Instance Methods:

bell

Output the Bell character to the terminal.

initialize

Initialize the global variables FunctionKey and MouseEvent to false and the state of the input mechanism to initialState.

mouseOffset

Answer the cursor position where the mouse button was pressed.

mouseSelectOn

Answer true if left button is down.

next

Answer the next character from the terminal (keyboard or mouse).

nextPut: aCharacter

Answer the argument aCharacter. Output aCharacter to the display screen.

nextPutAll: aString

Answer the argument aString. Output aString to the display screen.

read

Answer the next keyboard or mouse event.

write: aCharacter

Output aCharacter to the terminal.

TextEditor

A TextEditor processes input for its associated TextPane. Its input can be a cursor movement, scrolling command, menu request, text selection, or editing command.

Inherits From: ScrollDispatcher Dispatcher Object

Inherited By: PromptEditor

Named Instance Variables:

active

(From class Dispatcher)

modified

Contains true if the text has been modified since last save, else it contains false.

newSelection

Contains a Rectangle describing the new selection.

pane

(From class Dispatcher)

priorSelection

Contains the selection Rectangle prior to a cut, copy, or paste operation.

priorText

Contains a String which is the selected text prior to a cut, copy, or paste operation.

Class Variables:

CopyBuffer

Contains a String which is the selected text of the last cut or copy operation.

PageScroll

(From class ScrollDispatcher)

PriorCommand

Contains the prior command for the again menu selection (search or replace).

StandardEditMenu

Contains the standard editing menu normally used by a TextPane.

WindowActivateKey

(From class Dispatcher)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods:

menu

Answer the standard edit menu.

windowLabeled: aString frame: aRectangle

Create a new window with label aString, frame aRectangle and answer its dispatcher.

Instance Methods:

compilerError: aString at: anInteger in: codeString for: aClass

Display the error message aString in reversed form at the indicated position anInteger in the source codeString.

contents

Answer the contents of the text pane as a String.

cr

Append a line-feed to the end of the text in the pane.

isControlActive

Answer true if the receiver is active.

modified

Answer true if the text has been modified since the last save, else answer false.

modified: aBoolean

Change modified to aBoolean.

next: anInteger **put:** aCharacter

Put aCharacter to the receiver TextEditor anInteger number of times.

nextPut: aCharacter

Add aCharacter at the end of the text in the pane.

nextPutAll: aString

Add aString at the end of the text in the pane.

show: aString

Add aString at the end of the text in the pane and force it to be shown.

space

Append a space to the end of the text in the pane.

tab

Append a tab to the end of the text in the pane.

zoom

Zoom the pane

TextPane

Class TextPane provides functions to display and scroll a portion of the text held by the pane. In addition, it allows the user to edit the text. The text is usually represented as an instance of StringModel. When the user saves the edited text, the application model is informed to accomplish the saving.

Inherits From: SubPane Pane Object

Inherited By: (None)

Named Instance Variables:

changedArea

Contains a rectangle whose origin and corner are the begining and ending positions of the changed area of the text in the pane.

changeSelector

(From class SubPane)

curFont

(From class Pane)

dispatcher

(From class Pane)

frame

(From class Pane)

framingBlock

(From class Pane)

margin
 (From class SubPane)

model
 (From class Pane)

name
 (From class SubPane)

paneMenuSelector
 (From class Pane)

paneScanner
 (From class Pane)

reserved
 Reserved for future use.

scrollBar
 (From class SubPane)

selection
 Contains a rectangle whose origin and corner represent the beginning and ending points of a selection. Note that only the first and last lines within a selection may be partial lines and all the lines in between are entirely selected.

subpanes
 (From class Pane)

superpane
 (From class Pane)

textHolder
 Contains the text of the pane which is normally a StringModel.

topCorner
 (From class SubPane)

Class Variables:

NewString
 Contains a String to replace occurrences of OldString during a replace operation.

OldFrame
 Contains the original frame Rectangle of the text pane being zoomed.

OldString
 Contains a String whose occurrences will be replaced by NewString during a replace operation.

SearchString
 Contains the String to be searched for during a search operation.

WindowClip
 (From class Pane)

ZoomedPane
 (From class Pane)

Pool Dictionaries:

CharacterConstants
 Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:**unzoom**

Unzoom the zoomed pane, if there is one.

Instance Methods:**appendChar: aCharacter**

Append aCharacter to the end of the text.

appendText: aString

Append aString to the end of the text.

close

Close the pane.

compilerError: aString at: anInteger in: codeString for: aClass

Display the error message aString as the selected text at the indicated position
anInteger in the source codeString.

contents

Answer a String, the contents of the pane.

deactivatePane

Deactivate the receiver pane.

defaultDispatcherClass

Answer the default dispatcher.

display: aString reverseFrom: startInteger to: endInteger

Display aString and select characters between startInteger and endInteger.

displayChanges

Update the screen with all pending changes.

fileInFrom: aFileStream

Refresh the pane data with the current contents of aFileStream.

fileOutOn: aFileStream

Write the pane data out on aFileStream.

forceEndOntoDisplay

Force the end of the text to appear on the display screen.

forceSelectionOntoDisplay

Force the origin of the selection to appear on the display screen.

formCoordinates: aPoint

Convert string coordinates to form coordinates.

frame

Answer the receiver frame.

initialize

Initialize the receiver.

reframe: aRectangle

Change the frame of the receiver pane to aRectangle.

selectAfter: aPoint

Place the selection after aPoint.

selectAll

Select the entire text of the pane.

selectAtEnd

Place the gap selection at the end of the text.

selectBefore: aPoint

Place the gap selection before aPoint.

selectedString

Answer a String containing the text currently selected.

selectFrom: startPoint to: endPoint

Set the selection to the rectangle described by the origin startPoint and the corner endPoint.

selection

Answer a TextSelection describing the current selection.

selectTo: aPoint

Extend the selection to aPoint either before or after the original one.

showSelection

Make the selection visible.

showWindow

Redraw the contents of the receiver pane.

update

Refresh the pane area on the display screen through the model.

update: anObject

The model has changed. If anObject is a TextSelection, display it, else pass it up to superclass.

TextSelection

In a TextPane, the text is represented as an OrderedCollection of Strings. It can be looked upon as a two dimensional array. The position of each character can be identified by a Point whose x coordinate (column) is the index within the String and y coordinate (row) is the index in the OrderedCollection. When a selection is made in the TextPane, the selection is represented internally as two points: the positions of the first and last characters included in the selection. In the case of a gap selection, the column of the second point will be one less than the first while their rows are the same. Besides remembering these two points, a TextSelection also understands all the messages for manipulating the selection.

Inherits From:

Object

Inherited By: (None)

Named Instance Variables:

begin

Contains a Point representing the position of the first character in the selection.
end

Contains a Point representing the position of the last character in the selection.

extendOrigin

Contains a Point indication the starting position when a selection is being extended.

pane

Contains the Pane that this selection belongs to.

selectFlag

Contains a Boolean which is true when the selection is being shown; false when it is not shown.

Class Variables:

(None)

Pool Dictionaries: (None)

Class Methods:

new

Answer a new TextSelection.

origin: beginPoint corner: endPoint

Answer an instance of the receiver whose origin is beginPoint and corner is endPoint.

Instance Methods:

display

Display the gap selector or the selection.

gray

Color the non-gap selection gray.

hide

Hide the gap selector or the selection.

intersect: aTextSelection

Answer a Rectangle, the intersection of the receiver and aTextSelection.

isGap

Answer true if the selection is a gap.

merge: aTextSelection

Answer a TextSelection, the receiver merged with aTextSelection.

origin: beginPoint corner: endPoint

Change the origin and corner of the receiver to beginPoint and endPoint respectively.

selectAfter: aPoint

Place the selection after aPoint.

selectBefore: aPoint

Place the selection before aPoint.

selectTo: aPoint

Extend the selection to aPoint either before or after the original one.

Time

Class Time is used to represent a particular time of day to the nearest second. It defines the protocol for comparing, computing, and creating times.

Inherits From: **Magnitude Object**

Inherited By: **(None)**

Named Instance Variables:

seconds

Contains the number of seconds that have elapsed since midnight.

Class Variables:

TimeTickOn

Contains a Boolean indicating whether or not the clock ticks are to be monitored.

ValueArray

Contains a 4 element Array. The read current time primitive sets this variable to the current time whenever the primitive is invoked. This variable is filled as follows:
(hours minutes secons milliseconds).

Pool Dictionaries: **(None)**

Class Methods:

clockTickPeriod: anInteger

Enable the clock interrupt. Timer interrupts will occur every (55 * anInteger) milliseconds.

clockTicksOff

Turn off clock interrupts.

dateAndTimeNow

Answer an Array of two elements containing the current date and the current time.

fromSeconds: anInteger

Answer a Time which represents anInteger number of seconds from midnight.

millisecondClockValue

Answer the number of milliseconds from midnight of the current day to the current time.

millisecondsToRun: aBlock

Answer the number of milliseconds it takes to evaluate aBlock.

now

Answer a Time representing the current time in seconds.

totalSeconds

Answer the number of seconds from midnight of the current day to the current time.

Instance Methods:**< aTime**

Answer true if the receiver is less than aTime, else answer false.

<= aTime

Answer true if the receiver is less than or equal to aTime, else answer false.

= aTime

Answer true if the receiver is equal to aTime, else answer false.

> aTime

Answer true if the receiver is greater than aTime, else answer false.

>= aTime

Answer true if the receiver is greater than or equal to aTime, else answer false.

addTime: timeAmount

Answer a Time which is timeAmount seconds past the receiver time.

asSeconds

Answer an Integer representing the number of seconds of the receiver time.

hash

Answer the integer hash value for the receiver.

hours

Answer an Integer representing the number of hours of the receiver time.

minutes

Answer an Integer representing the number of minutes past the hour in the receiver time.

printOn: aStream

Append the ASCII representation of the receiver to aStream in the form: hh:mm:ss.

seconds

Answer an Integer representing the number of seconds past the minute in the receiver time.

seconds: anInteger

Answer the receiver. Set the number of seconds in the receiver to anInteger.

subtractTime: timeAmount

Answer the time that is timeAmount seconds before the receiver time.

TopDispatcher

A TopDispatcher processes input for its associated TopPane. Its input can be a cursor movement or menu request. It also provides functions for changing the visual cues of the window and answering some default window menus.

Inherits From: Dispatcher Object

Inherited By: (None)

Named Instance Variables:

active
(From class Dispatcher)
pane
(From class Dispatcher)

Class Variables:

TopPaneMenu

Contains the standard window menu.

TranscriptMenu

Contains the menu for the system transcript window.

WindowActivateKey

(From class Dispatcher)

WorkSpaceMenu

Contains the menu for the work space window.

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class Character) input from the keyboard or mouse.

Class Methods:

initialize

Set up the standard window menu.

menu

Answer the standard window menu.

Instance Methods:

highlightLabel

Inform the top pane to highlight the window label signaling the active window.

homeCursor

Move the cursor to home position of the first subpane.

isControlActive

Answer true if the receiver is the topDispatcher and its window has the cursor and the cursor is not in any subpanes, else answer false.

label

Prompt the user for a new label of the window and answer the label.

TopPane

A TopPane is responsible for all the operations related to its entire window (as opposed to operations related to the panes). For some operations (e.g., display window), it also invokes subpanes in sequence to complete the work.

Inherits From: Pane Object

Inherited By: (None)

Named Instance Variables:

backColor

Contains a mask form describing the background color of the window.

borders

Contains a Form with an image of the window. All of the window updates and visual cues are first output to this form which is then copied to the display screen.

collapsed

Contains a Boolean indicating whether or not the window is collapsed.

curFont

(From class Pane)

dispatcher

(From class Pane)

foreColor

Contains a mask form describing the text color of the window.

frame

(From class Pane)

framingBlock

(From class Pane)

iconArray

Contains an Array of two arrays of icons on the left and right of the window label.

label

Contains a Form whose content is an image of the window label.

minimumSize

Contains a Point describing the minimum width and height of the window.

model

(From class Pane)

paneMenuSelector

(From class Pane)

paneScanner

(From class Pane)

previousFrame

Contains a Rectangle of the collapsed window if the window is uncollapsed, else the uncollapsed window.

subpanes

(From class Pane)

superpane

(From class Pane)

Class Variables:

LabelIcons

Contains a Dictionary of dictionaries of label icons. The keys of the top dictionary are fonts. The keys of the lower dictionary are symbols of the icon names.

WindowClip

(From class Pane)

ZoomedPane

(From class Pane)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods:

(All private)

Instance Methods:

activateWindow

Activate the top dispatcher, display the label, and invoke window activation methods of all subpanes.

addSubpane: aPane

Add subpane aPane to the receiver.

backColor

Answer the background color of the window.

backColor: aColor

Set the window background color to aColor.

backupWindow

If a backup window is requested, save the window image on backup form.

close

Close the receiver and all subpanes.

deactivatePane

Window has been deactivated. Do nothing for a TopPane.

defaultDispatcherClass

Answer the default dispatcher.

displayWindow

Display the label and the contents of the subpanes excluding the portion outside of WindowClip.

foreColor: aColor

Set the window foreground color to aColor.

frame

Answer the window frame.

highlightLabel

Display the label string in reversed color.

label

Answer the label of the window.

label: aString

Change the window label to aString.

leftIcons: anArray

Request anArray of icons to be shown on the left side of the label.

minimumSize: aPoint

Change the minimum size of the window to aPoint.

reframe: aRectangle

Reframe the receiver window according to aRectangle.

rightIcons: anArray

Request anArray of icons to be shown on the right side of the label.

topPane

Answer the top pane of the window which is self.

update: aSymbol

If aSymbol equals #label then update the window label, else do nothing.

zapBackup

Purge the backup form for the speed mode.

True

Class True has a single instance, true, representing logical truth. This class defines the protocol for logical operations on true.

Inherits From: Boolean Object

Inherited By: (None)

Named Instance Variables: (None)

Class Variables:

(None)

Pool Dictionaries: (None)**Class Methods:** (None)**Instance Methods:****& aBoolean**

Answer true if both the receiver and aBoolean are true, else answer false.

and: aBlock

If the receiver is true, answer the result of evaluating aBlock (with no arguments), else answer false.

eqv: aBoolean

Answer true if the receiver is equivalent to aBoolean, else answer false.

hash

Answer the hash of true.

ifFalse: aBlock

If the receiver is false, answer the result of evaluating aBlock (with no arguments), else answer nil.

ifFalse: falseBlock ifTrue: trueBlock

If the receiver is true, answer the result of evaluating trueBlock, else answer the result of evaluating falseBlock. Both blocks are evaluated with no arguments.

ifTrue: aBlock

If the receiver is true, answer the result of evaluating aBlock (with no arguments), else answer nil.

ifTrue: trueBlock ifFalse: falseBlock

If the receiver is true, answer the result of evaluating trueBlock, else answer the result of evaluating falseBlock. Both block are evaluated with no arguments.

not

Answer true if the receiver is false, else answer false.

or: aBlock

If the receiver is false, answer the result of evaluating aBlock, else answer true.

xor: aBoolean

Answer true if the receiver is not equivalent to aBoolean, else answer false.

| aBoolean

Answer true if either the receiver or aBoolean are true, else answer false.

UndefinedObject

Class **UndefinedObject** has a single instance, **nil**, used to identify undefined values. The instance variables of any object are initialized to **nil** upon creation. This guarantees that every variable has a value which is an instance of some class.

Inherits From: **Object**

Inherited By: **(None)**

Named Instance Variables: **(None)**

Class Variables:

(None)

Pool Dictionaries:

CharacterConstants

Defines variables for some of the most frequently used characters (e.g., **Space** for the space character, **Lf** for the line-feed character, etc.).

FunctionKeys

Defines variables for the function key codes (of class **Character**) input from the keyboard or mouse.

Class Methods:

new

Create a new instance of the receiver. Disallowed for this class because there is only a single instance, **nil**.

new: anInteger

Create a new instance of the receiver. Disallowed for this class because there is only a single instance, **nil**.

Instance Methods:

deepCopy

Answer a copy of the receiver with shallow copies of each instance variable. Because there is only one **nil**, answer the receiver.

isNil

Answer true because the receiver is **nil**.

notNil

Answer false because the receiver is **nil**.

printOn: aStream

Append the ASCII representation of the receiver to **aStream**.

shallowCopy

Answer a copy of the receiver which shares the receiver instance variables. Because there is only one **nil**, answer the receiver.

storeOn: aStream

Append the ASCII representation of the receiver to aStream from which the receiver can be reconstructed.

subclass: classSymbol

instanceVariableNames: instanceVariables

classVariableNames: classVariables **poolDictionaries:** poolDictNames

Create or modify the class named by classSymbol to be a subclass of receiver with the specified instance variables, class variables, and pools. Used only to define class Object, because its superclass is nil.

WildPattern

An instance of WildPattern contains a finite state pattern for efficient matching which includes at least one wild card character. The wild card character will match zero or more elements in the matching collection until the rest of the pattern is matched or the end of the matching collection is reached.

Inherits From: Pattern Object

Inherited By: (None)

Named Instance Variables:

fail

(From class Pattern)

first

(From class Pattern)

input

(From class Pattern)

matchBlock

(From class Pattern)

patternCollection

Contains an OrderedCollection of Patterns which are the subpatterns of the original pattern separated at each wild card character.

state

(From class Pattern)

Class Variables:

WildcardChar

(From class Pattern)

Pool Dictionaries: (None)

Class Methods:

new: aCollection

Answer a new WildPattern with aCollection as the pattern to match.

Instance Methods:**match: anObject**

Compare anObject against the pattern. If this object completes the matching of the pattern, evaluate the match block.

match: aCollection index: anInteger

Answer a Point representing the start and stop of the subcollection within aCollection that matches the receiver starting at index position anInteger. Answer nil if no match.

reset

Reset the receiver to start matching at the beginning of the pattern.

WriteStream

A WriteStream allows streaming over an indexed collection of objects for write access, but not read access. A stream has an internal record of its current position. It has access messages to put the object(s) at the current position and cause the position to be advanced. Messages are defined for changing the stream position, so that random access is possible.

Inherits From: Stream Object

Inherited By: FileStream ReadWriteStream TerminalStream

Named Instance Variables:**collection**

(From class Stream)

position

(From class Stream)

readLimit

(From class Stream)

writeLimit

Contains the integer position of the highest position written in the collection being streamed over.

Class Variables:

(None)

Pool Dictionaries:**CharacterConstants**

Defines variables for some of the most frequently used characters (e.g., Space for the space character, Lf for the line-feed character, etc.).

Class Methods: (None)

Instance Methods:**contents**

Answer a collection representing the contents of the stream.

cr

Write the line terminating character (line-feed) to the receiver stream.

nextBytePut: aByte

Write the character whose ASCII value is aByte to the receiver stream. Answer aByte.

nextFourBytesPut: anInteger

Write anInteger as the next four bytes of the receiver stream.

nextPut: anObject

Write anObject to the receiver stream. Answer anObject.

nextPutAll: aCollection

Write each of the objects in aCollection to the receiver stream. Answer aCollection.

nextTwoBytesPut: anInteger

Write anInteger as the next two bytes of the receiver stream.

position: anInteger

Set the receiver stream position to anInteger. Report an error if anInteger is outside the bounds of the receiver collection.

setToEnd

Set the position of the receiver stream to the end.

space

Write a space character to the receiver stream.

tab

Write a tab character to the receiver stream.

Appendices

Appendix 1: SMALLTALK SYNTAX SUMMARY

How Syntax is Specified

The formal syntax specification is presented using the Extended Backus Naur Formalism (EBNF) used in *Programming in Modula-2* by Niklaus Wirth, Springer-Verlag, 1982. EBNF is used here in order to precisely and concisely specify the syntax.

What follows is a specification of EBNF syntax in EBNF. The syntax rules are:

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

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

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

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

Examples of meals defined by these rules are the following:

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

Smalltalk Syntax

The following is an EBNF syntax specification for Smalltalk and a cross-reference index to the syntax. Each line in the syntax specification begins with a number which is used to identify the line in the index. The index shows where each rule name is defined (line number preceded by minus sign) and used.

```

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

```

```

37 symbolConstant = "#" symbol.
38 symbol = unarySelector | binarySelector | keyword {keyword}.
39 identifier = letter {letter | digit}.
40 character = selectorCharacter | letter | digit |
41   "[" | "]" | "{" | "}" | "(" | ")" | "^" | ";" | "$" |
42   "#" | ":".
43 selectorCharacter = "," | "+" | "/" | "\\" | "*" | "~" |
44   "<" | ">" | "=" | "@" | "%" | "|" | "&" | "?" | "!".
45 letter = capitalLetter |
46   "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
47   "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
48   "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z".
49 capitalLetter =
50   "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
51   "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
52   "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z".
53 digits = digit [digit].
54 digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
55   "8" | "9".
56 bigDigits = bigDigit {bigDigit}.
57 bigDigit = digit | capitalLetter.
58 comment = ' ' {character | ' ' } ' '.
59 variableName = identifier.

```

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

keywordMessage	14	19	-22					
letter	39	39	40	-45				
literal	10	-28						
messageExpression	9	-11						
messagePattern	1	-3						
method	-1							
number	5	28	31	-33				
primary	9	-10	15	16	18	21	22	23
primitiveNumber	1	-5						
selectorCharacter	26	26	40	-43				
string	28	31	-35					
symbol	31	37	-38					
symbolConstant	29	-37						
temporaries	1	-6						
unaryExpression	11	-15	16	21				
unaryMessage	13	15	15	-20				
unarySelector	3	20	-27	38				
variableName	3	4	4	6	8	10	24	-59

Appendix 2: PRIMITIVE METHODS

How Primitive Methods Work

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

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

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

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

The assembly language part of a primitive is executed first. It concludes by either succeeding (returning an object that is the method result) or failing. If the assembly language part fails, the Smalltalk part is executed to return the method result. This shared responsibility works efficiently because the assembly language code handles the most common but simple cases. Since Smalltalk is much easier to write than assembly language, the Smalltalk code handles the infrequent but complex cases.

Primitive Number Assignments

The table below lists the primitive methods used by `Smalltalk/V 286`. For each primitive, the primitive number, the method selectors, and the classes in which it is used are presented.

<u>Primitive</u>	<u>Used in method:</u>	<u>Implemented in class:</u>
17	save	SystemDictionary
18	clockTickPrimitive:	Time class
19	clockOffPrimitive	Time class
20	rem:	Integer
21	+	Integer

<u>Primitive</u>	<u>Used in method:</u>	<u>Implemented in class:</u>
22	-	
23	<	
24	>	
25	<=	
26	>=	
27	=	
28	~=	
29	*	
30	cos	
31	\\"\\	
32	//	
33	quo:	
34	bitAnd:	
35	bitOr:	
36	bitXor:	
37	bitShift:	
38	allInstancesPrim	
40	fromInteger:	
41	+	
42	-	
43	<	
44	sqrt	
45	exp	
46	ln	
47	=	
48	sin	
49	*	
50	/	
51	truncated	
52	fraction	
53	exponent	
54	timesTwoPower:	
55	equals:	
	=	
56	<=	
57	primitiveLoadModule	
58	write:	
59	copyChars	
60	basicAt:	
	at:	

<u>Primitive</u>	<u>Used in method:</u>	<u>Implemented in class:</u>
61	basicAt:put: at:put:	Object Object
62	size basicSize	String Object
	size	Object
63	at:	String
	basicAt:	String
64	at:put: basicAt:put:	String String
65	next next	ReadStream ReadWriteStream
	next	FileStream
66	primitiveNextPut: nextPut:	FileStream WriteStream
67	atEnd	Stream
68	findFirst:attribute:buffer:	Directory
69	allReferencesPrim	Object
70	new basicNew	Behavior Behavior
71	new: basicNew:	Behavior Behavior
72	become:	Object
73	instVarAt:	Object
74	instVarAt:put:	Object
75	hash: asOop basicHash	Character Object Object Object
76	hash	Object
77	dosPrimitive:registers:value:	DOS
78	drawRectangle	Pen
79	value	Context
80	value: value:value:	Context Context
81	upTo:	Stream
82	tan	Float
84	perform:withArguments:	Object
85	arcTan	Float
86	status	Float class
87	findNext:	Directory

<u>Primitive</u>	<u>Used in method:</u>	<u>Implemented in class:</u>
88	spreadFrom:to:width startAt:mask:	Form
89	fillAtX:andY:	Pen
98	configureAs:	DisplayScreen class
99	ellipsePrim:aspectX:Y:	Pen
100	drawLoopX:Y:	BitBlt
101	mousePrimWith:with:with:with:	CursorManager
102	primitiveGetEvent	InputEvent
104	swapInAndRestore:	Process
105	replaceFrom:to:with:startingAt:	String
106	replaceFrom:to:withObject:	String
107	hash	String
108	copyStack	Process class
109	dropSenderChain	Process class
110	= = = ==	Character Symbol Object Object Object
111	class	Object
112	unusedMemoryPrim	SystemDictionary
113	flushFromCache:	MethodDictionary
114	interrupt:	Process class
115	copyBits	BitBlt
116	signal	Semaphore
117	wait	Semaphore
118	enableInterrups:	Process class
119	currentDateInto:	Date class
120	currentTimeInto:	Time class
123	primitiveEndByte	FileHandle
124	openIn:	FileHandle
125	readInto:atPage:	FileHandle
126	writeFrom:toPage:for:	FileHandle
127	primitiveClose	FileHandle

User-Defined Primitive Methods

User-defined primitives are implemented as protected mode subroutines. These subroutines are collected into primitive modules that are loaded by Smalltalk/V 286 after the image is loaded. The primitive modules are memory image files (.BIN format) and contain a table of the names and entry points of the primitives (subroutines) included.

Object Pointers

Smalltalk/V 286 uses 32-bit memory addresses as object pointers. Pointers with a segment value of 6 are *positive SmallIntegers*. Pointers with a segment value of 116H are *negative SmallIntegers*. The offset of a SmallInteger is a 15 bit magnitude (always positive). This gives a range of -32767 to +32767 for small integers. All other pointers are the direct memory address of the object.

Smalltalk/V 286 uses a generation scavenging garbage collector. When an object pointer is stored into an object, data structures internal to the memory manager may need to be updated if cross generational references have been created or modified. An assembly language macro is provided for this purpose.

Accessing Objects within Primitives

Certain object pointers may need to be referenced by a user primitive. These objects are at fixed addresses. The file fixdptrs.usr contains assembly language definitions for these fixed pointers.

All objects have a twelve byte header. This is followed by the instance variables. Instance variables of an object are all of the same size. Objects that contain object pointers have 32 bit instance variables; all others have eight bit instance variables. The order of instance variables in an object is the order in which the variables are defined by the class of the object (don't forget about inherited instance variables). Assembly language definitions for the structure of the object header and some common objects is in file object.usr.

Often you need to check the class of an object to see if it is appropriate for the primitive. You can use the ClassPtrHash field in the object header for this purpose. The file object.usr also contains assembly language definitions of the fixed class hashes in the environment.

Smalltalk methods cannot corrupt object memory because an object is not allowed to access outside itself. Primitive methods can access all of object memory, and therefore have the opportunity to corrupt object memory. The implementor of a primitive has the responsibility to guarantee that only the instance variables of the receiver object are changed. If you have been using (debugging) a primitive that may have stored erroneously in memory, discard the image.

Loading Primitives

A module containing user primitives is loaded into Smalltalk/V 286 by sending the message **loadPrimitivesFrom:** to an instance of **SystemDictionary**. The argument is the file path name where the file of primitives is located. For example:

```
Smalltalk loadPrimitivesFrom: 'example.bin'
```

If you have several files containing primitives, then evaluate the above expression for each file. These expressions are usually included in the **go** file to make the loading of primitives automatic.

Reserving Space for Primitives

Primitives are loaded into memory in the address space of DOS (below 640K). Unless you tell it otherwise, Smalltalk/V 286 will allocate all of memory to itself, therefore you must reserve memory for your primitives. You specify the amount of space to reserve for primitives and DOS as the argument to the **/d:** parameter on the command line that invokes Smalltalk/V 286. For example, the following command invokes Smalltalk/V and reserves 200K for DOS and primitives to use:

```
v /d:200
```

You must specify enough memory to load all of your primitives. If you are also using the DOS shell feature you need to add that to your reservation request size. Notice that the argument is a decimal number and specifies the amount of memory to reserve in multiples of 1024 bytes. Appendix 3, Configuring Smalltalk/V 286, explains all of the command line options in detail.

Macros

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

enterPrimitive

This must be the first instruction in a user primitive. It saves the old stack frame address and sets up **BP** to point to the arguments on the stack:

[BP+6] is the DWORD containing the receiver object pointer.

[BP+10] is the DWORD containing the first argument object pointer.

[BP+14] is the DWORD containing the second argument object pointer.

[BP+18] is the DWORD containing the third argument object pointer.

etc...

exitWithSuccess

This macro is used to exit the primitive when it has successfully computed an object pointer as the method result. The 32 bit method result is placed in **DX,AX** with **DX** containing the segment part and **AX** containing the offset part.

exitWithFailure

This macro is used to exit the primitive when it cannot compute the method result. An example would be a floating point primitive that is passed an argument that is not a floating point number.

oldToNewUpdate

This macro must be invoked after every store of an object pointer into an object. It updates several memory management data structures, if necessary. Failure to invoke this macro can lead to very bizarre and unpredictable results.

getElementSize segReg,offsetReg,resultWordReg,resultByteReg

This macro returns the number of instance variables in the object referred to by **segReg:OffsetReg**. The high order 8 bits of the result is in **resultByteReg** and the low order 16 bits is in **resultWordReg**. Note that this is not the same as the size of the object in bytes.

getByteSize segReg,offsetReg,resultWordReg,resultByteReg

This macro returns the number of bytes, including the header, occupied by the object referred to by **segReg:OffsetReg**. The high order 8 bits of the result is in **resultByteReg** and the low order 16 bits is in **resultWordReg**. Note that this is not the same as the number of instance variables.

isPointerObject segReg,offsetReg

This macro tests whether the object referred to by **segReg:OffsetReg** contains object pointers or bytes. For example, **Strings** contain bytes and **Arrays** contain object pointers. It sets the condition code **zero** if it contains bytes and sets the condition code **non-zero** if it contains object pointers.

isIndexedObject segReg,offsetReg

This macro tests whether the object referred to by **segReg:OffsetReg** contains indexed instance variables. For example, **Strings** and **Arrays** contain indexed instance variables, and **Dictionaries** and **Dates** do not. It sets the condition code **zero** if it does not contain indexable instance variables and sets the condition code **non-zero** if does.

isSmallObject segReg,offsetReg

This macro tests whether the object segment referenced by **segReg:OffsetReg** is contained in a single 64K byte segment. It sets the condition code **zero** if it does not fit in a single segment and sets the condition code **non-zero** if it does.

isSmallPosInt segmentExpression

This macro tests whether the object segment referenced by **segmentExpression** is a *positive SmallInteger*. It sets the condition code **equal** if it is and sets the condition code **non-equal** if it is not.

isSmallNegInt segmentExpression

This macro tests whether the object segment referenced by **segmentExpression** is a *negative SmallInteger*. It sets the condition code **equal** if it is and sets the condition code **non-equal** if it is not.

interruptVM

This macro places the interrupt number contained in **AL** onto the interrupt queue of the interpreter. It is used by primitives that need to issue Smalltalk interrupts. This macro is used in protected mode primitives. There is another macro for use in real mode interrupt service routines (described below).

Interrupt Service Routines

Smalltalk/V 286 lets you provide your own interrupt service routines, for example your own communications driver. Interrupt service routines are different from primitives in that they are not callable by the Smalltalk interpreter. They are entered in response to a machine interrupt. They also run in real mode instead of protected mode. This means that they do not have access to object memory. They do have access to all DOS and BIOS interrupts. They communicate with Smalltalk via virtual machine interrupts and via memory shared in the segment with Smalltalk primitives. It is for this reason that they are included in the same primitive module as the Smalltalk primitives that access the shared memory.

Interrupt service routines cannot store or refer to object pointers in any way. There is no guarantee as to the state of the garbage collector at the time of the hardware interrupt, so the pointers *may not be valid*.

The following macro is included in file **access.usr** to allow interrupt service routines to issue Smalltalk virtual machine interrupts.

ISVinterruptVM

This macro places the interrupt number contained in **AL** onto the interrupt queue of the interpreter. It is used by interrupt service routines that wish to issue Smalltalk interrupts, for instance a communications driver. This macro is only used by interrupt service routines and is only invokable from real mode.

Constructing Primitive Modules

A primitive module is composed of four pieces that are assembled or linked into a single memory image segment that starts at offset 0. These are the module header, the initialization routine, one or more primitive subroutines and interrupt service routines, and the entry point table. The file **example.prm** shows the assembly language source code for a complete primitive module. To build the primitive module, first assemble this source file into the file **example.obj**. Then link it with the linker to produce the executable file **example.exe**. Finally use the **exe2bin** program to turn it into the memory image file **example.bin**. Here are some sample DOS commands:

```
masm example.prm;
link example.obj;
exe2bin example.exe
```

The header must be the first 16 bytes of the segment when it is loaded into memory. The format of the 16 byte module header is:

- 0: module initialization entry point offset
- 2: reserved
- 4: reserved
- 6: physical segment address (filled in when loaded)
- 8: offset of table of primitive entry point offsets
- 10: physical segment address of interpreter parameter area (filled in when loaded)
- 12: reserved
- 14: reserved

The installation routine for the primitive module is entered via a far call after the module is loaded. This allows any load time initialization to be done by the module before any of the primitives are called.

A primitive is entered via a far call when the associated Smalltalk method is invoked. The primitive must either succeed or fail. If it succeeds, the resultant object pointer is returned in the **DX,AX** register pair, with **DX** holding the segment part and **AX** the offset part. If it fails, **DX** and **AX** must be set to zero.

User primitives are identified by names instead of numbers. The primitive entry point table contains the names and offset of all the primitives in the module. It does not have any of the interrupt service routines since these are not callable from Smalltalk. The

format of the entry point table is:

```

DB  'name of primitive 1'
DB  0
DW  offset of entry point 1
DB  'name of primitive 2'
DB  0
DW  offset of entry point 2
:
:
DW  0 ;marks end of table

```

Invoking User Primitives from Smalltalk

Whenever a method is invoked that gives the user primitive as its implementation, the user primitive is entered via a far call. User primitives are referred to by the name given them in the containing primitive module. For example, the sample primitive is named `userStringAt:` so the following method could be added to class `String`.

```

examplePrimAt: index
    "This method invokes the sample primitive in the file example.bin."
    <primitive: userStringAt: >
    ^self error: 'user prim failed'

```

After the primitive module `example.bin` is loaded, the following expression would invoke the user primitive:

```
'Now is the time' examplePrimAt: 5
```

Appendix 3: Configuring Smalltalk/V 286

This appendix presents information for configuring Smalltalk/V 286 for different memory, hardware, and BIOS configurations. If you are having trouble starting the system up for the first time and suspect hardware or software incompatibilities, please read the *read.me* file for up-to-date information.

Smalltalk/V is configured by adding parameters to the DOS command line that invokes the Smalltalk environment. There are five possible configuration parameters: three deal with memory configurations and two deal with hardware and BIOS configuration. Briefly, these parameters are (they are described in more detail in the following sections):

/d:nnn — Reserve **nnn** Kbytes (decimal) of memory for DOS shell and primitives in the address space of DOS.

/x:nnn — Reserve **nnn** Kbytes (decimal) of memory in the beginning of extended memory for other protected mode programs to use.

/u:nnn — Use a maximum of **nnn** Kbytes (decimal) of extended memory for Smalltalk.

/s:x where **x** is either **s** or **h** — Use hardware (**h**) or software (**s**) shutdown logic to switch to real mode from protected mode.

/t:x where **x** is either **p** or **s** — Use either primary (**p**) or secondary (**s**) startup logic to start up in real mode after a shut down in protected mode.

When more than one parameter is specified they may be separated by blanks. Letters may be either upper or lower case. Numeric arguments are always decimal (base 10). There cannot be a space between the : and the argument in a parameter. The following are both valid command lines:

```
v /D:100 /s:h  
V /t:P /U:2048
```

Memory Configuration

Unless told otherwise, Smalltalk/V 286 will use all of available memory. Logically speaking there are two kinds of memory available when the interpreter is invoked, available DOS memory (memory accessible in real mode) and available extended memory (memory accessible in protected mode only).

DOS Memory

If you are going to use the DOS shell or if you are going to load user primitive modules then you must reserve some of the DOS memory. The **/d:nnn** parameter specifies the amount of DOS memory to reserve in 1K (1024) byte increments. The **nnn** is in decimal.

When **Smalltalk/V 286** is invoked, it allocates parts of itself in extended memory if needed to reserve the amount of space requested. However certain parts must be in real mode memory (approximately 100K).

Extended Memory

Smalltalk/V 286 allocates its extended memory starting at the end of available extended memory (high physical address) and going forward (towards low physical address). There are two ways to control the amount of memory allocated in extended memory.

The **/u:nnn** parameter specifies the maximum amount of extended memory that **Smalltalk/V 286** is to allocate in 1K byte increments. The **nnn** is in decimal. This sets an upper bound. If less memory is available, less will be allocated.

The **/x:nnn** parameter specifies the amount of extended memory not to allocate at the front (low physical address) of extended memory in 1K byte increments. The **nnn** is in decimal. This reserves a portion of extended memory for other protected mode programs to use.

Note that **Smalltalk/V 286** detects the memory at the front of extended memory (low physical address) used by VDISK or multiple VDISKS and automatically treats the memory as not available. The parameters above only refer to available memory and not VDISK memory.

An Example

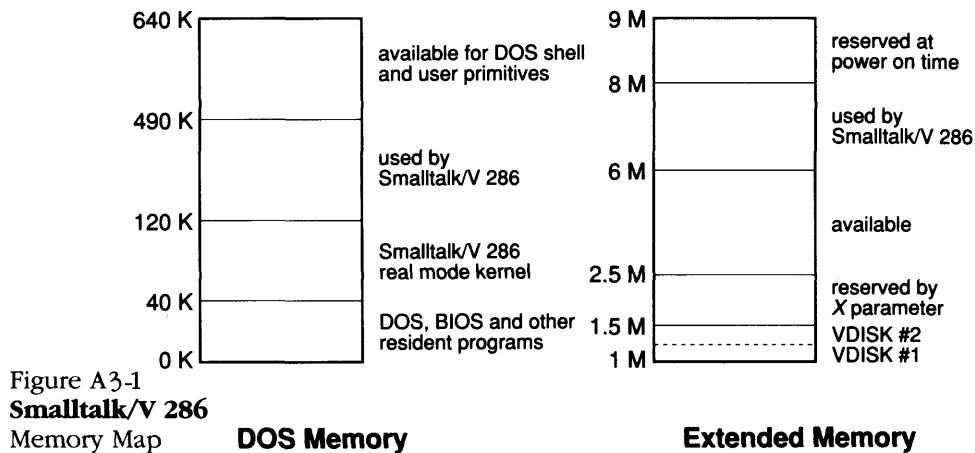
As an example, let's take the following situation:

1. 150K of DOS memory needs to be reserved for user primitives.
2. Two VDISKS of 256K each were created when the system was booted up.
3. A one megabyte area of extended memory was reserved at the high end of extended memory when the system was booted up.
4. At least an additional one megabyte needs to be reserved for use after **Smalltalk/V 286** is started up; more would be nice.
5. There are 8 megabytes of extended memory in the hardware.
6. The **Smalltalk/V** image we are going to run does not need more than two megabytes.

The command line would be:

```
v /d:150 /x:1024 /u:2048
```

And the memory map is given in figure A3-1.



Hardware and BIOS Configuration

Smalltalk/V 286 dynamically determines the configuration of your hardware and the type of BIOS installed when it is invoked. Due to the variety of hardware and manufacturers available, we included the following two parameters in case we have overlooked something. See file **read.me** for latest information about configuring for specific machines.

80386 cpu vs 80286 cpu

Since **Smalltalk/V 286** runs in protected mode, and all of the BIOS and DOS facilities are accessible only from real mode, switching from protected to real mode is necessary. If you have a **80386** cpu, this switching is done via 80386 software instructions. **Smalltalk/V** detects the presence of the **80386** cpu automatically.

If you have an **80286** cpu, then the switching between protected and real mode is accomplished by using the **80286** cpu to shutdown and restart in real mode. There are two issues, how to accomplish the shutdown and how the cpu is to start up.

80286 Shutdown

The /s: parameter specifies the type of shutdown to use. There are two choices: shutdown via hardware, /s:h, and via software, /s:s.

The hardware shutdown is done by intentionally causing what is commonly known as a triple fault condition in the 80286 cpu. This does require that the mother board cause an automatic restart condition when the 80286 enters the shutdown mode. All IBM PC/AT machines and most clones do this.

The software shutdown is done by requesting a cpu reset via software. This involves sending commands to the keyboard processor on AT class machines and issuing port I/O commands on some PS/2 machines.

80286 Restart

The /t: parameter specifies how the 80286 is to start up after a shutdown. This is done by writing a value into the CMOS memory (non-volatile mnemory) of the machine prior to the shutdown. The BIOS boot up code in the ROM looks at this value to determine the type of restart to do. There are two choices: primary start up, /t:p, and secondary start up /t:s.

The primary start up is done by using the documented BIOS function for starting up after switching from protected to real mode. For early PC/AT's and some clones, this function does not work properly. The secondary start up uses the BIOS block memory move function to cause the real mode start up.

Speed vs Space

The system menu contains a selection, speed/space, which lets you optimize Smalltalk/V for either speed (performance) or space (memory utilization). In speed mode, every window maintains a backup bitmap of itself for use in redrawing. In space mode, when a window is redrawn, it regenerates its screen image.

You can tell which mode the system is in by looking at the system menu. If it says speed/space, you are in speed mode. If it says space/speed, you are in space mode.

Appendix 4: METHOD INDEX

This appendix is an index into **Part 4: Encyclopedia of Classes**. It is a complete index of all the methods implemented in **Smalltalk/V 286**. The first column below contains the selectors of all methods in sorted order. For each selector, the second column contains the list of all classes that implement a method for that selector.

Method Selector	Implementing Classes
&	False, True
*	Float, Fraction, Integer, Number, Point
+	Float, Fraction, Integer, Number, Point
,	IndexedCollection, OrderedCollection
.	Float, Fraction, Integer, Number, Point
/	Float, Fraction, Integer, Number
//	Float, Fraction, Integer, Number, Point
<	Association, Character, Date, Float, Fraction, Integer, Magnitude, Point, String, Time
<=	Association, Character, Date, Float, Fraction, Integer, Magnitude, Point, String, Time
=	Association, Character, Date, Directory, Float, Fraction, IndexedCollection, Integer, Magnitude, Object, Point, String, Symbol, Time
==	Object
>	Association, Character, Date, Float, Fraction, Integer, Magnitude, Point, String, Time
>=	Association, Character, Date, Float, Fraction, Integer, Magnitude, Point, String, Time
@	Number
\`	Float, Fraction, Integer, Number, Point
abs	Number, Point
accept	DiskBrowser, PromptEditor, TextEditor, TextPane
accept:from:	ClassBrowser, ClassHierarchyBrowser, Debugger, DictionaryInspector, Inspector, MethodBrowser
acceptClass:from:	ClassHierarchyBrowser
acceptPrompt	TextPane
acceptReply:from:	Prompter
accessEmptyCollection	IndexedCollection
activate	Dispatcher
activatePane	GraphPane, Pane
activateWindow	Dispatcher, ScreenDispatcher, SubPane, TopPane
active	Dispatcher

Method Selector

add
 add:

 add:after:
 add:afterIndex:
 add:before:
 add:beforeIndex:
 add:name:color:
 add:withOccurrences:
 addAll:
 addAllFirst:
 addAllLast:
 addBreak
 addClasses:at:
 addClassVarName:
 addDays:
 addDependent:
 addFirst:
 addLast:
 addPrimitive:withEntryPoint:
 addSelector:withMethod:
 addSharedPool:
 addSubClass
 addSubclass:
 addSubpane:
 addTime:
 adjustBox
 adjustPoint:
 adjustSize
 after:
 after:ifNone:
 again
 allClasses
 allClassVarNames
 allDependents
 allInstances
 allInstancesPrim
 allInstVarNames
 allReferences
 allReferencesPrim

Implementing Classes

DictionaryInspector
 Bag, Collection, Dictionary, DispatchManager,
 FixedSizeCollection, IdentityDictionary,
 MethodDictionary, OrderedCollection, Set,
 SortedCollection, SymbolSet, SystemDictionary
 OrderedCollection, SortedCollection
 OrderedCollection
 OrderedCollection, SortedCollection
 OrderedCollection
 Animation
 Bag
 Collection, SortedCollection
 OrderedCollection, SortedCollection
 OrderedCollection, SortedCollection
 Debugger
 ClassHierarchyBrowser
 Class
 Date
 Object
 OrderedCollection, SortedCollection
 OrderedCollection, SortedCollection
 Compiler class
 Behavior
 Class
 ClassHierarchyBrowser
 Behavior
 TopPane
 Time
 PointDispatcher
 StringModel
 IdentityDictionary, Set
 OrderedCollection
 OrderedCollection
 TextEditor
 Behavior
 Behavior
 Object
 Behavior
 Behavior
 Behavior
 Object
 Object

Method Selector

allSubclasses
 allSubdirectoriesOf:level:into:
 allSuperclasses
 amountToPageLeft
 amountToPageUp
 amountToScrollLeft
 amountToScrollUp
 and:
 andRule
 appendChar:
 appendText:
 arcCos
 arcSin
 arcTan
 areaOnFormOf:
 argumentCount
 arguments
 arguments:
 asArray
 asArrayOfSubstrings
 asAsciiZ
 asBag
 asCharacter
 asciiValue
 asDate
 asFloat
 asInteger
 asLowerCase
 asOrderedCollection
 asPrinterErrorFlag
 asSeconds
 asSet
 assignClassHash
 associationAt:
 associationAt:ifAbsent:
 associationsDo:
 asSortedCollection
 asSortedCollection:
 asStream
 asString
 asSymbol
 asUpperCase

Implementing Classes

Behavior
 DiskBrowser
 Behavior
 ScrollDispatcher
 ScrollDispatcher
 ScrollDispatcher
 ScrollDispatcher
 ScrollDispatcher
 False, True
 Form class
 StringModel, TextPane
 StringModel, TextPane
 Number
 Number
 Float, Number
 TextPane
 CompiledMethod
 Message
 Message
 Collection
 String
 String
 Collection
 Integer
 Character
 String
 Float, Fraction, Integer
 String
 Character, String
 Collection
 SmallInteger
 Date, Time
 Collection
 Behavior
 Dictionary
 Dictionary, IdentityDictionary
 Dictionary, IdentityDictionary
 Collection
 Collection
 String
 String, Symbol
 String, Symbol
 Character, String

Method Selector

at:
 at;ifAbsent:
 at:put:
 atAll:put:
 atAllPut:
 atEnd
 ATTmono
 backColor
 backColor:
 background:
 backgroundColor:
 backspace
 backspaceSelection
 backup
 backupWindow
 baseDay
 basePoint
 basicAt:
 basicAt:put:
 basicHash
 basicNew
 basicNew:
 basicSize
 become:
 before:
 before;ifNone:
 beginMenu
 beginScroll
 beginSelect
 bell
 between:and:
 biColorForm
 bitAnd:
 bitAt:
 bitBltError:
 bitCoordinate:
 bitInvert
 bitmap
 bitOr:

Implementing Classes

Bag, ColorForm, Dictionary, Form, IdentityDictionary, Interval, Object, OrderedCollection, Set, String
 Dictionary, IdentityDictionary
 Bag, Dictionary, Form, IdentityDictionary, Interval, MethodDictionary, Object, OrderedCollection, Set, SortedCollection, String, Symbol, SystemDictionary
 IndexedCollection
 Bitmap, IndexedCollection
 FileStream, Stream
 DisplayScreen class
 BiColorForm, Form, TopPane
 BiColorForm, Form, TopPane
 DisplayScreen
 DisplayScreen class
 TextEditor
 TextPane
 TopPane
 TopPane
 Date class
 Font
 Object, String
 Object, String
 Integer, Object, True
 Behavior
 Behavior
 Object
 Object
 OrderedCollection
 OrderedCollection
 TerminalStream
 TerminalStream
 TerminalStream
 TerminalStream
 Magnitude, Point
 Form class
 Integer
 Integer
 BitBlt class
 TextSelection
 Integer
 Form
 Integer

Method Selector

bitShift:
 bitXor:
 black
 black:
 blank:width:
 blankRestFrom:
 boldLine:
 border
 border:
 border:clippingBox:rule:mask:
 border:rule:mask:
 bottom
 bounce:
 bounceBall
 boundingBox
 boxOfSize:
 breakpointInterrupt
 broadcast:
 broadcast:with:
 broadcastChangesIn:
 upTo:
 withExcess:
 browse
 browseBreakpoints
 browseBreakpoints:
 browseWalkback
 browseWalkback:
 build:
 buildDirectoryList
 byteCodeArray
 byteCodeArray:
 byteValueAt:put:
 byteValueAtX:Y:
 calendarForMonth:year:
 call:
 cancel
 cancelPrompter
 cantReturn
 canUnderstand:
 ceiling
 center
 center:in:

Implementing Classes

Integer
 Integer
 BiColorForm class, DisplayMedium, Form class, Pen
 DisplayMedium
 CharacterScanner
 CharacterScanner
 ListPane
 Pane
 DisplayMedium, Pane
 DisplayMedium
 DisplayMedium
 Rectangle
 Pen
 DemoClass
 DisplayObject
 Dispatcher
 Process class
 Object
 Object
 StringModel

 ClassHierarchyBrowser
 Debugger
 Debugger
 Debugger
 Debugger
 Debugger
 Pattern, WildPattern
 DiskBrowser
 CompiledMethod
 CompiledMethod
 Form
 ColorForm, Form
 Date class
 Dos
 TextEditor, TextPane
 Prompter
 Object
 Behavior
 Number
 Rectangle
 TopPane

Method Selector

centerText:font:
 change
 change:
 change:from:to:
 changeColor
 changed
 changed:
 changed:with:
 changed:with:with:
 change FileMode
 changeModeOf:to:
 changeNib:
 changeTo:
 characterInput
 charsInColumn
 charsInRow
 charSize
 charWidth:
 checkArgument:
 checkCharacter:
 checkDay:month:year:
 checkDay:year:
 checkDosError:
 checkIndex:
 checkMode:withAspect:
 checkMove
 chkdsk
 class
 class:
 classes
 classField
 classField:
 classPool
 classVariableString
 classVarNames
 classVarNames:
 cleanHandles
 clearScreen
 clipRect
 clipRect:
 clipRectAll:
 clipX

Implementing Classes

Pen
 CursorManager, NoMouseCursor
 SubPane
 DisplayScreen
 Form class
 Object
 Object
 Object
 Object
 DiskBrowser
 File class
 Pen
 NoMouseCursor
 InputEvent, TerminalStream
 GraphPane, SubPane
 SubPane
 Font
 Font
 Context
 String
 Date class
 Date class
 Dos class
 IndexedCollection
 DisplayScreen class
 TopDispatcher
 ScreenDispatcher
 Object
 ClassHierarchyBrowser
 ClassHierarchyBrowser
 CompiledMethod
 CompiledMethod
 Class, MetaClass
 Behavior
 Class, MetaClass
 Class
 FileHandle class
 DispatchManager
 BitBlt
 BitBlt, CharacterScanner
 Commander
 BitBlt

Method Selector

clipY
 clockOffPrimitive
 clockTickPeriod:
 clockTickPrimitive:
 clockTicksOff
 close

 closeIt
 closeWindow
 collapse
 collapsed
 collapsedLabel
 collect:
 collection
 color
 color:
 colorForm
 coloring:
 colors:
 colors:selectors:
 combinationRule:
 compatibleForm
 compatibleMask
 compile:
 compile:in:
 compile:in:notify:ifFail:
 compile:notify:
 compile:notify:in:
 compileAll
 compileAllSubclasses
 compiledMethodAt:
 compileLogic:
 compileLogic:notify:
 compilerError:at:in:for:
 compress:
 compressChanges
 compressChangesOf:into:
 compressSources
 compressSourcesOf:into:
 computeInstSize
 configureAs:withColor:
 containsPoint:

Implementing Classes

BitBlt
 Time class
 Time class
 Time class
 Time class
 File, FileHandle, FileStream, GraphPane, ListPane,
 Pane, Stream, TextPane, TopDispatcher, TopPane
 Dispatcher
 Dispatcher
 TopDispatcher, TopPane
 TopPane
 ClassHierarchyBrowser, DiskBrowser
 Collection, FixedSizeCollection
 WriteStream
 TopDispatcher
 BiColorForm class, ColorForm class, Form class
 Form class
 TopDispatcher
 Menu
 Menu class
 BitBlt
 ColorForm, ColorScreen, Form
 ColorForm, ColorScreen, Form
 Behavior
 Compiler class
 Compiler class, LCompiler class
 Behavior
 ClassHierarchyBrowser
 Behavior
 Behavior
 Behavior
 Behavior
 Behavior
 TextEditor, TextPane
 CompiledMethod class
 SystemDictionary
 SystemDictionary
 SystemDictionary
 SystemDictionary
 Behavior
 DisplayScreen class
 Icon, Menu, Rectangle

Method Selector

contents
 contextFor:
 continue
 continueScroll
 controlBreakInterrupt
 convertToString:
 copy
 copy:from:to:rule:
 copy:to:
 copyBits
 copyChars
 copyFile
 copyFrom:to:
 copyFrom:to:into:
 copyReplaceFrom:to:with:
 copySelection
 copyStack
 copyWith:
 copyWithout:
 corner
 corner:
 cornerFromUserOfOrigin:
 minExtent:
 cos
 countBlanks
 cr
 create
 create:
 createDirectory
 createFile
 createIcons:
 crossHair
 current
 currentDateInto:
 currentDisk
 currentTimeInto:
 cursorIn:
 cursorKey:
 cursorMoved
 cursorOffset:

Implementing Classes

Directory, ReadStream, ReadWriteStream, Set, Stream, TextEditor, TextPane, WriteStream
 Process
 ScreenDispatcher
 ScrollDispatcher, TerminalStream
 Process class
 StringModel
 Object
 Form
 File class
 BitBlt
 CharacterScanner
 DiskBrowser
 FileStream, IndexedCollection, OrderedCollection, SortedCollection, Stream
 FileStream
 FixedSizeCollection, IndexedCollection
 TextEditor
 Process class, ProcessScheduler
 IndexedCollection
 IndexedCollection
 CursorManager class, Rectangle, TextSelection
 Point, Rectangle, TextSelection
 PointDispatcher class
 Float, Number
 Stream
 Debugger, FileStream, TextEditor, WriteStream
 Directory
 Directory class
 DiskBrowser
 DiskBrowser
 TopPane
 CursorManager class
 Directory class
 Date class
 Directory class
 Time class
 SubPane
 CursorManager
 TerminalStream
 PointDispatcher

Method Selector

cursorOut:
 cutSelection
 cycle
 cyclePane
 cyclePane:
 darkGray
 dateAndTimeNow
 day
 day:
 dayIndex
 dayName
 dayOfMonth
 dayOfWeek:
 dayOfYear
 daysInMonth
 daysInMonth:forYear:
 daysInYear
 daysInYear:
 daysLeftInMonth
 daysLeftInYear
 deactivate
 deactivatePane
 deactivateWindow
 debug
 debugger
 debugger:
 decompress:
 deepCopy

 defaultDispatcherClass
 defaultNib:
 degreesToRadians
 delay
 delete:
 deleteCharIn:
 demoMenu
 denominator
 dependents
 dependsOn:
 destForm
 destForm:
 destForm:sourceForm:

Implementing Classes

Dispatcher
 TextEditor
 Dispatcher, DispatchManager, ScreenDispatcher
 Dispatcher, Pane, ScreenDispatcher
 TopPane
 BiColorForm class, Form class
 Date class, Time class
 Date
 Dispatcher
 GraphPane, ListPane, Pane, TextPane, TopPane
 Dispatcher, Pane
 Debugger
 Process
 Process
 CompiledMethod class
 Behavior, Boolean, Character, Collection, Dictionary,
 Float, Integer, Object, Symbol, UndefinedObject
 GraphPane, ListPane, TextPane, TopPane
 Pen
 Float, Number
 TerminalStream
 StringModel
 StringModel
 DemoClass
 Fraction, Number
 Object
 Object
 BitBlt
 BitBlt
 BitBlt
 BitBlt, BitBlt class

<u>Method Selector</u>	<u>Implementing Classes</u>
destForm:	BitBlt
sourceForm:	
halftone:	
combinationRule:	
destOrigin:	BitBlt
sourceOrigin:	
extent:	
clipRect:	
destOrigin:	BitBlt
destRect:	BitBlt
destX	BitBlt, Commander
destX:	BitBlt
destY	BitBlt, Commander
destY:	BitBlt
detect:	Collection
detect;ifNone:	Collection
deviceType:	Form
dictionaries	ClassBrowser
dictionary:	ClassBrowser
digitValue	Character
digitValue:	Character class
dir	ScreenDispatcher
direction	Pen
direction:	Commander, Pen
directories	DiskBrowser
directory	DiskBrowser, File
directory:	DiskBrowser
directoryListMenu	DiskBrowser
directorySort	DiskBrowser
disappear	Menu
diskLabel	ScreenDispatcher
dispatcher	Pane
dispatcher:	Pane
dispatchers	DispatchManager
display	Animation, CursorManager, Dispatcher, DispatchManager, DisplayObject, Icon, NoMouseCursor, TextSelection
display:	StringModel
display:at:	CharacterScanner
display:from:at:	CharacterScanner
display:from:to:at:	CharacterScanner
display:reverseFrom:to:	TextPane

Method Selector

displayAll
 displayAll:from:to:at:
 displayAt:
 displayAt:clippingBox:
 displayAt:font:
 displayBox:
 displayChanges
 displayForm:at:rule:
 displayGap
 displayIn:
 displayLabel
 displayLabelIcons
 displayOn:
 at:
 clippingBox:
 rule:
 mask:
 displayPage2
 displayPatch:
 displayScreen
 displaySelection
 displayWindow
 do:

 doesNotHandle
 doesNotUnderstand:
 doIt
 Doit
 doIt:
 doItResult:error:
 dosError:
 dosMenu
 dosPrimitive:registers:value:
 dotProduct:
 down
 downArrow
 dragon
 dragon:
 drawBox:
 drawFrom:to:
 drawLoopX:Y:
 drawRectangle

Implementing Classes

DispatchManager, StringModel
 CharacterScanner
 DisplayObject, Menu, String
 DisplayObject
 String
 PointDispatcher
 TextPane
 CharacterScanner
 TextSelection
 Dispatcher
 TopPane
 TopPane
 DisplayObject

 Form class
 TextSelection
 Form class
 TextSelection
 SubPane, TopPane
 Bag, Collection, Dictionary, IdentityDictionary,
 IndexedCollection, OrderedCollection, Set, Stream
 Dispatcher
 Object
 TextEditor, TextPane
 UndefinedObject
 Inspector, TextPane
 Inspector
 Dos class
 ScreenDispatcher
 Dos
 Point
 Commander, CursorManager, Pen
 CursorManager class
 DemoClass
 DemoClass, Pen
 PointDispatcher
 BitBlt
 BitBlt
 Pen

Method Selector	Implementing Classes
drive	Directory
drive:	Directory
driveBPresent	ScreenDispatcher
dropFrame	Process
dropSenderChain	Process class
dropTo:	Process
edit	Class, String
EGAcolor	DisplayScreen class
EGAcolorLowRes	DisplayScreen class
EGAlowRes	DisplayScreen class
EGAmono	DisplayScreen class
eightLine	Font class
elapsedDaysSince:	Date
elapsedMonthsSince:	Date
elapsedSecondsSince:	Date
ellipse:aspect:	Commander, Pen
ellipsePrim:aspectX:Y:	Pen
enableInterrupts:	Process class
endByte	FileHandle
environmentVariable:	Dos class
equals:	String
eqv:	False, True
erase	Form class
error:	Object
errorAbsentElement	OrderedCollection
errorAbsentKey	Dictionary
errorAbsentObject	Collection
errorIn:label:	InputEvent
errorInBounds:	IndexedCollection
errorInDay	Date class
errorInMonth	Date class
errorNotIndexable	Behavior, Collection
evaluate:	Compiler class
evaluate:in:to:notifying;ifFail:	Compiler class, LCompiler class
evaluating:	Prompter
even	Number
exchangeColor	Form class
execute	CursorManager class
execute:parameters:	ScreenDispatcher
executeCommands:	ScreenDispatcher
executeProgram:parameters:	ScreenDispatcher
exit	ScreenDispatcher

Method Selector

exit:
 exp
 expandBy:
 exponent
 extendOrigin:
 extendSelect
 extendTo:
 extent
 extent:
 extractDateTimeFrom:
 extractDirNameFrom:
 extractFileNameFrom:
 extractFlagsFrom:
 extractSizeFrom:
 factorial
 failAt:with:
 fanOut
 file
 file:
 fileExtension
 fileId
 fileId:
 fileIn
 fileInFrom:
 fileItIn
 fileListMenu
 fileName
 fileName:extension:
 fileOut
 fileOutOn:
 files
 fill:
 fill:clippingBox:rule:mask:
 fill:rule:mask:
 fillAt:
 fillAtX:andY:
 findCurrentLine
 findElementIndex:
 findFirst:
 findFirst:attribute:buffer:
 findFrameIndexOf:

Implementing Classes

SystemDictionary
 Float, Number
 Rectangle
 Float
 TextSelection
 TerminalStream
 StringModel
 BitBlt, DisplayObject, Form, Rectangle, StringModel,
 TextSelection
 BitBlt, Form, Point, Rectangle
 Directory class
 Directory
 Directory class
 Directory class
 Directory class
 Directory class
 Integer
 Pattern
 Commander
 DiskBrowser, FileStream
 Directory, DiskBrowser
 String
 File
 File
 Stream
 ClassReader, StringModel, TextPane
 TextEditor
 DiskBrowser
 String
 File class
 ClassHierarchyBrowser
 Class, ClassReader, StringModel, TextPane
 DiskBrowser
 DisplayMedium
 DisplayMedium
 DisplayMedium
 Pen
 Pen
 ListPane
 Set, SymbolSet
 IndexedCollection
 Directory
 Process

Method Selector

findKeyIndex:
 findLast:
 findNext:
 first
 firstDayInMonth
 firstDayOfMonth
 fixedWidth
 floatError
 floor
 flush
 flushFromCache:
 font
 for:
 forceEndOntoDisplay
 forceSelectionOntoDisplay
 forClass:
 foreColor
 foreColor:
 foreColor:backColor:
 forgetImage
 fork
 fork:
 fork:at:
 forkAt:
 form
 form:
 formatted
 formCoordinates:
 formPrint
 fourteenLine
 frame
 frame:
 frameAt:offset:
 frameAt:offset:put:
 frameBiasUnit
 frameIndexAt:
 frameOffset
 frameOffset:
 frameToProcessIndex:
 framingBlock:
 framingRatio:
 freeDiskSpace

Implementing Classes

Dictionary, IdentityDictionary
 IndexedCollection
 Directory
 IndexedCollection
 Date
 Date
 Font
 Float class
 Number
 File, FileStream
 MethodDictionary
 CharacterScanner, Pane
 StringModel class
 TextPane
 TextPane
 ClassReader class
 BiColorForm, Form, TopPane
 BiColorForm, Form, TopPane
 BiColorForm, BiColorForm class, Form class
 ScreenDispatcher
 Context
 ProcessScheduler
 ProcessScheduler
 Context
 GraphPane, Icon
 GraphPane, Icon
 Directory
 TextPane
 Date
 Font class
 CharacterScanner, Icon, Pane, Pen, TextPane, TopPane
 Pen, StringModel
 Process
 Process
 Process class
 Process
 HomeContext
 HomeContext
 Process
 SubPane
 SubPane
 Directory

Method Selector

from:to:
 from:to:by:
 fromDays:
 fromDisplay
 fromDisplay:
 fromInteger:
 fromSeconds:
 fromString:
 fullDirName
 functionInput
 gcd:
 getBits:
 getCurrentPen:
 getDate
 getDevices
 getEnvironment:keyWord:
 getIndex:
 getPointAt:
 getSelectionFrom:to:
 getSourceClasses
 glyphs
 go:
 goto:
 gotoDos
 gray
 gray:
 graySelection
 grid:
 grow
 growSize
 growSymbolHashArray
 growTo:
 halt
 hand
 hasBlock
 hasCursor
 hash
 hash:

Implementing Classes

Interval class
 Interval class
 Date class
 Form
 BiColorForm, Form, Form class
 Float class
 Time class
 Date class
 Directory
 InputEvent, TerminalStream
 Integer
 ColorForm, Form
 Animation
 File
 ScreenDispatcher
 Dos class
 Font
 StringModel
 StringModel
 SystemDictionary
 Font
 Commander, Pen
 Commander, Pen
 ScreenDispatcher
 BiColorForm class, DisplayMedium, Form class, Pen,
 TextSelection
 CharacterScanner, DisplayMedium
 GraphPane, ListPane, SubPane, TextPane, TopPane
 Pen
 Dictionary, IdentityDictionary, IndexedCollection,
 OrderedCollection, Set, SortedCollection, SymbolSet,
 WriteStream
 IndexedCollection
 SymbolSet
 OrderedCollection
 Object
 CursorManager class
 CompiledMethod
 Pane, TopPane
 Association, Character, Date, Float, Fraction, Integer,
 Magnitude, Object, Point, String, Symbol, Time, True
 Object

Method Selector	Implementing Classes
hasSignals	Semaphore
hasSubdirectory	Directory
hasZoomedPane	Pane
height	BitBlt, DisplayObject, Font, Form, Rectangle, TextSelection
height:	BitBlt, Rectangle
hercules	DisplayScreen class
hide	CursorManager, Icon, NoMouseCursor, TextSelection
hideCursor	BitBlt
hideDirectory	DiskBrowser
hideGap	SubPane, TextPane, TextSelection
hideGapFlag	SubPane, TextPane, TextSelection
hidelIconsExcept:	TopPane
hideSelection	ListPane, TextPane, TextSelection
hideSelection:to:	TextSelection
hideShow	ClassHierarchyBrowser, DiskBrowser
hideX:y:width:height:	NoMouseCursor
hierarchy	ClassHierarchyBrowser
hierarchy:	ClassHierarchyBrowser
highlightLabel	TopDispatcher, TopPane
highUserPriority	ProcessScheduler
home	Pen
homeContext	Context
homeCursor	Dispatcher, Pane, TopDispatcher
homeFrameOf:	Process
hop	Debugger
hotSpot	CursorManager
hours	Time
IBM3270	DisplayScreen class
icAt:	Process
iconsWidth:	TopPane
ifFalse:	False, True
ifFalse;ifTrue:	False, True
ifTrue:	False, True
ifTrue;ifFalse:	False, True
image	CursorManager
implementedBySubclass	Object
implementors	ClassBrowser, ClassHierarchyBrowser, Debugger, MethodBrowser
implementorsOf:	Behavior, SystemDictionary
includes:	Bag, Collection, Dictionary, DispatchManager, IndexedCollection, OrderedCollection, Set

Method Selector

includes:with:
 includesKey:
 includesSelector:
 increment
 indexOf:
 indexOf:ifAbsent:
 indexOfMonth:
 inheritsFrom:
 initBegin:end:incr:
 initDependents
 initFlags
 initForm:hotSpot:
 initHandles
 initialize

 initialize:
 initialize:font:
 initialize:font:dest:
 initialize:hotSpot:
 initializeClass
 initializeMouse
 initializeTranscript
 initialSize
 initialState
 initLabelIcons
 initPen:
 initPositions:
 initScanner
 initSystem
 initTopCorner
 initWindowClip
 initWindowSize

 inject:into:
 input

Implementing Classes

CompiledMethod
 Dictionary, IdentityDictionary
 Behavior
 Interval
 IndexedCollection
 IndexedCollection
 Date class
 Behavior
 Interval
 Object class
 TextSelection
 CursorManager
 FileHandle class
 Bag, Behavior class, Class, Date class, Dispatcher,
 DispatchManager, DispatchManager class, Dos,
 GraphPane, Icon, InputEvent, ListPane,
 MethodDictionary class, NoMouseCursor, Object class,
 Pane, Pattern class, PointDispatcher, Process,
 ProcessScheduler, ScreenDispatcher class, Semaphore,
 TerminalStream, TextEditor, TextEditor class,
 TextPane, TextSelection, Time class, TopDispatcher
 class, TopPane
 Animation, Commander, Set
 CharacterScanner
 CharacterScanner
 CursorManager
 Class
 CursorManager, NoMouseCursor
 TextEditor class
 IdentityDictionary class
 TerminalStream
 TopPane class
 Pen
 OrderedCollection
 GraphPane
 DisplayScreen class
 GraphPane
 Pane class
 ClassHierarchyBrowser, Debugger, DiskBrowser,
 MethodBrowser, TopDispatcher
 Collection
 Pattern

Method Selector

insetBy:
 inspect
 inspectMenu
 inspectSelection
 installFixedSize:
 charSize:
 startChar:
 endChar:
 basePoint:
 instance
 instance:
 instanceClass
 instanceHeaderOn:
 instances
 instanceVariableString
 instSize
 instVarAt:
 instVarAt:put:
 instVarList
 instVarNames
 instVarNames:
 integerCos
 integerSin
 intern:
 internalForm
 interrupt:
 interruptFrame:
 intersect:
 intersects:
 invalidMessage
 ioErrorInterrupt
 isAlphaNumeric
 isBefore:
 isBits
 isBytes
 isContext
 isControlActive
 isControlWanted
 isDigit
 isEmpty
 isFixed
 isGap

Implementing Classes

Rectangle
 Dictionary, Object
 DictionaryInspector, Inspector
 Debugger, DictionaryInspector, Inspector
 Font
 Debugger, DictionaryInspector, Inspector
 ClassHierarchyBrowser
 MetaClass
 ClassReader
 ClassHierarchyBrowser
 Behavior
 Behavior
 Object
 Object
 DictionaryInspector, Inspector
 Behavior
 Behavior
 Number
 Number
 Symbol class
 Form class
 Dos, Process class
 Process
 Rectangle, TextSelection
 Rectangle
 Object
 Process class
 Character
 Point
 Behavior
 Behavior
 Context, Object
 Dispatcher, PromptEditor, TextEditor, TopDispatcher
 Dispatcher
 Character
 Collection, Stream
 Behavior
 TextSelection

Method Selector

isGapSelection
 isHidden
 isKindOf:
 isLetter
 isLowerCase
 isMemberOf:
 isNil
 isPointers
 isSeparator
 isThereInput
 isUpperCase
 isUserIF
 isVariable
 isVowel
 isZoomed
 jump
 jumpDown
 jumpLeft
 jumpRight
 jumpUp
 key
 key:
 key:value:
 keyWithValue:
 keyWithValue:ifAbsent:
 keyboardInterrupt
 keys
 keysDo:
 kindOfSubclass
 label

 label:
 labelArray:lines:selectors:
 labelFrame
 labels:lines:
 labels:lines:selectors:
 last
 launch
 lcm:
 leapYear:
 leapYearsTo:
 left

Implementing Classes

TextPane
 Icon
 Object
 Character
 Character
 Object
 Object, UndefinedObject
 Behavior
 Character
 CursorManager
 Character
 Process
 Behavior
 Character
 TextPane
 Debugger
 CursorManager
 CursorManager
 CursorManager
 CursorManager
 Association
 Association, Association class
 Association class
 Dictionary
 Dictionary, IdentityDictionary
 Process class
 Dictionary, IdentityDictionary
 Dictionary
 Behavior
 ClassHierarchyBrowser, Debugger, DiskBrowser,
 MethodBrowser, TopDispatcher, TopPane
 Debugger, MethodBrowser, TopPane
 Menu class
 TopPane
 Menu
 Menu class
 IndexedCollection
 SystemDictionary
 Integer
 Date class
 Date class
 CursorManager, Rectangle

Method Selector

leftArrow
 leftButton:
 leftIcons:
 leftPartBefore:
 lightGray
 lineAt:
 lineDelimiter
 lineDelimiter:
 lineInPane:
 linesIn:
 lineToRect:
 lineUpFrom:to:
 listMenu
 listString:
 literal:
 ln
 loadEntireFile
 loadPrimitivesFrom:
 location
 log:
 logEvaluate:
 logSource:forClass:
 logSource:forSelector:inClass:
 lookUpKey:
 lowRes
 lowUserPriority
 magnify:by:
 magnifyBy:
 makeCurrent
 makeLabel:
 makeSelectionVisible
 makeUserIF
 mandala
 mandala:diameter:
 mask
 mask:
 maskForm:
 match:
 match:index:
 matchBlock:
 max:
 maxLineBetween:and:

Implementing Classes

CursorManager class
 TerminalStream
 TopPane
 StringModel
 BiColorForm class, Form class
 ListPane, StringModel, TextPane
 FileStream, Stream
 FileStream, Stream
 ListPane
 StringModel
 ListPane
 Commander
 MethodBrowser
 MethodBrowser
 CompiledMethod, MethodBrowser
 Float, Number
 DiskBrowser
 SystemDictionary
 Commander, Pen
 Number
 SystemDictionary
 SystemDictionary
 SystemDictionary
 Dictionary
 DisplayScreen class
 ProcessScheduler
 Form
 String
 Directory
 Debugger
 TextPane
 Process
 DemoClass
 Pen
 BitBlt
 BitBlt
 BiColorForm
 Pattern, WildPattern
 Pattern, WildPattern
 Pattern
 Magnitude, Point
 ListPane, StringModel

Method Selector

menu
 menu:
 merge:
 message:
 method
 method:
 methodAt:
 methodAt:put:
 methodDictionaries
 methodDictionaries:
 methodDictionary
 methodList
 methods
 millisecondClockValue
 millisecondsToRun:
 min:
 minBoxExtent:
 minimumSize
 minimumSize:
 minutes
 misc
 model
 model:
 modified
 modified:
 monthIndex
 monthName
 monthNameFromString:
 mouseButton
 mouseButtonDown
 mouseButtonUp
 mouseClockValue
 mouseMove
 mouseOffset
 mousePrimWith:
 mousePrimWith:with:
 mousePrimWith:with:with:
 mouseScroll
 mouseSelectOn
 mouseStillDown
 move

Implementing Classes

ClassHierarchyBrowser, Debugger, Prompter,
 TextEditor class, TopDispatcher class
 Pane
 Rectangle, TextSelection
 Menu class
 Debugger
 MethodBrowser
 Process
 Process
 Behavior
 Behavior
 Behavior
 Behavior
 MethodBrowser
 Behavior
 Time class
 Time class
 Magnitude, Point
 PointDispatcher
 TopPane
 TopPane
 Time
 TextEditor
 Pane
 Pane
 Dispatcher, TextEditor
 TextEditor
 Date
 Date
 Date class
 InputEvent
 InputEvent, TerminalStream
 InputEvent, TerminalStream
 Time class
 InputEvent, TerminalStream
 TerminalStream
 CursorManager
 CursorManager
 CursorManager
 ScrollDispatcher
 TerminalStream
 TerminalStream
 TopDispatcher

Method Selector

move:
 move:by:
 moveBy:
 moveCursor:
 moveDownIn:
 moveOrSizeBox:
 moveTo:
 moveUpIn:
 multiEllipse
 multiMandala
 multiPentagon
 multiPolygon:
 multiSpiral
 mustBeBoolean
 mustBeSymbol:
 name
 name:
 name:
 environment:
 subclassOf:
 instanceVariableNames:
 variable:
 words:
 pointers:
 classVariableNames:
 poolDictionaries:
 comment:
 changed:
 nameOfDay:
 nameOfMonth:
 negated
 negative
 new

TopPane
 Animation
 Point, Rectangle
 Menu
 SubPane
 PointDispatcher
 Rectangle
 SubPane
 DemoClass
 DemoClass
 DemoClass
 DemoClass
 DemoClass
 Object
 Symbol class
 Class, File, Icon, MetaClass, Process
 Icon, Process, SubPane
 MetaClass

Date class
 Date class
 Float, Fraction, Integer, Number, Point
 Number
 Bag class, Behavior, BiColorForm class, Boolean class,
 Character class, ColorForm class, Dispatcher class,
 DispatchManager class, DisplayScreen class, Dos class,
 Form class, Icon class, IdentityDictionary class,
 InputEvent class, NoMouseCursor class, Number class,
 OrderedCollection class, Pane class, Pen class,
 PointDispatcher class, Process class, Semaphore class,
 Set class, TextSelection class, UndefinedObject class

Implementing Classes

Method Selector

new:
 newDay:month:year:
 newDay:year:
 newdropFrame
 newFile:
 newLabel
 newMethod
 newNameSymbol:
 newPage2
 newSize:
 next
 next:
 next:put:
 nextByte
 nextBytePut:
 nextChunk
 nextChunkPut:
 nextEvent
 nextFourBytesPut:
 nextFrameAt:
 nextLine
 nextMatchFor:
 nextPiece
 nextPut:
 nextPutAll:
 nextTwoBytesPut:
 nextWord
 noChanges
 noGraphPane
 nonIntersections:
 normal
 north
 not
 notEmpty
 notifier:content:at:

Implementing Classes

Behavior, Boolean class, Commander class,
 IdentityDictionary class, Number class,
 OrderedCollection class, Pattern class, Pen class, Set
 class, SortedCollection class, Symbol class,
 UndefinedObject class, WildPattern class
 Date class
 Date class
 Process
 Directory
 TopDispatcher
 ClassHierarchyBrowser
 Behavior, MetaClass
 DisplayScreen class
 IdentityDictionary class
 FileStream, ReadStream, ReadWriteStream,
 TerminalStream
 Stream
 Stream, TextEditor
 ReadWriteStream
 WriteStream
 Stream
 Stream
 InputEvent
 WriteStream
 Process
 FileStream, Stream
 Stream
 Stream
 Debugger, FileStream, ReadWriteStream,
 TerminalStream, TextEditor, WriteStream
 Debugger, FileStream, ReadWriteStream,
 TerminalStream, TextEditor, WriteStream
 WriteStream
 Stream
 TextPane
 TopPane
 Rectangle
 CursorManager class
 Pen
 False, True
 Collection
 GraphPane class

Method Selector

notifier:content:at:menu:
 notNil
 now
 nullEvent
 numerator
 numerator:denominator:
 occurrencesOf:
 odd
 offset
 offset:
 on:
 open
 open:in:
 openChangeLogIn:
 openClassBrowser
 openDiskBrowser
 openIn:
 openOn:
 openWindow
 openWorkspace
 or:
 origin
 origin:
 origin:corner:
 origin:extent:
 orRule
 orThru
 other
 outByte:toPort:
 output:head:tail:
 output:head:tail:headSize:
 outputToPrinter
 outputToPrinterUpright
 outputToPrinterUpright:
 over
 overClickDelay
 overrunInterrupt
 pageSize
 pageSize:

Implementing Classes

GraphPane class
 Object, UndefinedObject
 Time class
 InputEvent, TerminalStream
 Fraction, Number
 Fraction, Fraction class
 Bag, Collection, Dictionary, Set
 Number
 CursorManager, DisplayObject, Form
 CursorManager, DisplayObject, Form, NoMouseCursor
 Stream class
 Dispatcher, File, ListPane, SubPane, TextPane,
 TopPane
 File class, FileHandle class
 SystemDictionary
 ScreenDispatcher
 ScreenDispatcher
 Dispatcher, FileHandle
 ClassBrowser, ClassHierarchyBrowser, DiskBrowser,
 Inspector, MethodBrowser
 Dispatcher
 ScreenDispatcher
 False, True
 CursorManager class, Rectangle, TextSelection
 Icon, TextSelection
 Rectangle, Rectangle class, TextSelection, TextSelection
 class
 Rectangle, Rectangle class
 Form class
 Form class
 ScreenDispatcher
 Dos
 Form
 Form
 DisplayScreen, Form, String
 DisplayScreen, Form
 Form
 Form class
 TerminalStream
 Process class
 File class
 File class

Method Selector

pane
 pane:
 paneScanner
 pasteSelection
 pathName
 pathName:
 pathName:in:
 peek
 peekFor:
 peekFrom:
 perform:
 perform:with:
 perform:with:with:
 perform:with:with:with:
 perform:withArguments:
 performMenu
 pi
 place:
 pointer:word:variable:
 pointFromUserDisplaying:offset:
 poke:to:
 polygon:sides:
 popUp:
 popUp:at:
 popUpAt:
 popUpAt:for:
 position
 position:
 positionsOf:in:
 positionsOf:in:notifying;ifFail:
 positive
 previousWeekday:
 primitive:
 primitiveClose
 primitiveFailed
 primitiveGetEvent
 primitiveLoadModule:
 primitiveNextPut:
 primitiveNumber
 print
 printerMode:
 printFile

Implementing Classes

Dispatcher
 Dispatcher, TextSelection
 Pane
 TextEditor
 Directory, FileStream
 Directory, Directory class, File class
 File class
 Stream
 Stream
 Dos
 Object
 Object
 Object
 Object
 Object
 Object
 SubPane, TextPane, TopPane
 Float class
 Commander, Pen
 Class
 PointDispatcher class
 Dos
 Pen
 Pane
 Pane
 Menu
 Menu
 CursorManager, FileStream, Stream
 FileStream, Stream, WriteStream
 Compiler class
 Compiler class
 Number
 Date
 CompiledMethod
 FileHandle
 Object
 InputEvent
 SystemDictionary
 FileStream
 CompiledMethod
 TextEditor, TextPane
 Form class
 DiskBrowser

Method Selector

printFraction:
printIt
printLimit
printOn:

printOn:base:
printPaddedTo:
printRecursionOn:
printRounded:
printString
priority
priority:
privateAdd:
processControlKey:
processFunctionKey:

processInput
processInputKey:
processKey:
processLastInput:
prompt:default:
prompt:defaultExpression:
promptForPathName
promptWithBlanks:default:
purgeUnusedSymbols
putHeaderOf:into:
putMethod:withIndex:to:
putSpaceAfter:
putSpaceAtEnd
putSpaceAtStart
queueWalkback:
 makeUserIF:
 resumable:
quo:
radiansToDegrees
radix:
raisedTo:
raisedToIntegér:
read

Implementing Classes

Number
TextEditor
Collection
Array, Association, Behavior, Boolean, Character, Collection, Date, Float, Fraction, Integer, Number, Object, Point, Rectangle, SmallInteger, String, Symbol, Time, UndefinedObject
Integer
Integer
Object
Number
Object
Process
Process
SymbolSet
Dispatcher, PromptEditor, TextEditor
Dispatcher, GraphDispatcher, PointDispatcher, PromptEditor, ScreenDispatcher, ScrollDispatcher, TextEditor
Dispatcher, PointDispatcher, TextEditor, TopDispatcher
Dispatcher, TextEditor
Dispatcher, PointDispatcher
Dispatcher
Prompter, Prompter class
Prompter class
DiskBrowser
Prompter class
Symbol class
SystemDictionary
SystemDictionary
OrderedCollection
OrderedCollection
OrderedCollection
Process class

Integer, Number
Float, Number
Integer
Number
Number
TerminalStream

Method Selector

readBuffer:atPosition:
 readInto:atPage:
 readInto:atPage:pageSize:
 readInto:atPosition:
 readLimit
 receiverAt:
 reciprocal
 recompile:
 recover:
 recoverLine:
 redraw
 reflectDrawX:Y:
 reframe
 reframe:
 reframeLabel
 refreshAll
 refreshColor
 refreshFrom:for:atX:Y:
 registers
 rehash
 rehashFrom:
 reinitialize
 reject:
 release
 rem:
 remove
 remove:
 remove;ifAbsent:
 removeAll:
 removeAssociation:
 removeBreak
 removeClassVarName:
 removeDirectory
 removeFile
 removeFirst
 removeFromSystem
 removeIndex:
 removeKey:
 removeKey;ifAbsent:
 removeLast

Implementing Classes

File
 FileHandle
 FileHandle
 FileHandle
 FileStream, Stream
 Process
 Float, Fraction, Integer, Number
 Behavior
 CharacterScanner
 Menu
 ScreenDispatcher
 Pen
 TopPane
 CharacterScanner, GraphPane, SubPane, TextPane,
 TopPane
 TopPane
 ListPane, TextPane
 ColorScreen, DisplayScreen
 ListPane
 Dos
 Dictionary, IdentityDictionary, Set
 Dictionary, IdentityDictionary, Set
 DispatchManager
 Collection
 Object, Pane
 Integer, Number
 DictionaryInspector, Directory
 Collection, Directory class, DispatchManager, File class
 Bag, Collection, Dictionary, FixedSizeCollection,
 OrderedCollection, Set
 Collection
 Dictionary
 Debugger
 Class
 DiskBrowser
 DiskBrowser
 OrderedCollection
 Class
 OrderedCollection
 Dictionary
 Dictionary, IdentityDictionary, MethodDictionary
 OrderedCollection

Method Selector

removeSelector
 removeSelector:
 removeSharedPool:
 removeSubClass
 removeSubclass:
 rename:
 rename:in:
 rename:to:
 renameFile
 replace:withChar:
 replace:withText:
 replaceAll
 replaceAllOld
 replaceAtColumns:by:
 replaceAtColumns:by:startAt:
 replaceAtPattern:by:
 replaceCrsIn:
 replaceFrom:to:with:
 replaceFrom:to:with:startingAt:
 replaceFrom:to:with:Object:
 replaceGapBefore:withChar:
 replaceLinesIn:with:
 replaceString:
 replaceWithChar:
 replaceWithLf:
 replaceWithTab:
 replaceWithText:
 reply
 reply:
 reset
 resetPrinter
 resize
 resize:
 reSort
 respondsTo:
 restart
 restartAt:
 restore
 restoreDirList
 restoreMode
 restoreSelected

Implementing Classes

ClassBrowser, ClassHierarchyBrowser, MethodBrowser
 Behavior
 Class
 ClassHierarchyBrowser
 Behavior
 Class
 Class
 File class
 DiskBrowser
 StringModel
 StringModel
 TextEditor
 TextPane
 TextPane
 StringModel
 TextPane
 DiskBrowser
 IndexedCollection, OrderedCollection
 Bitmap, ByteArray, IndexedCollection, String
 Bitmap, IndexedCollection, String
 StringModel
 StringModel
 TextPane
 TextPane
 StringModel
 StringModel
 TextPane
 Prompter
 Prompter
 CursorManager, NoMouseCursor, Pattern, Stream,
 WildPattern
 DiskBrowser
 TopDispatcher
 TopPane
 SortedCollection
 Object
 Debugger
 Process
 ListPane
 DiskBrowser
 DisplayScreen class
 ListPane

Method Selector

restoreSelected:
 restoreWithRefresh:
 resumable:
 resume
 resume:
 return:
 returnIndex:
 reverse
 reverse:
 reverseContents
 reversed
 reverseDo:
 reverseLine:
 right
 rightArrow
 rightButton:
 rightIcons:
 rightPartAfter:
 rounded
 roundTo:
 run
 runDemo
 save
 saveAs
 saveExit
 saveGraph
 saveImage
 scaleBy:
 scaleTo:
 scanForWordAt:
 scanner:
 scanZero:
 schedule
 schedule:
 scheduleWindow
 scroll
 scrollBarFini
 scrollBarIncludes:
 scrollBarInit
 scrollBarUpdate
 scrollDelay:
 scrollDownAt:

Implementing Classes

ListPane
 ListPane
 Debugger
 Debugger, DispatchManager, Process
 Process, ProcessScheduler
 PointDispatcher
 ListPane
 Form, Form class
 CharacterScanner
 Stream
 IndexedCollection
 IndexedCollection
 Menu
 CursorManager, Rectangle
 CursorManager class
 TerminalStream
 TopPane
 StringModel
 Integer, Number, Point, Rectangle
 Number
 DemoClass, DispatchManager
 ScreenDispatcher
 ScreenDispatcher, SystemDictionary
 DiskBrowser
 ScreenDispatcher
 GraphPane
 ScreenDispatcher
 Rectangle
 Rectangle
 StringModel
 StringModel
 Directory class
 ProcessScheduler
 DispatchManager
 Dispatcher
 CursorManager class
 SubPane
 SubPane
 GraphPane, SubPane
 SubPane
 ScrollDispatcher
 ScrollDispatcher

Method Selector

scrollHand:to:
 scrollLeft:
 scrollTopCorner:
 scrollUp:
 scrollUpAt:
 search
 searchBack
 searchBack:for:
 searchBackOld
 searchForActiveDispatcher
 searchForActivePane
 searchForLineToShow:
 searchFrom:for:
 searchInit
 searchOld
 seconds
 seconds:
 select

 select:
 selectAfter:
 selectAll
 selectAtCursor
 selectAtEnd
 selectBefore:
 selectDirectory:
 selectedString
 selectFrom:to:
 selectInstance:
 selection
 selection:
 selectLineAtCurrentSelection
 selectLines:height:
 selector
 selector:

 selectorMenu
 selectors
 selectors:
 selectTo:
 selectToBit:
 selectToCursor

Implementing Classes

ListPane, TextPane
 ListPane, TextPane
 ListPane, TextPane
 ListPane, TextPane
 ScrollDispatcher
 TextEditor, TextPane
 TextEditor, TextPane
 StringModel
 TextPane
 DispatchManager
 Dispatcher, TopPane
 ListPane
 StringModel
 TextPane
 TextPane
 Time
 Time
 Dispatcher, ListSelector, PointDispatcher,
 ScreenDispatcher, TopDispatcher, TopPane
 Collection, Dictionary, FixedSizeCollection
 TextPane, TextSelection
 TextPane
 GraphPane, ListPane, TextPane
 TextPane
 TextPane, TextSelection
 DiskBrowser
 TextPane
 TextPane
 Inspector
 ListPane, TextPane
 GraphPane, ListPane
 TextPane
 TextPane
 CompiledMethod, Message
 ClassBrowser, ClassHierarchyBrowser, CompiledMethod,
 Message
 ClassBrowser, ClassHierarchyBrowser
 Behavior, ClassBrowser, ClassHierarchyBrowser
 Menu
 TextPane, TextSelection
 TextSelection
 TextPane

Method Selector

selectToShifted
 selectWordAtCurrentSelection
 selfCopyToX:Y:
 senders

 sendersOf:
 sendFrame
 sendFrame:
 setBackground
 setClass:
 setCollection:
 setDate:
 setDispatchers
 setFont:
 setForeColor:backColor:
 setInstList
 setLimits

 setName:setDirectory:
 setOffsetX:Y:
 setPaletteRegister:to:
 setReg:to:
 setRegHigh:to:
 setRegLow:to:
 setSysFont:
 setToEnd
 setWidth:height:
 shallowCopy

 sharedPools
 sharedPools:
 sharedVariableString
 shiftRate:
 show
 show:
 show:from:at:
 showCurrentLine
 showDirectory
 showForm
 showGap
 showIcons

Implementing Classes

TextPane
 TextPane
 CharacterScanner
 ClassBrowser, ClassHierarchyBrowser, Debugger,
 MethodBrowser
 Behavior, SystemDictionary
 Process
 Process
 Animation
 ClassReader
 Stream
 File
 DispatchManager
 CharacterScanner
 CharacterScanner
 DictionaryInspector, Inspector
 FileStream, ReadStream, ReadWriteStream,
 WriteStream
 File
 CursorManager
 Dos
 Dos
 Dos
 Dos
 Font class
 ReadWriteStream, Stream, WriteStream
 DisplayScreen
 Behavior, Boolean, Character, Collection, Dictionary,
 Float, IndexedCollection, Integer, Object, Symbol,
 UndefinedObject
 Class, MetaClass
 Class
 Behavior
 Animation
 Icon
 Stream, TextEditor
 CharacterScanner
 Menu
 DiskBrowser
 GraphPane
 SubPane, TextPane, TextSelection
 TopPane

Method Selector

showPartialFile
 showSelection
 showSelection:to:
 showSelection:to:with:
 showSelectionFrom:to:
 showWindow
 sign
 signal
 significand
 sin
 singleStep
 sixteenLine
 size

 skip
 skip:
 skipTo:
 solidEllipse:aspect:
 sort:to:
 sortBlock
 sortBlock:
 sortBy:
 sortByDate
 sortByName
 sortBySize
 sortMenu
 source
 sourceCodeAt:
 sourceForm
 sourceForm:
 sourceIndex
 sourceIndex:sourcePosition:
 sourceOrigin:
 sourcePosition
 sourceRect:
 sourceString
 sourceString:
 sourceX
 sourceX:
 sourceY
 sourceY:

Implementing Classes

DiskBrowser
 ListPane, TextPane
 TextSelection
 TextSelection
 TextPane
 GraphPane, ListPane, TextPane
 Number
 Semaphore
 Float
 Float, Number
 Debugger
 Font class
 Bag, File, FixedSizeCollection, IndexedCollection,
 Interval, Object, OrderedCollection, Set, Stream,
 String
 Debugger
 Stream
 Stream
 Pen
 SortedCollection
 Class class, SortedCollection
 SortedCollection, SortedCollection class
 DiskBrowser
 DiskBrowser
 DiskBrowser
 DiskBrowser
 DiskBrowser
 DiskBrowser
 CompiledMethod
 Behavior
 BitBlt
 BitBlt
 CompiledMethod
 CompiledMethod
 BitBlt
 CompiledMethod
 BitBlt
 CompiledMethod
 BitBlt
 BitBlt
 BitBlt
 BitBlt

Method Selector

space
 species
 speed:
 speedSpace
 spiral:angle:
 splitPath:in:
 spread:from:by:spacing:direction:
 spreadFrom:
 to:
 width:
 startAt:
 mask:
 headSize:
 sqrt
 squared
 startPosition:endPosition:
 startUp
 status
 stepInterrupt
 storeOn:
 storeString
 strictlyPositive
 string
 string:
 stringCoordinate:
 stringHash
 stringIn:
 stringWidth:
 structure
 structure:
 subclass:
 instanceVariableNames:
 classVariableNames:
 poolDictionaries:
 subclasses
 subclasses:
 subclassOf:
 subdirectories
 subtractDate:
 subtractDays:

Implementing Classes

TextEditor, WriteStream
 Interval, Object, Symbol, SymbolSet
 Animation
 ScreenDispatcher
 Pen
 File class
 Form
 Form
 Float, Number
 Number
 OrderedCollection
 SystemDictionary
 Float class
 Process class
 Array, Association, Boolean, Character, Collection,
 Dictionary, FixedSizeCollection, Number, Object,
 String, Symbol, UndefinedObject
 Object
 Number
 StringModel
 StringModel
 TextPane
 String
 StringModel
 Font
 Behavior
 Behavior
 Class, UndefinedObject
 Behavior
 Behavior
 MetaClass class
 Directory
 Date
 Date

Method Selector

subtractTime:
 superclass
 superclass:
 superpane:
 suspendActive
 symbol
 symbolAt:
 systemDispatcher
 systemMenu
 systemPrimFor:
 tab
 tabStringAt:
 take:from:
 tan
 tell:bounce:
 tell:direction:
 tell:go:
 tell:goto:
 tell:place:
 tell:turn:
 tempAt:number:
 tempAt:number:put:
 tempCount
 template
 tempList
 tempValue
 text
 textMenu
 textMenuInit
 textModified
 textPane:
 timerInterrupt
 timesRepeat:
 timesTwoPower:
 to:
 to:by:
 to:by:do:
 to:do:
 today
 top
 topCorner
 topCorner:

Implementing Classes

Time
 Behavior
 Behavior
 Pane
 ProcessScheduler
 Behavior
 SymbolSet
 DispatchManager
 ScreenDispatcher class
 Compiler class
 TextEditor, WriteStream
 StringModel
 Debugger
 Float, Number
 Animation
 Animation
 Animation
 Animation
 Animation
 Animation
 Animation
 Process
 Process
 CompiledMethod
 ClassHierarchyBrowser
 Debugger
 Debugger
 ClassBrowser, ClassHierarchyBrowser, MethodBrowser
 DiskBrowser
 DiskBrowser
 TopPane
 StringModel
 Process class
 Integer
 Float, Number
 Number
 Number
 Number
 Number
 Date class
 Rectangle, TextSelection
 GraphPane, ListPane, TextPane
 ListPane, StringModel, TextPane

Method Selector

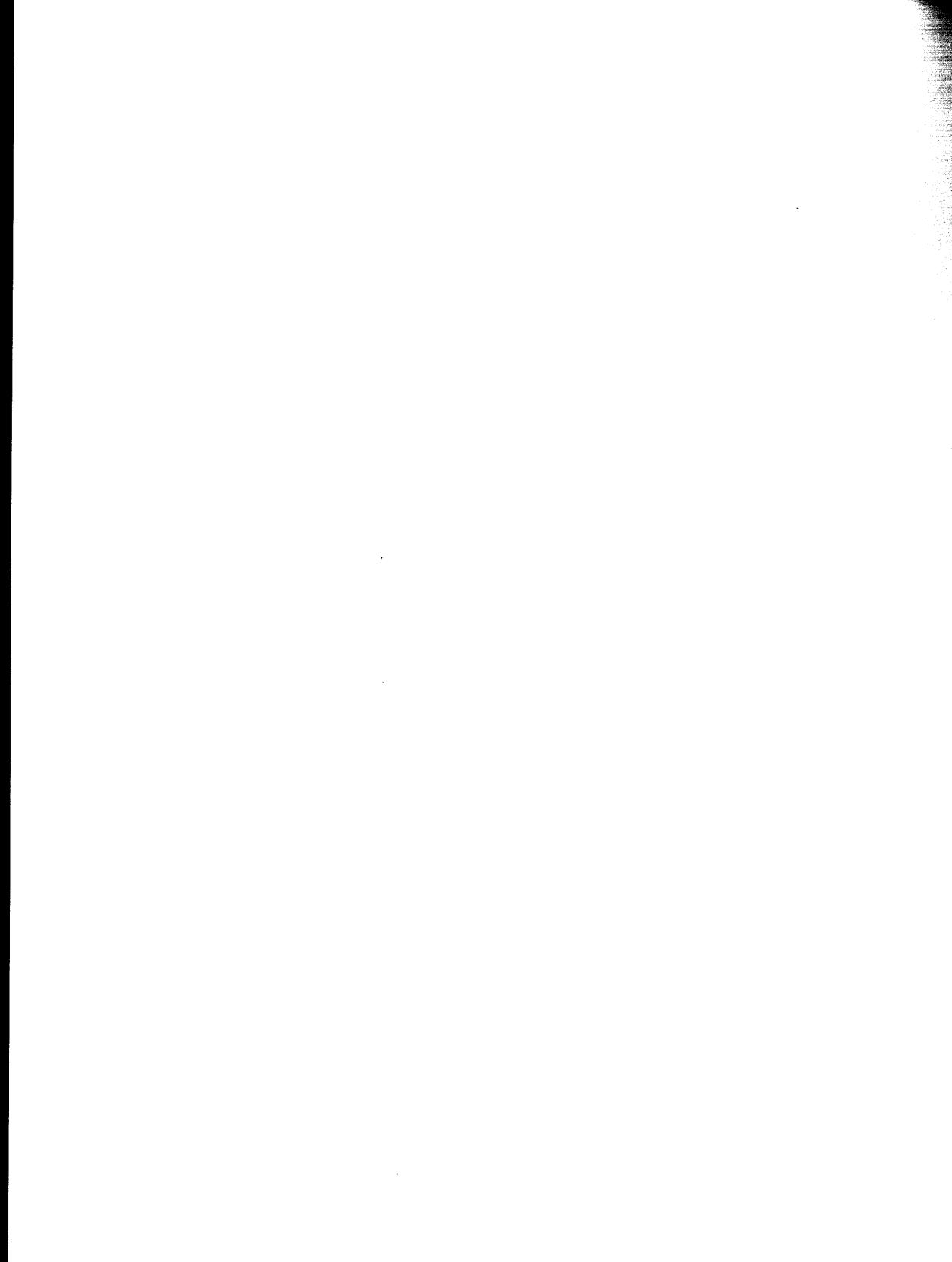
topDispatcher
 topPane
 topPriority
 toshiba
 totalLength
 totalSeconds
 transcriptMenu
 transientWriteFini:
 transientWriteOn:
 translateBy:
 transpose
 trimBlanks
 truncate
 truncated
 truncateTo:
 turn:
 type
 type:
 under
 unusedMemory
 unusedMemoryPrim
 unzoom
 up
 upArrow
 update
 update:
 update:with:
 update:with:with:
 updateBreaks
 updateLastByte
 updateSortPane
 upTo:
 userPrimFor:
 userPriority
 validateClass:
 instanceVariableNames:
 value
 value:
 value:value:
 values

Implementing Classes

Dispatcher, DispatchManager
 SubPane, TopPane
 ProcessScheduler
 DisplayScreen class
 GraphPane, ListPane, StringModel, TextPane
 Time class
 TopDispatcher
 DispatchManager
 DispatchManager
 Rectangle
 Point
 String
 ReadWriteStream
 Float, Fraction, Integer, Point, Rectangle
 Number
 Commander, Pen
 InputEvent
 InputEvent
 Form class
 SystemDictionary
 SystemDictionary
 Pane, TextPane class
 Commander, CursorManager, Pen
 CursorManager class
 ClassHierarchyBrowser, DiskBrowser, GraphPane,
 ListPane, SubPane, TextPane
 ClassHierarchyBrowser, Object, SubPane, TextPane,
 TopPane
 Object, SubPane
 Object, SubPane
 Debugger
 FileStream
 DiskBrowser
 Stream
 Compiler class
 ProcessScheduler
 Class
 Association, Context, InputEvent
 Association, Character class, Context
 Context
 Dictionary, IdentityDictionary

Method Selector	Implementing Classes
variableByteSubclass:	Class
classVariableNames:	
poolDictionaries:	
variableSubclass:	Class
instanceVariableNames:	
classVariableNames:	
poolDictionaries:	
VGA640x480	DisplayScreen class
vmInterrupt:	Object
volumeLabel	Directory
wait	Semaphore
walkback	Debugger
walkback:	Debugger
walkbackFor:label:	Debugger
walkbackLabel:	Debugger
walkbackMenu	Debugger
walkbackOn:	Process
walkLine	DemoClass
whileFalse:	Context
whileTrue:	Context
white	BiColorForm class, DisplayMedium, Form class, Pen
white:	DisplayMedium
width	BitBlt, DisplayObject, Font, Form, Icon, Rectangle
width:	BitBlt, Rectangle
width:height:	Form, Form class
width:height:initialByte:	ColorForm, Form
width:height:initialColor:	ColorForm
wildcardChar	Pattern class
windowClip	Pane class
windowClip:	Pane class
windowFrame	TopPane
windowLabeled:frame:	TextEditor class
with:	Collection class, FixedSizeCollection class
with:do:	IndexedCollection
with:with:	Collection class, FixedSizeCollection class
with:with:with:	Collection class, FixedSizeCollection class
with:with:with:with:	Collection class, FixedSizeCollection class
withAllSubclasses	Behavior
withBlank:	Prompter
withCrs	String
workSpaceMenu	TopDispatcher
write:	TerminalStream

<u>Method Selector</u>	<u>Implementing Classes</u>
writeBuffer:ofSize:atPosition:	File
writeFrom:toPage:for:	FileHandle
writeFrom:toPage:for:pageSize:	FileHandle
writeFrom:toPosition:for:	FileHandle
writeLimit	WriteStream
writePage	FileStream
Wyse640x400	DisplayScreen class
x	Point
x:	Point
xor:	False, True
y	Point
y:	Point
year	Date
yield	ProcessScheduler
yourself	Object
zapBackup	TopPane
zeroDivisor	Integer
zoom	TextEditor, TextPane, TopDispatcher, TopPane
	False, True
~ =	Integer, Object
~~	Object



INDEX

- abstract class, 79
- abstract data types, 68
- accept function, 139,277
- access.usr file, 500-502
- accessing,
 - files, 217
 - streams, 211,213
- activating a window, 30,32,264
- active process, 258-259,260
- active window, 30,32,264
- add breakpoint function, 307-308
- add function, 304
- add subclass function, 298,299
- add: message, 95,225,229
- addAll: message, 225
- adding
 - breakpoints, 308
 - classes,
 - 88,92-93,99,193-194,298-299
 - methods, 70,72,300,302-303
- addSubPane: message, 141
- allReferences message, 300
- and rule, 125-126,128,247
- and: message, 59,64-65,201
- animal,
 - classes, 82-83,149-151
 - habitat, 99-102,141-154,164,166
 - hierarchy, 80-82
- Animation, 136-138,146-154,254-255
- application,
 - case study, 161-183
 - class, 230-237
 - model,
 - 142,157-159,163-165,171-181,231-232,233-237
 - states, 233
- application development,
 - 155-161,161-184
- application development cycle,
 - 155-160,161-184
 - describing object states of, 155,157
- drawing window of,
 - 155,156,163-164
- identifying classes for, 155,157,165
- implementing methods of,
 - 155,159-160
- listing object interfaces, 155,158
- stating problem for,
 - 155-156,162-163
- application development tips,
 - 164,172,174,175,178,179,181,183-184
- arguments,
 - blocks as, 97,200,201,226
 - message, 7,8,46,47,48,68,195
 - method, 68,113,188-189
- arguments of block, 61,188-189,199-200
- arithmetic, 199,207-210
 - coprocessor, 48,210
 - floating point, 48,210
 - messages, 48,199,203,207
 - mixed mode, 207,208
 - point, 118,242
 - rational, 48,207,208-210
- Array, 88,94,188,196,198,223,228
- array, iterate, 62
- array literals, 45,88,198,223,224
- arrow key, 263
- ASCII, 130,204,250,283,289
- Aspect, 132-133,253-254,278
- aspect ratio, 132-233,253-254,278
- asSet message, 96
- assignment expressions, 7,52,187
- Association, 189,227
- at: message, 88,94
- at:put: message, 94,188,279
- atEnd message, 60,212,213,215
- automatic logging of changes,
 - 282-283,284,285,299,300
- available memory, 277
- background color, 122,130,250
- backing up Smalltalk/V, 284,285,286
- backspace key, 39,41,271,273
- Bag, 11,95,110,223,225,226-227
- ball, 31
- Bell, 280
- BiColorForm,
 - 120,122-123,124,244,247,248,255
- binary expression, 199

- binary message, 48-49,198-199,207
 binary selector, 199
BitBlt, 121-129,241,243-249
 creation, 248-249
 defaults, 248-249
 messages, 249
 subclasses of, 130-138,230,251
BitEditor, 292
bitmap, 117,120,241,244
 images, 250,289
bitmapped graphics, 117,241,245,289
black, 117,131
block, 51,58-65,97,195,199-200
 arguments of, 61,188-189,199-200
 as argument, 97,200,201,226
 execution, 97,200
 exit, 200
 invoking, 189
 sort, 230
blocked process, 258-259
Boolean, 79-80,201
 expressions, 8,59,201
border, window 26,263
breakpoints, 115,307
 adding, 308
 removing, 308
browse classes function, 32,44,79
browse disk function, 32,43,293
browse function, 301
browsers, 43,164,178,268,293
browsing, 43-44
 a class, 301-303
 classes, 44,69,297-301
 directories, 293,294-295
 disk, 32,43,293-297
 files, 295-296
 lists, 43,231
 methods, 300,302
Bs, 280
buffered files, 217
button, 26-27
 close, 27,33,47,140,142,266
 collapse, 27,267
 hop, 115,307
 jump, 115,307
 resize, 27-28,32,33,267
 skip, 115,307
 zoom, 27,39,178,267
buttons, mouse, 25,263
byte instance variables, 188,193-194
ByteArray, 223,228
cancel function, 277
caret, 9,52,64,72,198,200
carriage return, 63
cascaded messages, 50,198,199
change log,
 277,280,282-283,284,285,296,299,300
 and saving the image, 282,286
 compressing, 277,284,285
 format of, 282-283,286,296
change: message,
 159,178,233-234,235-236
changed: message, 149,236
changes, logging,
 282-283,284,285,299,300
changing,
 shape of pen, 131,252
 system font, 298
 the cursor, 144,291
Character, 196-197,204-205
character literals, 196,197,204
CharacterConstants, 278,280-281
CharacterScanner,
 121,130-131,220,250-251,289
class, 13,67,69,75,189-194,278
 abstract, 79
 application, 230-237
 changing definition, 179,299
 definition,
 69,89,99,193-194,296,299
 hierarchy, 15-17,79-80,190-191,297
 instances of, 13,67,187
 messages, 192-193
 methods, 89,194,298,300-301,302
 names, 278
 pane, 89
 pool for, 189
 protocol, 194

selecting, 298
 variables, 75,189,192,193,299
 inheritance of, 192
 Class Browser window, 231-237,264,301-303
 Class Hierarchy Browser, 32,35,44,69-70,72,79-80,171,193,293,297-301
 class message, 187
 classes,
 adding, 88,92-93,99,193-194,298-299
 browsing, 44,69,297-301
 defining, 14,81,172,193-194,299
 filing out, 298
 identifying application, 155,157
 removing, 298,299-300
 specifying, 193-194
 classifying objects, 67
 click, 264
 double, 38,79,272,293,307
 clipping rectangle, 129,245,247-248,250
 cloning, 287
 close button, 27,33,47,140,142,266
 close message, 217
 closing a window,
 27,33,140,142,266,268
 coprocessor, 48,210
 code, recycling, 162,165-175
 through borrowing existing code, 165-166,171-176
 through inheritance, 165-170
 collapse button, 27,267
 collapsing a window, 27,267,268
 collect: message, 64,96,226
 collecting garbage, 9-10,17-18,287,299-300,499,502
 Collection, 86,94-96,98-106,222-230
 attributes of, 223
 class hierarchy, 174
 common protocol, 222,225-226
 conversions, 224
 creating instance of, 224
 enumerating messages, 226
 fixed sized, 224,228-229
 variable sized, 224,228,229-230
 color, background, 122,130,250
 color, foreground, 122,130,250
 color of a pen, 131,132
 ColorForm, 120,121-123,124,241,244,247,255
 ColorScreen, 121,122-123,241,244,247,255
 combination rule, 125-129,245,246,247
 ComEvent, 165-170
 comma, 49
 command line, 500,505-508
 command templates, 276
 COMMAND.COM, 288-289
 Commander, 135,254
 comments, 53,174,197,276
 comparing messages, 57,58,203,204,206
 compatibleForm message, 255
 compatibleMask message, 255
 compilation errors, 40-41,276,300
 compiling methods, 70,300,301
 compressChanges message, 280,284
 compressing,
 the change log, 277,284,285
 the source file, 280,284-285
 compressSources message, 280,285
 computing letter pair frequencies, 98
 concatenation, 49
 conditional execution, 8,58-59,201
 conditional expressions, 57
 configuring,
 memory, 505-507
 Smalltalk/V, 505-508
 contents message, 92,212,213
 continue function, 275
 continuous scrolling, 38,270
 control structures, 57-65,200-201
 control-break, 111,305,306
 converting files, 218
 copy function, 295
 copy text, 30,72,171,273-274
 Cr, 218,281
 cr message, 140,215

create function, 295
creating,
 cursor shape, 292
 directories, 218,295
 file streams, 217
 forms, 120,244-245
 instances, 89
 menus,
 143-144,175,233-234,236-237
 methods, 70,72,300,302-303
 objects, 70,192
 panes, 233-235
 points, 117,208,241-242
 processes, 256,258
 rectangles, 118,243
 streams, 91,212
CurrentEvent, 219,278
CurrentProcess, 278
Cursor, 278,291
cursor, 25,144,263,278,290-292
 arrow keys, 263
 changing, 144,291
 displaying, 291-292
 hiding, 291
 hot spot, 292
 hour glass, 144,290,291
 moving, 25,263
 shapes, 290-292
 creating, 292
 prebuilt, 291
CursorManager, 290-292
cut text, 30,273-274
cycle windows function, 30,265,268
cycling,
 panes, 265
 windows, 265,268
data abstraction, 14
data structures, 14,189,222
 recursive, 87
Date, 70-71,205-207
deactivating a window, 30,32,264
dead process, 258-259
debug function, 111,112,306
debugger window,
 36,111-115,293,304,306-308
 label bar, 115,306-307
debugging, 107-116,259,304-308
 adding breakpoints, 308
 removing breakpoints, 308
 restarting execution,
 113,114,307-308
 resuming execution,
 111,306,307-308
 stepping through execution, 115,307
defining classes, 14,81,172,193-194,299
Del, 281
deleting text, 39,273
DemoClass, 74-75
demonstration program, 30-31,73-74
Dependents, 179
deselect text, 37,272
destination,
 bits, 246,247
 form, 121-123,129,245,247,248
 rectangle, 129,245,248
detect: message, 226
device driver interrupts, 261
Dictionary, 94-95,165,176-177,223,227
dictionary,
 pool, 167,189,193,227,280-281,299
 system, 189,277-281
dictionary inspector, 95,177,178,304
Directory, 216,217,218-219
directory,
 contents of, 296-297
 browsing, 293,294-295
 creating, 218,295
 selecting, 294
 Smalltalk, 11
disasters, 285-287
discrete event simulation, 257
Disk, 53,217,218,278
disk, free space, 294
Disk Browser, 32,34,43,44,293-297
 label bar, 294
disk devices, 218
DiskA, 218

DiskB, 218
Dispatcher,
 129,221,230,231-232,236,237,238-239
 classes, 231,238
DispatchManager, 141,239-240
Display, 121,244,250,252,255,278
 displayable bitmapped shapes, 250
 displaying,
 forms, 120,168-170
 strings, 250-251
 the cursor, 291-292
DisplayMedium, 224
DisplayObject, 224
DisplayScreen,
 120-121,123,241,244,255
do it, 30,40,276,295
do: message, 62-63,213
document retrieval system, 107
doesNotUnderstand: message,
 114,181,285
doing, 276
DOS,
 command processor, 288
 exit command, 288
 File System, 216-219
 menu, 288
 memory, 506
 reserving space for, 288,500
 shell, 288-289,500
dos shell function, 288
double click, 38,79,272,293,307
dragon, 135
drawing, 253
 lines, 249
 the window, 155,156
driver, mouse, 290
edit message, 301
editing,
 files, 295-296
 text, 271-274,276
eightLine message, 289
enableInterrups: message, 260
encapsulation of code and data, 13-14,17
Encyclopedia of Classs, 178
environment, 18,21-44
equality, 57,87-88,228
equivalence, 87-88,228
erase rule, 125-126,247
error recovery, 285-287
errors,
 compilation, 40-41,276,300
 runtime, 6,41-42,304-308
Esc, 281
evaluating an expression, 40,275-277
evaluation order, 8,48,49,198,199,207
events, reading, 219-220
example.prm file, 503
executeCommands: message, 288
execution state, 115
exiting,
 a block, 200
 a menu, 29,265
Smalltalk/V, 24-25,67,275
expressions, 194,198-199
 assignment, 7,52,187
 binary, 199
 boolean, 8,59,201
 classes in, 189
 conditional, 57
 evaluation of, 40,275-277
 keyword, 199
 message, 194,198-199
 nesting of, 59
 return, 9,52,59,175
 series of, 7,49-50,200
 unary, 199
extend selection, 271
extended memory, 506
factorial message, 46,49
factorial method, 71
false, 57-60,63,201
far call, 503
Ff, 281
Fibonacci method, 72
Fibonacci number, 72
File, 216,217,219
file out function, 298

- file streams, 109,216,217-218,219
 - accessing, 217
 - buffered, 217
 - converting, 218
 - creating, 217
 - iterating across, 62-63
- FileHandle, 216,219,249
- files,
 - browsing, 295-296
 - editing, 295-296
 - installing, 81-82
 - selecting, 295
- FileStream, 91,216
- filling out classes, 298
- fixdptrs.usr file, 499
- FixedSizeCollection, 223,228-229
- Float, 196,207,210
- floating point arithmetic, 48,210
- flush message, 217
- Font, 130,250,289-290
- forget image function, 25,275,286
- forgetting the image, 25,275,284,286
- foreground color, 122,130,250
- fork message, 256,257-258
- forkAt: message, 256,257-258
 - creation, 120,244-245
 - device type, 244
 - displaying, 120,168-170
 - messages, 197-198
- form,
 - destination,
 - 121-123,129,245,247,248
 - mask, 124-125,245,247,248
 - source, 121-123,129,247-248
- format a floppy disk, 288
- formatting of code, 50,51
- fourteenLine message, 289
- Fraction, 68,69-70,196,207-210
- frame function 30,33,268
- framing windows, 267,268
- framingBlock: message, 233-234
- framingRatio: message, 147-148,159
 - free space on disk, 294
 - function, 49
 - call, 46
 - definition, 68
 - name, 46
 - parameter 46
 - function keys, 220,221,278,281
 - FunctionKey, 220,221,278,281
 - FunctionPrefix, 281
 - garbage collection, 9-10,17-18,287,299-300,499,502
 - generalized iterators, 62-65
 - generic code, 79,85,96,204
 - global variables, 52-53,75,84-85,94,189,194,278,280
 - creating, 84-85,94
 - go file, 23,500
 - grab and pull, 270
 - graph panes, 147,268
 - graphic classes, 117-138,241-256
 - graphical program, 53-54
 - graphics, 103-138,241
 - graphics controllers supported, 2,22-23
 - GraphPane, 164,231,238
 - halftone form, 124-125,245,247,248
 - halt message, 111,303,304
 - hard disk, 21-23
 - hide/show function, 79-80,82,283,295,298
 - hiding a menu, 29,265
 - hiding the cursor, 291
 - hop button, 115,307
 - hot spot, 292
 - hourglass cursor, 144,290,291
 - I-beam, 25,37,39,271
 - identifier, 195
 - IdentityDictionary, 223,228
 - if statements, 8,58-59
 - if messages, 58-59,72,201
 - image, 274-275,283,286
 - file, 18,274,275,282-287
 - forgetting, 25,275,284,286
 - saving, 18,25,67,274-275,283-284,286,287

Image diskette, 2,21-22
implementors function, 301,302,307-308
implementors window, 308-309
implementorsOf: message, 280
includes: message, 95,225
includesKey: message, 94
indexed instance variables,
 71,76,140,188,193-194,303
IndexedCollection, 87,223,228
indexOfCollection: method, 86,88
indexOfString: method, 72-73,86
 information hiding, 13-14,68
 inheritance, 17,79-90,165-170,190,192
 of class variables, 192
 of instance variables, 81-82,192
 of methods, 83-84,192
inject: message, 226
InputEvent, 219-220
 inserting text, 39,271-272
 insertion point, 25,37,39,271
inspect function, 177,304,307
inspect message, 75-76,178,303
 inspecting dictionaries, 95,177,178,304
Inspector,
 35,75-76,95,113,178,293,303-304
install function, 286-287,296
 installation program, 23
 installing,
 a file, 81-82
 methods, 286-287,296,300,301
 Smalltalk/V, 21-23
instance, 13,67,187
instance pane, 89,99
instance methods, 194,300,301-302
instance variables,
 13,67,68,71,81-82,157,187,188,192,
 193,299,303
 byte, 188,193-194
 indexed,
 71,76,140,188,193-194,303
 inheritance of, 81-82,192
 named, 71,76,188,193-194,303
 pointer, 188,193-194
instances, creating, 188
 instances, lost, 179
 instances, removing, 179,300
Integer, 196,207,210
 interrupt service routines, 502-503
 interrupts, 260-261
 device driver, 261
 predefined events, 261
InterruptSelectors, 261
Interval, 208,223,228,229
 interval creation, 208
 invoking user primitives, 504
isEmpty message, 214,215,225
 iteration, 9,61-65,200,201
 iterative statements, 9,61-65
 iterators, generalized, 62-65
 iterators, simple, 61
 jump button, 115,307
keyboard,
 input, 219,230,238
 scan codes, 220
 using the, 26,29,37-39,263
KeyboardSemaphore, 219,259,279
keypad, 25,27,28,29,263-264
keys message, 280
keyword expression, 199
keyword messages, 47,49,198-199
label bar,
 26-28,32,33,39,40,47,115,238,263,264,
 266-267,294,307
label bar, Disk Browser 294
label function, 268
label: message, 233-234
LabelFont, 289
LargeNegativeInteger, 210
 precision of, 210
LargePositiveInteger, 210
 precision of, 210
 letter pair frequencies, 98
Lf, 218,281
 line delimiters, 217-218
 line drawing, 249
 line feed, 63
 list panes, 112,147,268,294
ListFont, 289

- ListPane, 164,178,231,238
lists,
 browsing, 43,231
 scrolling, 38-39,43,269-270
literals, 194,196-198,204
 arrays, 45,88,198,223,224
 characters, 196,197,204
 numbers, 196,207-210
 strings, 41,45,196,224
 symbols, 197-198,224
loadPrimitivesFrom: message, 500
loading primitives, 500
logging of changes,
 282-283,284,285,299,300
Logitech mouse, 290
looping messages, 50-51,60,201
lost instances, 179
macros, 500-502
Magnitude, 15,85,203-204
Magnitude class hierarchy, 203
magnitude classes, 15,203-210
maintaining Smalltalk/V, 281-287
mandala method, 132
mask bits, 246
mask form, 124-125,245,247,248
max: message, 85
memory,
 available, 277
 configuring, 505-507
 DOS, 506
 extended, 506
menu: message, 143,159,233-234
menus,
 28-31,143-144,164,175,236-237,
 265-266
 creating,
 143-144,175,233-234,236-237
 designing, 156,164
 DOS, 288
 exiting, 29,265
 hiding, 29,265
 modifying, 74
 pane,
 28,29,30,143-144,159,236-237,266
pop-up, 26,29,265-266
selecting from, 24,29,265
system,
 24,29,30,32,34,43-44,74,265
window, 27,29,30,238,265-266,277
message, 13,45-55,194-195,198-201
arguments, 7,8,46,47,48,68,195
expressions, 194,198-199
lookup, 83,192
names, 8
pattern, 200
protocol, 85,189
result, 52,195
selector, 13,46,47,48,195,305
send, 115
separator, 50
messages, 13,45-55,194-195,198-201
 arithmetic, 48,199,203,207
 binary, 48-49,198-199,207
 cascaded, 50,198,199
 class, 192-193
 evaluation order, 8,48,198,199,207
 keyword, 47,49,198-199
 looping, 50,51,60,201
 messages inside of, 49
 ordering, 57,58,203,204-207
 sending, 7,13,195
 simple, 46
 unary, 47,49,198-199
metaclass, 192-193
metalanguage definition, 195,491
Method Brower,
 280,293,300,301,308-309
Method Index, 178
method, 13,68-74,194-201
 arguments, 68,113,188-189
 implementation, 155,159-160
 inheritance, 83-84,192
 lookup, 83,192
 name, 68,195
 primitive, 200,495-504
 result, 9,52,59,72
 selector, 200
 source code, 69

specification, 200
 template, 89,300
 temporaries, 172,188,195
 walkback, 112,113,305-306,307
 methods, 13,68-74,189,194-201
 adding, 70,72,300,302-303
 browsing, 300,302
 class, 89,194,298,300,301-302
 compiling, 70,300,301
 generic, 79,85,96,204
 installing, 286-287,296,300,301
 instance, 194,300,301-302
 interim versions of, 175
 modifying, 301,302-303
 reinstalling, 282,284,286
 removing, 300,302
 selecting, 300,302
 Microsoft mouse, 290
 mixed mode arithmetic, 207,208
 mode function, 295
 model class,
 142,157-159,163-165,171-181,231-232,
 233-237
 model,
 pane, 142,159
 state transition, 161-162
 super, 140
 model: message, 142,159,233-234
 modifying methods, 301,302-303
 MonitoredArray, 88-90
 mouse,
 buttons, 25,263
 driver, 290
 input, 219,230,238
 using 2,24,25,29,37-39,272
 MouseButton, 281
 MouseEvent, 220,221
 moving a window, 33,267
 moving the cursor, 25,263
 multi-pane windows,
 145-154,171-176,178-181
 multi-windowed application, 240
 multiprocessing classes, 256-261
 name: message, 142,159,178,233-234
 named instance variable,
 71,76,188,193-194,303
 names,
 class, 278
 message, 8
 method, 68,195
 path, 216
 user-primitive, 503
 variable, 7,187,194,195
 nested data structures, 88
 nested expressions, 59
 Network, 102-106,134,164,165,166
 NetworkNodes,
 102,104-106,133-134,164,166-170
 new message, 192,224
 new method function,
 70,72,89,168,171,300
 new: message, 188,192,224,228
 next message, 60,212,213,215
 next: message, 214
 next:put: message, 215
 nextPut: message,
 60,92,212,213,215,221
 nextPutAll: message, 92,213,218
 nib, 252
 nil, 70,138
 nodes, network,
 102,104-106,133-134,166
 NoMouseCursor, 290
 Number, 207-210,229
 number radix, 196
 numbers, 196,207-210
 numeric,
 classes, 196,207-210
 functions, 209-210
 keypad, 25,27,28,263-264
 messages, 207-210
 precision, 210
 numerical methods, 207-210
 Object, 15,79-80,83,190-191,236,297
 object,
 checking class of, 499
 creation, 70,192
 displaying, 71

- header, 499
- interface list construction, 155,158
- pointers, 188-189,195,210,499
 - state, 51,155,157,188
 - state description, 155,157
- object.usr file, 499
- object-oriented development, 155-161
- objects,
 - 13,14,45-46,51,187-189,274,287
 - classifying, 67
- occurrencesOf: message, 225
- on: message, 91,212
- opaque rule, 127-128
- opening a window, 32,266
- operator precedence, 48,49,198,199,207
- or: message, 59,201
- or rule, 125-126,128,246,247
- order of evaluation, 8,48,49,198,199,207
- OrderedCollection, 223,229
- ordering messages, 57,58,203,204-207
- orThru rule, 125-127,247
- over rule, 125-127,247
- pane,
 - 26,28,140-143,145-148,230-237,238
 - changing contents of, 235-236,240
 - class hierarchy, 230,231,238
 - clean up, 149
 - creation, 233-235
 - cycling, 265
 - initialization, 142,235
 - menu,
 - 28,29,30,143-144,159,236-237,266
 - creating, 143,236-237
 - model, 142,159
 - position, 147-148,159
 - reinitialization, 149
 - size, 147-148,159
 - synchronization, 179,231,235-236
- panes, 263,268-270
 - defining new, 231
- graph, 147,268
 - list, 112,147,268,294
 - text, 147,268,275,294
- parallel processing, 256
- parentheses, 8,48,57,199
- Pascal, 6-11
- pasting text, 30,39,72,171,273-274
- path names, 216
- path name messages, 216
- Pattern, 97
- pattern matching, 72-73,86,97
- PC mouse, 290
- peek message, 214-215
- peekFor: message, 214-215
- Pen,
 - 70-71,121,131-134,135-136,251-254
 - changing shape, 131,252
 - color of, 131,252
 - creation of, 70-71,252
 - drawing with, 253
 - messages, 252-253
- PendingEvents, 259
- period separator, 50,198
- Point, 117-118,208,241-242
 - arithmetic, 118,242
 - comparing, 118
 - creation, 117,208,241-242
 - messages, 241-242
- pointer instance variables, 188,193-194
- pointers, object, 188-189,195,210,499
- polygon flower, 53-54,73-74
- polymorphism, 15,85-88
- pool dictionaries,
 - 167,189,193,277,280-281,299
- pool variables, 189
- pop up menu, 26,29,265-266
- position message, 92,213,215
- position: message, 92,213
- prebuilt cursor shapes, 291
- precedence, operator, 8,48,198,199,207
- precision, numeric, 210
- primitive number, 200,495
- primitive number assignments, 495-498
- primitive methods, 200,495-504
 - accessing objects within, 499
 - invoking, 504
 - loading, 500
 - macros for writing, 500-502

names, 503
 reserving space for, 500
 Smalltalk/V 286,495-498
 user-defined, 499-504
primitive modules,
 constructing, 503-504
primitive: message, 495
PrinterStream, 92-94
print function, 295
printOn: message, 216
printOn: method, 216
printString message, 15,211
printString method, 211-212
priority of processes, 256,258,260
private variables, 187,195
problem statement, 155-156,162
Process, 183,256,257,258,260
process priority, 256,258,260
process state transitions, 258-259
process,
 active, 258-259,260
 blocked, 258-259
 created, 256,258
 dead, 258-259
 forking, 256,258
 ready, 258-259,260
 signal, 257-258,259
 user interface, 259
 wait, 257-258
processing, parallel, 256
processFunctionKey: method, 221
Processor, 259,279
ProcessScheduler, 257,259-260
Prolog, 183
Prompter,
 11,42,74,139,239,276-277,299
prompting for input, 239
protocol, class 194
protocol, message 85,189
purging unused symbols, 287
queue operations, 229
Quick Tour-windows and menus, 33-36
radix, 196
random access, 216
rational arithmetic, 48,207,208-210
rational numbers, 210
read it function, 117,296
reading,
 events, 219-220
 streams, 213-215
reading streams, 213-215
ReadStream, 91-92,212-215
ReadWriteStream, 91,216,220
ready process, 258-259,260
recovery from crash, 281,285-287
Rectangle,
 117,118-119,216,241,242-243
 messages, 216,243
rectangle,
 destination, 129,245,248
 source, 121,129,245,247
rectangles, creation of, 118,243
recursion, 71-72,103-104
recursive data structures, 71,87-88
recycling code, 165-175
 through borrowing existing code,
 165-166,171-176
 through inheritance, 165-170
redraw screen function, 50,117
reinitialize message, 179
reinstalling methods, 282,284,286
reject: message, 63,64-65,96,226
relational operators, 58
remove breakpoint function, 307,308
remove class function, 298,299
remove function, 294,295,300,304
remove: message, 225
removeAll: message, 225
removing,
 breakpoints, 308
 classes, 299-300
 instances, 300
 methods, 300,302
removeKey: message, 280
rename function, 295
replacing text, 273

requesting input, 277
 reserving space, 500
 reset message, 213
 resize button, 27-28,32,33,267
 resizing a window,
 15,27-28,30,33,267,268
 restart cpu, 507-508
 restart execution, 113,114,307-308
 restore function, 273,304
 restoring text, 273
 resume execution, 111,306,307-308
 return expressions, 9,52,59,175
 return key, 21-22,42,139,271,277
 reusing code, 162,165-170
 reusing text, 37,43,276
 reverse rule, 125-126,128-129,247
 run demo function, 30-31
 runtime errors, 6,41-42,304-308
 SalesCom, 162-183
 sample.sml file, 276
 save function,
 70,76,89,113,172,177,304,307
 save as function, 296
 save image function, 275
 saving text, 273
 saving the image,
 18,25,67,274-275,283-284,286,287
 scan codes, 220
 scanning input, 211
 Scheduler, 141,239,279
 scheduling windows, 141,239
 screen, 244,264
 screen aspect ratio,
 132-133,253-254,278
 ScreenDispatcher, 288
 scroll bar, 28,38,269,270
 scroll cursor, 28,38,269,270
 scrolling lists, 38-39,43,269-270
 scrolling,
 continuous, 38,270
 grab and pull, 270
 horizontal, 38,269
 pause, 270
 quick jump, 269,270
 resume, 270
 speed of, 264,270
 terminate, 270
 vertical, 38,269
 with keyboard, 38,269-270
 with mouse, 269,270
 select: message, 63,64,96,226
 select: method, 96
 select key, 263,264,265,271
 selecting,
 a class, 298
 a directory, 294
 a file, 295
 a menu item, 24,29,265
 a window, 264
 from a list, 43
 large pieces of text, 39
 methods, 300,302
 selecting text, 37-38,271-273
 draw through, 272
 single characters, 272
 single lines, 38,273
 single words, 38,272
 using the keyboard, 37-38,271-272
 using the mouse, 37,272
 selector,
 message, 13,46,47,48,195,305
 method, 200
 self, 70,84,187,192,260,303,305,307
 Semaphore, 183,219,256,257-258
 semi-colon separator, 50,199
 send, message, 115
 senders function, 300,302,307-309
 senders window, 308-309
 sendersOf: message, 280
 sending messages, 7,13,195
 separator,
 message, 50
 period, 50
 semi-colon, 50
 series of expressions, 7,49-50,200
 series of statements, 7,49-50
 Set, 96,222,223,227
 setLoc message, 281

shared variables, 75,187,188,189,193,195
 shell, DOS, 288-289
 shift key, 263
 show it function, 30,40,46,53,139,276,295
 showing, 276
 shutdown, hardware 507-508
 shutdown, software, 507-508
 signal message, 257,260
 signalling a process, 257-258,259
 simple,
 iterators, 61
 loops, 50-51,60
 messages, 47
 objects, 45-46
 simulation, discrete event, 257
 single pane window, 140,141-145
 size message, 188,228
 skip button, 115,307
 skip: message, 213,215
 skipTo: message, 214-215
 SmallInteger, 210,228
 range of, 210
 Smalltalk, 189,277,278,279,500
 directory, 11
 expressions, 194,198-199
 methodology, 188
 vs conventional languages, 6-11
 Smalltalk/V,
 backing up, 284,285,286
 configuring, 505-508
 exiting, 24-25,67,275
 installing, 21-23
 maintaining, 281-287
 starting up, 23-24
 syntax summary, 491-494
 smalstalk.bat, 288
 sort blocks, 230
 SortedCollection, 17,98,223,230
 sorting, 230
 source bits, 246,247
 Source diskette, 2,21-22
 source file, 18,280,282,284-285
 compressing, 280,284-285
 source form, 121-123,129,247-248
 source rectangle, 121,129,245,247
 Sources, 279
 Space, 281
 space/speed function, 508
 space message, 215
 space mode, 508
 specifying classes, 193-194
 speed/space function, 508
 speed mode, 508
 stack operations, 229
 starting up Smalltalk/V, 23-24
 state,
 execution 115
 object, 51,155,157,158
 state transition model, 161-162
 statement separator, 7
 statements,
 assignment, 7,52,53
 if, 8,58-59
 iterative, 9,61-65
 series of, 7,49-50
 states, application, 233
 stating the problem, 155-156
 Stream, 91-94,212-216
 class hierarchy, 91,211
 protocol, 213-216
 streams, 91-94,212-216
 accessing, 211,213
 creating, 91,212
 growing, 211
 positioning, 92,211,213-215
 random access, 211
 reading, 213-215
 testing for end of, 211
 writing, 215-216
 String,
 72,94,196,211-212,222,223,224,228
 string literals, 41,45,196,224
 StringModel, 231
 strings, displaying, 250-251
 strings, iterate, 62

subclass, 17,79-81,88,190-193
 subclass class type, 92,193-194
 SubPane, 231,238
 subscripted variable access, 8
 super, 84,192
 super model, 240
 superclass,
 79-80,81,83,84,86,88,190-193,299,305
 Symbol, 196,197-198,223,224,228,287
 symbols, 197-198,224
 symbols, unused, 287
 syntax summary, Smalltalk, 491-494
 SysFont, 279,289
 system dictionary, 189,277-281
 system menu, 24,29,30,32,34,43-44,265
 SystemDictionary, 277,281,500
 System Transcript,
 34,53,84,94,179,216,264,279
 Tab, 281
 tab key, 271
 tab message, 215
 template, method, 89,300
 temporary variables,
 51-52,113,172,188,195,200
 Terminal, 220,279
 terminal input and output, 219-222
 TerminalStream, 91,219-221,264
 testing messages,
 58,203,204-205,208-209
 text,
 copying, 30,72,171,273-274
 cutting, 30,273-274
 deleting, 39,273
 deselecting, 37,272
 edit buffer, 273-274
 editing, 271-274,276
 editor, 69,76,271-274,277
 inserting, 39,271-272
 pasting, 30,39,72,171,273-274
 replacing, 273
 restoring, 273
 reusing, 37,43,276
 saving, 273
 selecting, 37,38,271-273
 text insertion point, 25,37,39,271
 text pane zoom, 27,39-40,178,267
 text panes, 147,268,275,294
 TextEditor, 140
 TextFont, 289
 TextPane, 140,164,177,178,231,238
 Time, 70-71,205-207
 timesRepeat: message, 60
 tips, application development,
 164,172,174,175,178,179,181,183-184
 to dos function, 288
 TopDispatcher, 238,239
 TopPane, 141,231,238
 Transcript,
 34,53,84,94,179,216,264,279
 true, 57-60,63,201
 Turtle, 50-51,53,55,60
 turtle graphics, 131,251-253
 tutorial files, 44
 unary expression, 199
 unary message, 47,49,198-199
 UndefinedObject, 70
 under rule, 125-126,247
 unused symbols, 287
 unusedMemory method, 280
 update function, 82,93,108,294,298
 UpperToLower, 281
 upTo: message, 214-215,218
 user-defined primitives,
 user interface process, 259
 and debugging, 259
 value messages, 97,188-189,200,201
 variable names, 7,187,194,195
 variableByteSubclass class type, 193-194
 variableSubclass class type, 89,193-194
 variable types, 187-188
 variables, 15,187-189,195
 class, 75,189,192,193,299
 global,
 52-53,75,84-85,94,189,194,278,
 280
 indexed instance,
 71,76,140,188,193-194,303

instance,
 13,67,68,71,81,82,157,187,188,
 192,193,299,303
kinds of, 187-188
named instance,
 71,76,188,193-194,303
pool, 189
private, 187,195
self,
 70,84,187,192,260,303,305,307
shared, 75,187,188,189,193,195
super, 84,192
temporary,
 51-52,113,172,188,195,200
video adapters supported, 22-23
wait message, 257,259
walkback list, 112,113,305-306
walkback window,
 36,41-42,46-48,51,83,110-112,173,179,
 293,304-306
walkback, method, 112,113,305-306,307
while messages, 9,60,73,201
white, 117,131
window, 26-28,230-240
 border, 26,263
 button, 26-27
 classes, 230-240
 default size, 146,148
 label bar,
 26-28,32,33,39,40,47,115,238,
 263,264,266-267,294,307
 menu, 27,29,30,238,265-266,277
 size, 146,148
windows, 26-28,139-154
 activating, 30,32,264
 active, 30,32,264
 closing of, 27,33,140,142,266,268
 collapsing, 27,267,268
 cycling of, 265,268
 deactivating, 30,32,264
 debugger,
 36,112-115,293,304,306-308
 drawing, 155,156
 framing, 267,268
labeling, 268
moving, 33,267
multi-pane,
 145-154,171-176,178-181
multiple in application, 240
opening, 32,266
resizing, 15,27-28,30,33,267,268
scheduling, 141,239
selecting, 264
single-pane, 140,141-145
walkback,
 36,41-42,46-48,51,83,110-112,
 173,179,293,304
WordIndex, 107-110
Workspace, 263,266
WriteStream, 91-93,211-213,215
writing streams, 215-216
zoom button, 27,39,178,267
zooming of text pane, 27,39-40,178,267